Rectify: Black-Box Intrusion Recovery in PaaS Clouds

David R. Matos   Miguel L. Pardal   Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal
{david.r.matos,miguel.pardal,miguel.p.correia}@tecnico.ulisboa.pt

Abstract

Web applications hosted on the cloud are exposed to cyberattacks and can be compromised by HTTP requests that exploit vulnerabilities. Platform as a Service (PaaS) offerings often provide a backup service that allows restoring application state after a serious attack, but all valid state changes since the last backup are lost. We propose Rectify, a new approach to recover from intrusions on applications running in a PaaS. Rectify is a service designed to be deployed alongside the application in a PaaS container. It does not require modifications to the software and the recovery can be performed by a system administrator. Machine learning techniques are used to associate the requests received by the application to the statements issued to the database. Rectify was evaluated using three widely used web applications – Wordpress, LimeSurvey and MediaWiki – and the results show that the effects of malicious requests can be removed whilst preserving the valid application data.

CCS Concepts **• Computer systems organization** *Maintainability and maintenance*;

Keywords PaaS, Rollback, Recovery, Intrusion Removal

ACM Reference format:
David R. Matos   Miguel L. Pardal   Miguel Correia. 2017. Rectify: Black-Box Intrusion Recovery in PaaS Clouds. In *Proceedings of Middleware '17, Las Vegas, NV, USA, December 11–15, 2017,* 13 pages.
https://doi.org/10.1145/3135974.3135978

1 Introduction

Platform as a Service (PaaS) is a cloud computing model that allows easy deployment of elastic web applications in public clouds [11, 29, 39, 40]. In a PaaS offering, a system administrator is able to configure and maintain complex applications without the burden of managing the full software stack in a set of servers. Automatic scaling allows having stable throughput even when dealing with high peaks of traffic. The PaaS model provides more versatility to run user applications than the Software as a Service (SaaS) model, and easier management than the Infrastructure as a Service (IaaS) model [29]. Three well-known examples of PaaS offerings are: Red Hat OpenShift [35], Google App Engine [10], and Force.com [43].

As with any other application, those running in PaaS services may also be attacked and have their state illegitimately modified. PaaS offerings allow developers and system administrators to focus on the application, as they tend to hide most of the complexity underneath. However, developers/administrators can introduce implementation and configuration vulnerabilities, which may allow adversaries to execute successful attacks. Moreover, web applications are known for being plagued by a variety of vulnerabilities that allow compromising their state in the database, e.g., authentication and session management issues, cross-site request forgery, and SQL injection [38, 44].

When an adversary illegitimately modifies the state of a web application running on a PaaS offering, its administrator may have to recover the state by rolling it back to a point in time before the intrusion, e.g., by restoring a backup or by using a checkpointing mechanism. This approach removes the effects of the intrusion, but will most likely discard many legitimate state modifications that occurred after the intrusion. The latest correct state is usually old, meaning that a large portion of valid data is lost.

Older work used logs and snapshots to recover databases [12], but there is a more recent line of research concerned with recovering from requests (operations) at the application layer, not at the data layer. This line of research aims to *remove the effects of intrusions* (and mistakes) from system or application state without discarding all modifications after the intrusion [1, 6–8, 14, 20, 32]. For instance, the administrator may want to recover from an illegitimate request with data filled in a web form that is inserted in the database using several SQL statements. He wants to undo the effect of the request which implies undoing all the statements, and possibly undoing any statements that used this data afterwards.

The problem of web application recovery has been studied in the past decade [1, 7, 8], including more recently web applications running in PaaS [30]. Although these services are capable of removing the effects of intrusions in web applications, they require modifications to the applications source code, i.e., to support intrusion recovery an application has to be modified to support the recovery service. Such modifications are needed at least to allow mapping requests to database statements, e.g., for the recovery service to understand that a certain type of request causes a certain update statement to the database. This is a significant drawback. First, in some cases the organization does not even have access to the source code of the application. Second, it may be difficult to find skilled programmers with adequate knowledge about how the application was built to safely implement the required modifications. Finally, the cost, in terms of time and money, of implementing the required modifications for the recovery system have to be considered, since most companies usually run under constrained budgets and tight schedules.

In this paper we propose a new approach that allows undoing the effects of intrusions in web applications *without any software modifications*, thus considering that *the application is a black-box*, solving this limitation of previous works. Rectify leverages a log of requests (i.e., application operations) and a log of database statements. Given an illegitimate operation executed by the application, Rectify is capable of finding its effects in the database and automatically remove them. As far as we know, this is the first recovery tool that allows a system administrator to identify malicious requests at the application level and automatically remove the intrusions at the database level without modifying the application.

Rectify was designed to be deployed in PaaS offerings. It is installed as an add-on module that can be attached to the web application through the PaaS administration console. It works with both open source and proprietary applications. Rectify recovers an application by executing compensation operations at the database, which undoes the statements that were issued by malicious HTTP requests. The detection of intrusions and the identification of illegitimate requests may be done automatically using an intrusion detection system (IDS) [19, 23, 31, 36] or by the administrator, e.g., by searching in the request log for any requests coming from a suspect IP address. Either way this problem has been widely studied for a couple of decades and is different from the problem of intrusion recovery, so we do not aim to solve it in this paper.

The main challenge we had to solve was the association between the requests received by the application and the statements it issues to the database. By modifying the application it would be possible to annotate the statements with additional information about the request that caused them [30], but we want to avoid such modifications. Another solution would be to inspect the two logs after running the application for a while and to write a set of rules that associated the statements to requests. However, this would be a tedious and error-prone procedure.

Our solution uses *supervised machine learning* [22] algorithms to automatically find correlations between HTTP requests and the issued database statements. A learning program analyses the two logs obtained during a teaching phase and produces two *classifiers* that are able to do the association automatically during runtime. With this technique it is possible to setup Rectify to different applications without the effort of modifying the code of the application or writing rules to associate requests to statements. Our technique requires the web application to be implemented using good practices, i.e., the URLs have to follow a clear and predictable structure. Rectify does not understand Web applications that use URL parameters to load different pages, since it assumes that the parameters are simply variables of the request.

We implemented Rectify in Java and configured it as a container ready to be deployed in PaaS offerings. This implementation is available online.[1] We run it in the Google App Engine [10]. Rectify was evaluated using three widely used web applications – Wordpress, LimeSurvey and MediaWiki – and the results show that the effects of malicious requests can be removed whilst preserving the valid application data. Moreover, they show that it is possible to undo one malicious request in less than one minute.

The main contributions of this paper are the following: (1) an approach for recovering from intrusions in PaaS applications without deleting all modifications since the intrusion and without modifying the applications source code, i.e., considering the application is a black-box; (2) a novel service that implements this approach, provided as a container easily deployable in a PaaS offering; (3) a machine learning scheme to automatically associate database statements to the request that generated them.

2 Platform as a Service

Platform as a Service is one of the three original cloud computing models, alongside IaaS and SaaS [29]. This model supports automated configuration and deployment of applications in data-centers. These applications are typically web applications, i.e., software that runs in web servers, backed by databases, and that communicates with browsers using the HTTP protocol. PaaS offerings normally support elastic web applications that scale horizontally by deploying more virtual machines when there is an increase in demand.

Figure 1 presents the basic architecture of a PaaS platform. The *application delivery controller* (ADC) is responsible for directing client requests to the adequate servers, based on the application being accessed and the server load. Applications are deployed in a virtualized environment called a *container*, e.g., in a Linux container provided by Docker [34]. Containers are logically isolated from other containers using mechanisms such as *cgroups* (or control groups), which allow limiting the resources used (CPU, I/O, etc.), and *namespaces*, which provide an abstract layer for names of resources (processes, users, network interfaces, etc.). Each container has a runtime environment that depends on the application. For example, a Java EE application container can include a Tomcat

[1] https://github.com/davidmatos/Rectify.git

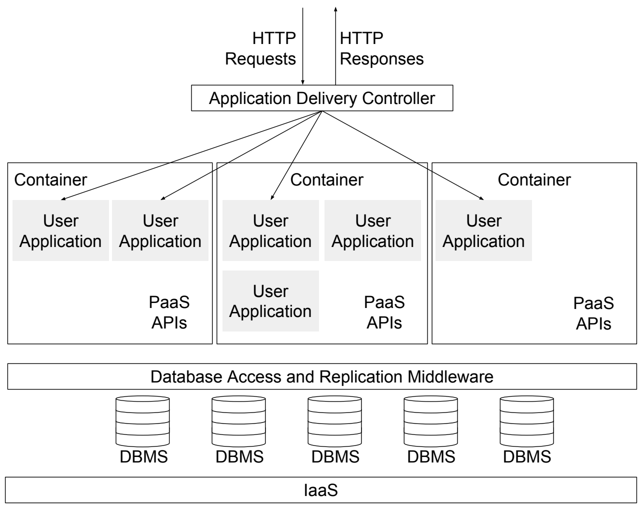
HTTP Requests
HTTP Responses
Application Delivery Controller
Container
User Application
PaaS APIs
Database Access and Replication Middleware
DBMS
IaaS

Figure 1. Typical PaaS Architecture.

server with a Java Virtual Machine; for PHP applications the container includes an Apache 2 server with the Zend engine. Containers provide APIs for applications to access the functionality provided by the PaaS environment. An important example are the APIs for accessing the data layer, i.e., the *database management systems* (DBMSs). These APIs interact with the *database access and replication middleware* that hides the complexity of interacting with databases that often are replicated in several physical servers.

In this paper we consider the following main players. A *user* is an individual or a company that owns and deploys applications in the PaaS platform. The *clients* are the individuals who access applications deployed in the PaaS. The *system administrator* is the responsible for the configuration of an application running in a PaaS offering.

3 Rectify

Rectify denominates both an approach and a service for undoing the effects of intrusions in web applications. The objective is for the service to be deployed at PaaS offerings without modifications to the source code of the applications. Rectify is concerned with the *integrity* of applications' state, not with data *confidentiality*.

3.1 System Model

A web application that is configured to be recoverable by Rectify is called a *protected application*. A protected application receives HTTP requests (at the application level) from its clients. These HTTP requests generate database statements that alter the state of the application by inserting, deleting or updating database records. Although we frequently mention HTTP, the requests may equally be received over HTTPS.

The state of the protected application becomes corrupted when it receives a *malicious request* (at the application level) which in turn generates *malicious statements* (at the database level). A malicious request is any request that is illegitimate for some reason, e.g., because it is issued by someone who should not have access to the application (e.g., a hacker). We consider that all statements caused by a malicious request are themselves malicious. Nevertheless, *select* queries do not tamper the state, as they only read data and do not modify the database, so there is no need to remove their effects.

The way Rectify recovers an application consists in *undoing* malicious requests by undoing malicious statements. We use the term *undo* [6] because after recovery the state of the application is intended to be such as if the malicious operation never took place in the past. One of the ways Rectify undoes malicious operations is by executing a special operation, called *compensation operation* [21], which removes the modifications that resulted from a malicious operation.

3.2 Threat Model

Our threat model considers that the *state* of a protected application can be tampered only by malicious requests. In other words, we assume the computational infrastructure of the PaaS and of the Rectify service are not compromised by adversaries. This does not mean that such problems may not occur – they can – only that we do not consider them in this paper as we focus on recovering the state of the application.

3.3 System Architecture

Rectify contains two sets of components: HTTP and DB. The HTTP components are in charge of intercepting and logging the HTTP requests issued to the application. The DB components play a similar role for the database, i.e., they intercept and log the statements the application issues to the database. The Rectify service was designed to be deployed in a different container than the application.

The logs, as well as the configuration values of Rectify, are stored in a *database management system* (DBMS). Having the logs in a DBMS makes it easier to do searches and perform complex queries with multiple criteria in order to find malicious HTTP requests. Keeping old logs allows us to recover from malicious operations that took a long time to detect. However, logs grow indefinitely with time. A solution is to move old parts of the log to a data archival service (cold storage) in the same cloud (e.g., AWS Glacier [41]). This allows keeping the logs for longer periods for a fraction of the price (e.g., in AWS, the cost for normal storage is around \$0.02 per GB per month, whereas in Glacier it is only around \$0.005 per GB per month).

Figure 2 shows the architecture of Rectify. The user runs the protected application in Container 1 and Rectify in Container 2. During normal operation, when a client sends a request to the protected applications it is forwarded to the HTTP proxy (arrow *a* in the figure). This proxy logs the client request in the HTTP log and redirects it to the protected application (arrow *b* in the figure). Every time the protected application issues a statement to the database the statement is

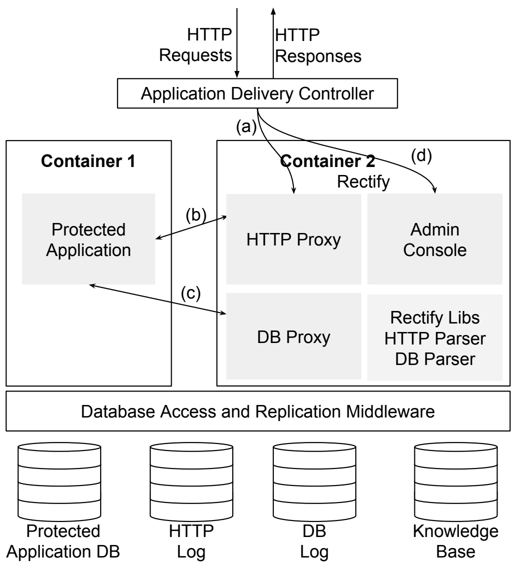

Figure 2. Rectify system architecture. Rectify itself is the set of components inside Container 2.

intercepted and logged by the DB proxy (arrow *c* in the figure), which is configured with the address and access credentials of the database used by the application. The administrator accesses the recovery service using the *admin console* (*d*), after providing his authentication credentials (e.g., password).

Next we present these in more detail.

HTTP proxy: reverse proxy 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.

HTTP log: data-store used by the proxy to keep every request. It records: the entire payload of the HTTP request, the address of the host that triggered the request, a timestamp and the URL of the request. Each request stored in the log is called a *HTTP log entry*.

DB proxy: component responsible for intercepting every statement to the database and logging it in the DB log. It only saves the statement in the log if the execution returned without errors. Invalid statements are not recorded since they do not have to be recovered.

DB log: data-store that saves every successfully executed database statement that changed the state of the database. Besides the executed statements, the DB log also saves a timestamp, the identifier given by the DB proxy and the primary key of the affected records. Each statement stored in the log is called a *DB log entry*.

Admin console: component that allows the system administrator to manage Rectify and to recover the protected application. It provides a graphical user interface with a search engine to navigate and find incorrect operations in both the HTTP and the DB log. It also provides an interface to run the learning phase, so it can later successfully correlate an HTTP request with the corresponding DB queries.

Rectify libs: software libraries required to parse requests and database statements. These libraries are used by the remaining components of the service. The main libraries are the HTTP parser and the DB parser. The *HTTP parser* breaks requests into parts so that the classification model is able to analyze these parts and associate the request with the database queries. The parts extracted by the parser are: HTTP method, URL, timestamp, names and values of parameters, and the host that issued the request. The *DB parser* breaks a SQL statement into parts to identify the signature of the statement. Some of these parts will be found in the HTTP request that generated the statement. The relevant parts that identify a SQL statement are: statement type (*insert*, *select*, etc.), table affected by the statement, columns used by the statement and timestamp. The HTTP parser and the DB parser are used by Rectify to analyze the requests and queries in two phases: during the training phase, when the knowledge base is being constructed; and during the recovery process, when Rectify needs to find the queries issued by the malicious requests.

Knowledge base: In order to correlate HTTP requests with their corresponding database operations, a collection of examples (HTTP request – database statements) is necessary. These examples are stored in a data-store called the knowledge base. A detailed explanation of how the knowledge base is loaded can be found in Section 4.

4 Rectify Learning Phase

Rectify uses supervised machine learning to find relations between the faulty HTTP request and the corresponding database statements. Rectify considers that the application is a *black-box*, so it observes HTTP requests and DB statements and finds the relations between them without looking into the application code or requiring modifications to that code. As any other supervised learning algorithm, it is necessary to provide Rectify with samples or examples. Each sample allows Rectify to learn that a specific HTTP request will generate a certain kind of database statement.

Each example in the knowledge base is identified by an application *route*. A route is a URL pattern that is mapped to a resource of the web application. For example, the following URL `www.app.com/posts/welcome` is derived from the route `www.app.com/posts/{title}`. This route in particular points to a web page that displays a post with the title *welcome*. By analyzing the route, there is a fixed part (`www.app.com/posts/`) and a variable part (`title`) which is replaced by the title of the post.

The loading of the knowledge base with samples is done by setting the operation mode of Rectify to *learning* and let it execute a list of all the *routes* of the application. Rectify will then automatically execute each HTTP request for each route in the list, one at a time. While this happens, both the

Table 1. Example of a signature record

	Feature	Example
HTTP	Method	GET
	URL	/posts/new_post.php
	Nr. of parameters	2
	Parameters	[title, content]
	Values	[t, c]
Nr. of statements		2
SQL1	Type	UPDATE
	Nr. of columns	1
	Columns	{id}
	Values	{1}
	Tables	[users]
SQL2	Type	INSERT
	Nr. of columns	2
	Columns	[title, content]
	Values	[t, c]
	Tables	[posts]

HTTP request and the database statements generated will be captured and stored in the knowledge base. For Rectify to learn which database statements are issued by the HTTP request being executed each HTTP request has to be executed at a time, with some delay before executing the next one; if there were HTTP requests being executed simultaneously, it would be hard to know to which request corresponded the database statements.

Rectify is assisted by a *web crawler* to discover all routes, as it is crucial to execute every possible route of the application in the learning phase. A route that is not learned by Rectify cannot be undone later, since there is no information in the knowledge base that allows to later recognize the database statements issued by the HTTP request. Also, routes that are not learned by Rectify will generate database statements that cannot be associated to any HTTP request and, as explained in Section 5.3, these statements will be marked as *suspected*. The crawler systematically browses the web application and identifies all existing requests and the correspoding routes. It is similar to other web crawlers [18], although it obtains information of a single application, not of a large portion of the web.

The knowledge base contains one association between each HTTP request and a set of database statements. Each entry in the knowledge base is denominated a *signature record* and is divided in parts. For example, consider the URL:

```
r = /posts/new_post.php?title=t&content=c
```

that generates the three database statements:

```
SELECT name FROM users WHERE id = 1
UPDATE users SET ts = NOW() WHERE id = 1
INSERT INTO posts VALUES (t, c)
```

Table 1 presents the signature record of the HTTP request `r`. The record is divided in HTTP and SQL parts: an HTTP request part; SQL parts for each statement (SQL1 and SQL2) issued by the HTTP request that modifies the database. The column *feature* contains the name of each part and the column *example* contains an example of such feature based on the example given above.

5 Two-Step Classification

In order to identify the database statements issued by a malicious HTTP request, Rectify needs to solve two classification problems. The first, *signature matching*, consists in identifying the signature record of the malicious HTTP request. The second, *DB statements matching*, consists in finding in the DB log the actual statements that were created by the malicious HTTP request.

Figure 3 presents a level-0 data flow diagram with the tasks performed in these two classification problems. In the figure, the system administrator queries the HTTP log for a malicious request. This malicious request generated a set of database statements that corrupted the database. Given the malicious HTTP request, Rectify will first solve a classification problem to identify the signature record of such HTTP request (*signature matching*). Once Rectify has the signature record, in other words, once it knows the kind of application request, it is able to obtain the database statements that may be issued by that HTTP request. This is only possible because the knowledge base contains examples of the HTTP requests and corresponding database statements. The second step uses the obtained database statements to find the malicious database statements present in the DB log. Both steps are explained in more detailed in the following sections.

5.1 Step 1 - Signature Matching

In the first step, Rectify solves the classification problem of identifying the signature record of an HTTP request. Figure 4 presents a data flow diagram describing the tasks done to identify the signature. In the figure, a malicious HTTP request, given by the system administrator, is parsed in order

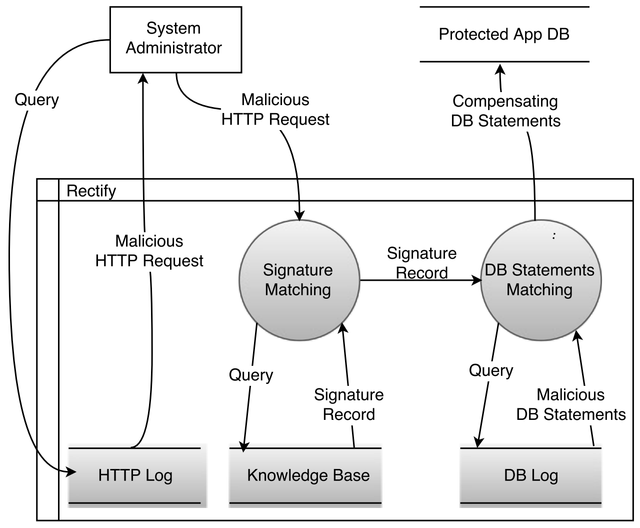

Figure 3. Level-0 data flow diagram of the tasks performed to solve the 2-step classification problem.

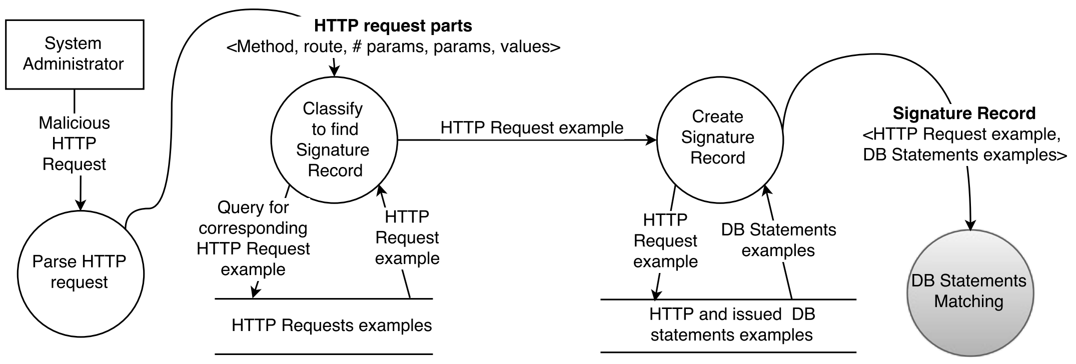

Figure 4. Level-1 data flow diagram with the tasks performed to identify the signature record of a malicious HTTP request.

to extract its relevant parts (shown in Table 2). Using the parts of the malicious HTTP request, the classification algorithm is executed to find the corresponding signature record (a pair containing an example of an HTTP request and its corresponding database statements) in the knowledge base.

The classification algorithm works in two phases: first, the knowledge base, which is structured by features, is loaded by a machine learning algorithm. This algorithm runs a training phase in which it creates a model, based on the features, that defines a classifier. Later, when an HTTP request needs to be identified, it is processed by the classifier which will predict the most likely class for that HTTP request. In this case the class is a signature record. A more detailed explanation of how classifiers work can be found in [22]. We did not implement a new classification algorithm to solve this problem because we found that there are several well studied algorithms that are capable of solving this problem.

5.2 Step 2 - DB Statements Matching

In second step, the list of database statements issued by the malicious HTTP request is identified. Using this signature, obtained in the first step, it is possible to find the corresponding database statements issued by the malicious request. Figure 5 presents the various tasks needed to find the malicious database statements. As shown in the figure, first a set of generic database statements is calculated. This process is described in detailed in Algorithm 1.

Algorithm 1 begins by getting all the database statements of the signature record (line 2). Then, each statement from the signature record is cloned (line 3). The rationale behind this step is that the database statements generated by *mr* should be very much like the ones in the signature record, except for the values that should come (in part) from the HTTP request. Then it verifies in each column if the value comes from the HTTP request (line 5). If so, then it will modify *p_stmt* to include the values from *mr* (lines 6 to 8). If the value did not come from the HTTP request, then that column is set to `NULL` (line 10). Finally, the calculated database statement is added to *GenericStmts* (line 13) and returned (line 15).

Algorithm 1 Calculates an approximation of the DB statements issued by *mr* (malicious request) based on *sr* (signature record).

```
    INPUT mr // malicious HTTP request
    INPUT sr // signature record
    RETURNS GenericStmts // candidate list of DB statements generated by mr
 1: GenericStmts ← ⊥
 2: for stmt ∈ sr.getDBStatements() do
 3:     p_stmt ← stmt
 4:     for col ∈ p_stmt.getColumns() do
 5:         if sr.valueComesFromHTTPRequest(stmt, col) then
 6:             attr ← sr.getHTTPAttrFromDBCol(col)
 7:             value ← mr.getValue(attr)
 8:             p_stmt.setColumnValue(col, value)
 9:         else
10:             p_stmt.setColumnValue(col, NULL)
11:         end if
12:     end for
13:     GenericStmts ← GenericStmts ∪ p_stmt
14: end for
15: return GenericStmts
```

Table 2. Features used to classify HTTP requests

ID	Feature	Description
C1-1	Method	GET, POST, PUT, etc.
C1-2	URL	The address
C1-3	Nr. of param.	Number of parameters
C1-4	Parameters	Attributes' names

With this set of generic statements it is possible to solve a classification problem in which we want to find, in the DB log, the database statements most similar to the generic statements. This classification problem is solved using the features listed in Table 3.

The classification problems presented in Sections 5.1-5.2 can be solved using several classification algorithms. In our work we studied several types of algorithms for this problem: *logistic regression*, *naive Bayes*, *decision tree*, *k-Nearest*

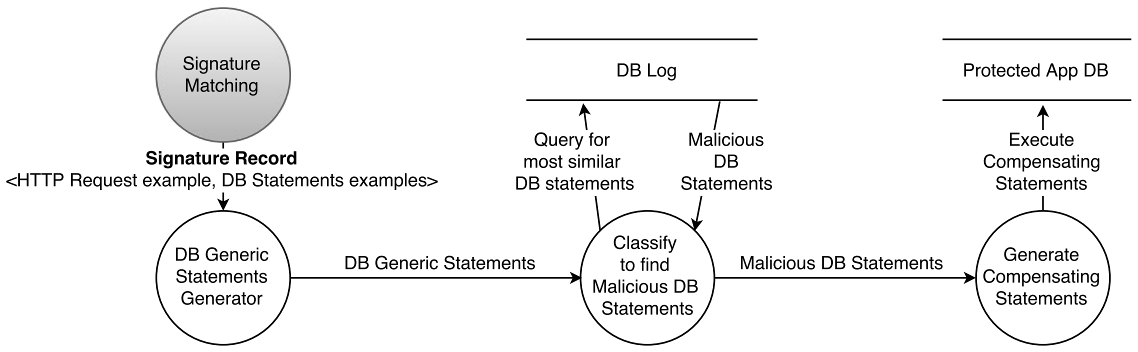
Signature Matching
Signature Record
<HTTP Request example, DB Statements examples>
DB Generic Statements Generator
DB Generic Statements
DB Log
Query for most similar DB statements
Malicious DB Statements
Classify to find Malicious DB Statements
Malicious DB Statements
Protected App DB
Execute Compensating Statements
Generate Compensating Statements

Figure 5. Level-1 data flow diagram with the performed tasks to identify the malicious database statements issued by the malicious HTTP request and generate compensation statements.

Table 3. Features used to classify database statements

ID	Feature	Description
C2-1	Statement type	*select*, *insert*, etc.
C2-2	Nr. columns	The number of columns in the statement
C2-3	Columns	The column names
C2-4	Values	Values of the columns
C2-5	Tables	Tables' names

Neighbors and many others. A full list of the studied algorithms is presented in Section 8.

5.3 Dealing with SQL Injection

One of the most common attacks against web applications is SQL injection [15].[2] In this kind of attack, an attacker is able to illegally inject database statements by sending an HTTP request containing special SQL characters, e.g., `OR 1 = 1#`. Such an attack would create corrupted database statements that are not present in the knowledge base, so they cannot be identified using the approach we have presented. It is not possible to teach Rectify SQL injection examples as the possibilities are unlimited.

To cope with this issue, Rectify does not look directly for the database statements caused by the malicious HTTP request, but for the statements that were not caused by the non-malicious HTTP requests received more or less at the same time. Specifically, Rectify searches the HTTP log for the set R_{good} of all requests received in the period $[t_0 - T_{max}, t_0 + T_{max}]$ excluding the malicious request, where t_0 is the instant when the malicious request was received and T_{max} an estimate of the maximum time it takes to execute the last statement caused by any request. Then, Rectify does the two steps of Sections 5.1 and 5.2 for all the requests in R_{good}, and obtains a set S_{good} of all statements issued to the database in consequence of these requests. Then it extracts from DB log the set of all statements S issued in the interval $[t_0, t_0 + T_{max}]$. Finally, it identifies as caused by the malicious HTTP request all the statements in S that are not in S_{good}.

[2]Despite the term SQL in the name, all data access languages based on statements are vulnerable to such attacks including NoSQL statements [28].

6 Recovery with Rectify

Rectify removes the effects of an incorrect statement from the database by calculating a set of database statements that undo what the malicious statement corrupted. These statements are called *compensation operations* or *compensation transactions* [21]. In a simplistic scenario in which tables have no relations and a statement that affects a record does not affect other records, we know that in order to undo an *insert* it is necessary to execute a *delete*; to undo an *update* it is necessary to *update* the record back to its previous value; and to undo a *delete* to execute an *insert* with the latest value. However, this problem becomes more difficult to solve in relational DBMSs. This kind of database allow the existence of relations between records (defined by foreign key columns that link to records of different tables). It is not recommended (in some cases it is not even allowed) to remove a record that is related with other records. It could happen that a record that was created by a malicious statement is related to different records that are to be kept. This means that deleting the malicious record would make the database inconsistent. A variant of this problem was already investigated [2, 26].

We based our approach to deal with dependencies on the work presented in [2]. In this work an algorithm to calculate a graph of dependencies among transactions is presented. Rectify uses this algorithm to calculate the graph of dependencies which will be used to execute the compensation operations. However, an attack can cause transitive effects that are visible in the application level. For instance, a record that was created by a malicious HTTP request can be later read by another, non-malicious HTTP request, which propagates it. Currently Rectify does not include a mechanism to track the transitive effects of attacks. A mechanism similar to Shuttle's dependency graphs might be added for this purpose [30].

The algorithm to calculate and execute the compensation operations is the *two pass repair* algorithm [2] modified to allow undoing single statements instead of only undoing complete transactions. It works as explained next.

Undoing an insert: An *insert* can be undone by deleting the malicious record. Rectify will produce an equivalent delete in the form, `DELETE FROM table-1 WHERE pk = PK`. As explained in Section 3.3, `PK`, the primary key value, is stored alongside every insert, update and delete operation. By using the primary key, Rectify does not affect valid records with its compensation operations. After calculating the *delete* statement of the malicious record, it is necessary to calculate the equivalent *delete* statements for the dependent records that were calculated earlier. To do so, Rectify will calculate a *delete* operation for each depended record created that references the faulty record.

Undoing an update: An *update* can affect several records. Since Rectify stores the primary key of each affected record in the log, it is possible to reconstruct each affected record separately. For each affected record, Rectify will obtain from the log every operation that affected it. From the insert that create it until the very last update. Rectify only queries the log for valid operations; the faulty ones are discarded. Then, Rectify can build a record in memory, i.e., without actually executing statements in the database. When every valid operation of the log that affected that record was executed in memory, Rectify will have a version of the record that corresponds to a valid record that was never affected by any faulty operation. Finally, Rectify executes an *update* to set every column of the affected record with the values of the valid record calculated in memory.

Undoing a delete: A *delete*, like an *update*, may affect several records. Rectify undoes a *delete* the same way it undoes an *update*. It first collects a lists of all affected records. Then it reconstructs, in memory, the record. At the end it will have the record, the way it was before it was removed by the faulty operation. Finally, Rectify will execute an *insert* with the record calculated in memory.

Undoing a drop table: A *drop table* operation is recovered in a different way. Instead of reconstructing each record in memory, Rectify will execute every valid statement that targeted the deleted table. Reconstructing every affected record in memory would require too much memory.

Undoing a drop database: The *drop database* is undone in the same way as a drop table. Every valid operation in the log is executed on the database.

Note that for undoing the *update*, the *delete* and the *drop* statements, we assume that every statement was present in the log. That may not be true, since as it was explained in Section 3.1, some statements may be so old that they are not present in the logs anymore. In this case Rectify will ask the system administrator to provide the old log archive in order to retrieve the most valid recent version of the affected records. Then Rectify is able to reconstruct the affected database records on top of that version.

In summary, at setup time, Rectify requires a system administrator to perform a learning phase. Then, when an attack occurs, the administrator indicates to Rectify a list of malicious HTTP requests that need to be undone. Rectify then uses its two-step classification in order to find the malicious database statements in the DB log. Finally, Rectify calculates the compensation statements that undo the effects of the attack. It is possible to execute the compensation statements in an application that is either online or offline. However, if the application is online, users may observe inconsistencies in the application state.

7 Implementation

This section describes our implementation of Rectify and how it can be deployed in a PaaS offering.

7.1 Rectify

To evaluate our proposal we implemented Rectify as an application ready to be deployed in a PaaS container. All the components of the system were written using Java Enterprise Edition (JEE), making it easy to deploy in any PaaS that supports Java. The HTTP and database logs, as well as the knowledge base of Rectify, are stored in a MySQL database. By using a relational database it is possible to easily search for entries in the log and store relationships between requests and responses.

Our implementation supports web applications that use a MySQL database, but it is simple to extend for different DBMSs. Table 4 lists the number of classes and lines of code of each module in our implementation.

Table 4. Number of classes and lines of code of our implementation of Rectify

Module name	Number of classes	Lines of code
Common Classes	3	1898
DBParser	1	287
DBProxy	1	244
HTTPProxy	1	224
HTTPParser	1	342
Recovery	3	1203
Machine Learning	2	895

7.2 Deployment in a PaaS

Rectify was designed to be deployed in a PaaS offering. All its components are installed in a single container. The data-stores containing the DB and the HTTP logs are also deployed in the same container. This approach gives two main benefits: *modularity*, Rectify is an additional module to the protected application and can be easily added to an existing PaaS configuration; *automatic scaling*, in terms of the data-stores containing the DB and HTTP logs as well as the HTTP and DB proxies that intercept every request and operation. This way, Rectify does not introduce a bottleneck in the application. We deployed Rectify in the Google App Engine [10] PaaS offering, which provides JEE containers.

8 Experimental Evaluation

With our experiments we wanted to answer to the following questions: (a) What is the cost, in terms of performance, of using Rectify with real world web applications? (b) How much space does Rectify require to store its logs and knowledge base? (c) How much time does it take to recover a web application in different scenarios? (d) How accurate are the algorithms used in steps 1 and 2 of the classification?

To evaluate Rectify we setup three web applications that are diverse in terms of the executed database statements and functionality: *Wordpress* [4], *LimeSurvey* [25] and *MediaWiki* [3]. *Wordpress* is a widely-adopted content management system (CMS) that provides a blog and a news page, as well as user registration. These features are interesting to test with Rectify since the state of a *Wordpress* application includes articles, posts and comments that are dependent between them. For instance, an article and a blog post are written by a user and can be commented by a group of users. The comments themselves can also be commented by users. Rectify needs to take this into consideration when calculating dependencies in order to ensure consistency of the database. *LimeSurvey* is a survey application that allows users to create and answer polls. Unlike *Wordpress*, *LimeSurvey* stores database records (with the polls and respective answers) that tend to be smaller than the records of a CMS[3]. This in turn will generate different HTTP requests which contain less information to work with. With this application we will assess if Rectify is capable of calculating correlations with less information. *MediaWiki* is similar to *Wordpress* in the sense that both applications can do content management. However, *MediaWiki* was designed to be open, meaning that all content is accessible to all users, while in *Wordpress* and *LimeSurvey* some contents may be only visible and editable by a restricted group of registered users. In both *Wordpress* and *LimeSurvey*, executed database statements store information about the user that performed them. If a malicious user performs an attack it is possible to query the log for every operation that he performed. Since *MediaWiki* allows unregistered users to perform modifications to the database, it is impossible to know the actions taken by an attacker.

8.1 Performance Overhead

Rectify uses two proxies to intercept HTTP requests and database statements respectively. These components impose an overhead to the web application, since every request needs to be logged before being handled. To evaluate the performance overhead of both proxies, we setup Wordpress, LimeSurvey and MediaWiki in separate containers of the Google App engine. Then, for each application we setup Rectify in a different container. Each container was a n1-standard-8 (8 vCPUs, 30 GB memory). Every container was setup in the same geographic zone (Western Europe) so that the network latency would not affect the results. Then, in a separate container, we executed a workload using JMeter [16] with 1,000 concurrent users, with each user issuing 1,000 requests (chosen randomly from a list of 10 HTTP URLs for each application). At the end of the experiments, 1,000,000 HTTP requests were issued in each application. JMeter is a testing framework that collects statistics about the execution of complex workloads. It is a Java application which can be deployed and executed in a PaaS container.

[3]Blogs and articles are usually records with hundreds or thousands of words

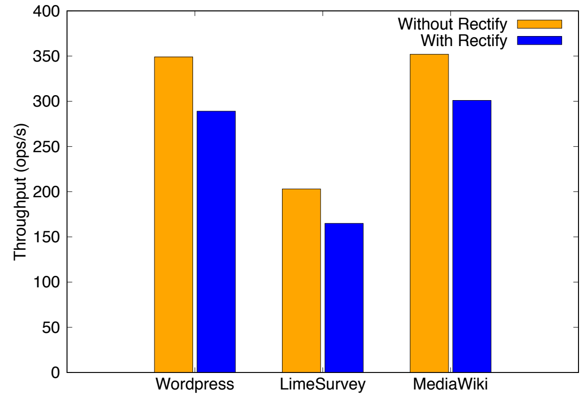

Figure 6. Performance overhead of using Rectify with three different applications measured in operations per second.

Figure 6 shows the overhead of using Rectify in all three applications. The results are shown in requests per second. The first bar of each group represents the throughput of the application without Rectify and the second bar represents the throughput of each application with Rectify logging the HTTP requests. There is a performance degradation between 14% to 18% in using Rectify to log the requests. This is expected since every request needs to be logged first before being resolved. This overhead might be reduced if our interception code was made more efficient, as we did not make a big effort to optimize it. Moreover, we consider the present overhead is acceptable for many applications, given the benefit provided by the service.

8.2 Space Overhead

Over time the space occupied by the logs of Rectify will grow. We wanted to know how much space does Rectify need to store the logs of a certain amount of requests. After the experiments presented in Section 8.1, we checked how much space the logs were taking in the database (uncompressed). Table 5 lists the space occupied by Rectify logs. After one million requests the log size varies from 5.13GB, for LimeSurvey, to 8.2GB, for MediaWiki. It is a considerable amount of space but since the data in the log is only read to perform recovery, it is viable to consider to store the logs in an external cloud storage service, such as Amazon S3 [33]. This way the scalability of Rectify would be managed automatically and the cost would be lower than storing the logs in the PaaS container.

Table 5. Space occupied by the Rectify logs in each application after 1,000,000 HTTP requests.

Application	HTTP Logs (GB)	DB Logs (GB)	Total (GB)
Wordpress	5.50	1.76	7.26
LimeSurvey	3.60	1.53	5.13
MediaWiki	7.00	1.20	8.20

8.3 Total Time to Recover

The total time to recover (TTTR) is the time elapsed between the moment the system administrator clicks the button *recover*, after selecting the malicious operations to undo, and the moment the application is recovered. TTTR varies depending on the size of the log and the number of incorrect operations to undo. To understand how the TTTR varies in these different scenarios we used the WordPress, MediaWiki and LimeSurvey applications that were deployed in the containers presented in the previous section. Then we injected 1,000,000 HTTP requests in each application. Finally, we performed several recovery operations by undoing sets of HTTP requests, from a single request to 1,000 in intervals of 100. Each experiment was repeated 10 times.

Figure 7 shows the average time to recover each set of requests with the standard deviation. The time to recover grows linearly with the number of operations to undo. Undoing a single request took an average of 12 seconds while undoing 1,000 requests took around 16 minutes.

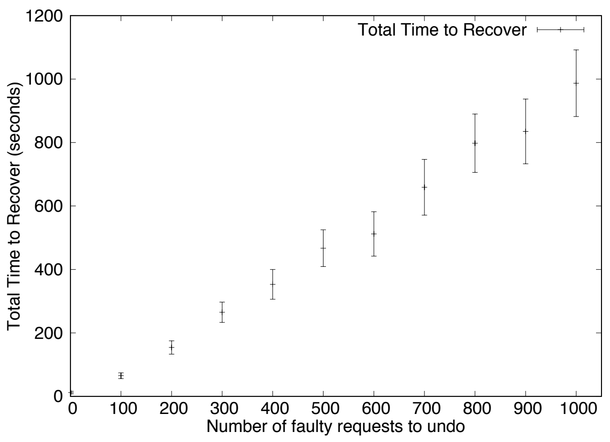

Figure 7. Total time to recover the protected application.

8.4 Accuracy of the Classification Algorithms

There are several algorithms that can be used to solve the classification problems presented in Section 5. We executed every classification algorithm available in Weka [17], using the default configurations. Weka is a machine learning tool which provides several well-known classification algorithms and data mining tools. We used different datasets of all three applications. Each dataset was modified by us to include the results that should be given by the classification algorithms, this way it is possible to calculate the accuracy, which is measured by the total of instances well classified and is given by: $Accuracy = (TP + TN)/(P + N)$, where P and N correspond to the positive and negative classes given by the Weka algorithms and TP and TN correspond to the true positive and true negative classes that should have been given by the classifiers.

8.4.1 Signature matching accuracy

For the signature matching we collect a dataset with 3,000 HTTP requests, 1,000 requests for each application. These datasets contain the classes and the identifier of the signature record that should be assigned by each classification algorithm. The HTTP requests were again generated by JMeter. In our experiments JMeter executed a list of 10 URLs randomly until it reached a list of 1,000 requests per application. These 10 URLs are a representative number of the types of requests that can be made using the APIs of the applications. For instance, in WordPress there are about 15 different operations, however, the 10 URLs used in this experimental evaluation are the most relevant ones. Then, every available classification algorithm of Weka was executed using the default configuration values.

The accuracy results of each algorithm are listed in Table 6. The first column lists the names of the algorithms, the second column the percentage of correctly classified HTTP requests and the third column the percentage of incorrectly classified HTTP requests.

There are several algorithms that reach 100% accuracy for the three studied applications (Wordpress, LimeSurvey and MediaWiki). Analysing the results we concluded that a reason for such good performance is that these applications were built taking into account good software development practices, namely the routes (URLs patterns of the several web pages) used in the application obey to strict rules, such as, the first part usually corresponds to a controller of the application, the second part to an action and the remaining parts to the parameters. These rules help the classification algorithms to correctly identify the requests.

8.4.2 Signature matching with irregular routes

To understand the difference with other applications that do not follow such good practices, we tested the signature matching algorithm in a sample application created by the Yii2 [37] framework. Yii2 is a PHP framework that allows to implement PHP applications with the Model View Controller (MVC) paradigm. The sample application created by Yii2 contains user registration, blog features and a set of static web pages. By default Yii2 does not use strict rules for the routes. For example, the route

```
www.app.com/controller/action/?p=user-login
```

points to the login page, whereas the route

```
www.app.com/controller/action/?p=contact
```

Table 6. Accuracy of the tested classification algorithms performing the first classification problem.

Algorithm	Correct	Incorrect
bayes.BayesNet	100%	0%
bayes.NaiveBayes	100%	0%
bayes.NaiveBayesMultinomialText	26%	73%
bayes.NaiveBayesUpdateable	100%	0%
functions.Logistic	100%	0%
functions.MultilayerPerceptron	100%	0%
functions.SimpleLogistic	100%	0%
functions.SMO	100%	0%
lazy.IBk	100%	0%
lazy.KStar	100%	0%
lazy.LWL	100%	0%
meta.AdaBoostM1	57%	42%
meta.AttributeSelectedClassifier	100%	0%
meta.Bagging	100%	0%
meta.ClassificationViaRegression	100%	0%
meta.CVParameterSelection	26%	73%
meta.FilteredClassifier	100%	0%
meta.IterativeClassifierOptimizer	100%	0%
meta.LogitBoost	100%	0%
meta.MultiClassClassifier	100%	0%
meta.MultiClassClassifierUpdateable	100%	0%
meta.MultiScheme	26%	73%
meta.RandomCommittee	100%	0%
meta.RandomizableFilteredClassifier	100%	0%
meta.RandomSubSpace	100%	0%
meta.Stacking	26%	73%
meta.Vote	26%	73%
meta.WeightedInstancesHandlerWrapper	26%	73%
misc.InputMappedClassifier	26%	73%
misc.SerializedClassifier	100%	0%
rules.DecisionTable	100%	0%
rules.JRip	100%	0%
rules.OneR	100%	0%
rules.PART	100%	0%
rules.ZeroR	26%	73%
trees.DecisionStump	56%	43%
trees.HoeffdingTree	100%	0%
trees.J48	100%	0%
trees.LMT	100%	0%
trees.RandomForest	100%	0%
trees.RandomTree	100%	0%
trees.REPTree	100%	0%

Table 7. Accuracy of the tested classification algorithms performing the second classification problem

Algorithm	Correct	Incorrect
bayes.BayesNet	100%	0%
bayes.NaiveBayes	100%	0%
bayes.NaiveBayesMultinomialText	72%	28%
bayes.NaiveBayesUpdateable	34%	66%
functions.Logistic	54%	46%
functions.MultilayerPerceptron	100%	0%
functions.SimpleLogistic	100%	0%
functions.SMO	100%	0%
lazy.IBk	100%	0%
lazy.KStar	100%	0%
lazy.LWL	100%	0%
meta.AdaBoostM1	3%	97%
meta.AttributeSelectedClassifier	100%	0%
meta.Bagging	100%	0%
meta.ClassificationViaRegression	100%	0%
meta.CVParameterSelection	54%	46%
meta.FilteredClassifier	100%	0%
meta.IterativeClassifierOptimizer	100%	0%
meta.LogitBoost	100%	0%
meta.MultiClassClassifier	100%	0%
meta.MultiClassClassifierUpdateable	100%	0%
meta.MultiScheme	87%	13%
meta.RandomCommittee	100%	0%
meta.RandomizableFilteredClassifier	100%	0%
meta.RandomSubSpace	100%	0%
meta.Stacking	54%	46%
meta.Vote	50%	50%
meta.WeightedInstancesHandlerWrapper	39%	61%
misc.InputMappedClassifier	72%	28%
misc.SerializedClassifier	0%	100%
rules.DecisionTable	0%	100%
rules.JRip	0%	100%
rules.OneR	0%	100%
rules.PART	100%	0%
rules.ZeroR	28%	72%
trees.DecisionStump	19%	81%
trees.HoeffdingTree	0%	100%
trees.J48	0%	100%
trees.LMT	0%	100%
trees.RandomForest	0%	100%
trees.RandomTree	0%	100%
trees.REPTree	0%	100%

points to a contact form. This characteristic, which may be present in other web applications, makes the classification algorithm identify both routes with the same signature record. This happens because the parameter p is treated as a parameter and not as the name of the page. After running every classification algorithm in this application we reached an accuracy below 20%. This issue might be solved by configuring Yii2 to use strict routes or performing an additional URL normalization step to make for better URLs.

8.4.3 DB Statements matching accuracy

The second classification problem, DB statements matching, can also be solved using existing machine learning classification algorithms. To evaluate which algorithms are the best to use in this problem, we generated a dataset with 1,000 HTTP request per application. In this dataset we added an *id* to identify the dependencies between the requests and the statement. Then we executed every available classification algorithm from Weka and compared how it classified the requests with how it was supposed to classify them. Table 7 lists the classifications algorithms (first column) and their accuracy rate (second and third columns). There are several algorithms that correctly identify 100% of the database statements issued by a malicious HTTP request.

8.5 Identifying SQL Injection Statements

As mentioned in Section 5.3, SQL injection attacks generate database statements that are not learned by Rectify during the learning phase and, therefore, cannot be identified by the DB statements matching algorithm. This creates a problem, since these statements are clearly malicious and need to be removed from the database. Our proposal to solve this problem consists in identifying, not only the database statements issued by a malicious HTTP request but also every database statement issued in a time interval around the instant the malicious HTTP request was received. This would result in some database statements not being identified as being issued by any HTTP request, which would mean that they were injected.

To evaluate this approach we tested the DB statements matching algorithm with OWASP WebGoat [13]. WebGoat is a Java Web application developed by OWASP which provides several kinds of vulnerabilities. It is mainly used to study the security of web applications. One class of attacks that can be performed in WebGoat is SQL injection.

In the experiments we exploited the following SQL injection attacks: string SQL injection, parameterized SQL injection and numeric SQL injection. For each attack we executed 2 examples. In every example, our DB statements matching algorithm was capable of identifying the injected statements as suspected since they were not issued by any known route of the application.

9 Related Work

The use of logs and snapshots to recover databases after a crash is far from new and is covered in textbooks in the area, e.g., [12]. This work follows a more recent line of work on recovering databases [9, 26], operating systems [20], web applications [30] and other services [6] by eliminating the effect of undesirable requests without any software modifications. We discuss some of these works next.

The *"three R's"* is a model to undo incorrect operations in applications [6]. It consists of three basic operations: *Rewind*, to rollback to the latest valid snapshot in which no incorrect operation took place; *Repair*, in which invalid operations are corrected with the aid of the system administrator; *Replay*, in which the entire log is re-executed over the snapshot. A system that implements the *three R's* model is Undo for Operators [5, 6]. This system consists in a recovery system for Email Servers. It logs every message in a log (*Timeline Log*) with the help of a Proxy (*Undo Proxy*) and provides an control panel (*Undo Manager*) that is controlled using an interface (*Control UI*) by the system administrator. The architecture of Rectify was inspired by this work. It also uses proxies to intercept operations that are stored in logs and provides a control panel for the system administrator.

Warp [7] is a recovery system for web applications. It works by rolling back a part of the database to a point in time prior to the intrusion and then apply compensation operations to correct the state of the database. This approach is similar to Rectify's. In order to reproduce the HTTP requests, Warp also requires an extension to be installed on the web browser. Rectify re-executes the requests without using a client tool. Warp requires software modifications to the database.

Aire [8] is a recovery service for applications that use Web Services, so they need a recovery system that propagates the compensation operations. We assume a system model similar to Aire, in which the application that the system administrator intends to recover may use web services of different services. However, currently we only recover the application, without affecting external services, since we assume that in some cases the external web services may not be the system administrator is responsibility and must not be affected by recovery of the application. In some cases it is necessary to propagate the recovery operations across every interconnected system, and for that Aire provides a better approach.

The problem of undoing malicious operations from databases was investigated in [2] and later in [26]. These works explore the problems of dependencies between records and the calculations of compensation operations that need to need to respect such dependencies. The algorithm we use to find dependencies between records is based on [2] and the algorithm to undo malicious statements was also inspired by that work.

The subject of recovering web applications deployed in a PaaS was explored in a single work, to the best of our knowledge. Shuttle [30] consists in a recovery service for application deployed in a PaaS. It provides the same capabilities as Rectify, i.e., it removes the effects of intrusions without discarding correct operations that have occurred after the intrusion. Shuttle and Rectify differ in two main aspects: Shuttle was designed to be deployed with the application in the same container, while Rectify is isolated in a different container, which is important security-wise. Shuttle also needs software modifications to work, while Rectify does not. However, Rectify requires a configuration and training phase.

To the best of our knowledge this is the first work that automatically associates requests to statements using machine learning. The more generic problem of relating the requests at application level with operations at database level appeared before, but existing solutions require modifying the application code. In [42], the authors try to find an accurate way of identifying these associations to reduce the errors of anomaly-based detection systems. In [24, 27], the authors want to use the association information to improve web caching systems. Although in these works the authors do find a way to associate requests with operations, in their solutions they modify the software of the application and/or the DBMS.

10 Conclusion

This article presented Rectify, a black-box intrusion recovery service for PaaS applications. Rectify is a novel approach to recover attacked web applications that does not require modifications to the source code and that can be performed by a system administrator. Such approach can be adopted by cloud providers to provide a valued-added service to be used by their customers. The Rectify approach uses machine learning classification techniques to identify dependencies and associate database statements to HTTP requests. This accuracy relies on the application use of regular routes (well structured URL patterns), a common good practice of web design. Our implementation could recover 1,000 malicious HTTP requests in around 16 minutes.

Acknowledgements This work was supported by the European Commission through project H2020-653884 (SafeCloud) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] İ. E. Akkuş and A. Goel. 2010. Data recovery for web applications. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*. 81–90.

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

[3] D. J. Barrett. 2008. *MediaWiki*. O'Reilly.

[4] A. Brazell. 2011. *WordPress Bible*. John Wiley and Sons.

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

[6] A. Brown and D. Patterson. 2003. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the USENIX Annual Technical Conference*. 1–14.

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

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

[9] T. Chiueh and D. Pilania. 2005. Design, implementation, and evaluation of a repairable database management system. In *Proceedings of the 21st IEEE International Conference on Data Engineering*. 1024–1035.

[10] E. Ciurana. 2009. *Developing with Google App Engine*. APress.

[11] B. Cohen. 2013. PaaS: new opportunities for cloud application development. *Computer* 46, 9 (2013), 97–100.

[12] H. Garcia-Molina, J. Ullman, and J. Widom. 2008. *Database Systems: The Complete Book* (2nd ed.). Pearson.

[13] M. Gegick, E. Isakson, and L. Williams. 2006. An early testing and defense web application framework for malicious input attacks. In *ISSRE Supplementary Conference Proceedings*.

[14] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara. 2005. The Taser intrusion recovery system. In *ACM SIGOPS Operating Systems Review*, Vol. 39. 163–176.

[15] W. G. Halfond, J. Viegas, and A. Orso. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*. 13–15.

[16] E. H. Halili. 2008. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd.

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter* 11, 1 (2009), 10–18.

[18] A. Heydon and M. Najork. 1999. Mercator: A scalable, extensible web crawler. *World Wide Web* 2, 4 (1999), 219–229.

[19] K. L. Ingham and H. Inoue. 2007. Comparing anomaly detection techniques for HTTP. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*. 42–62.

[20] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. 89–104.

[21] H. F. Korth, E. Levy, and A. Silberschatz. 1990. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*. 95–106.

[22] S. B. Kotsiantis. 2007. Supervised Machine Learning: A Review of Classification Techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. 3–24.

[23] C. Kruegel, G. Vigna, and W. Robertson. 2005. A multi-model approach to the detection of web-based attacks. *Computer Networks* 48, 5 (2005), 717–738.

[24] P. A. Larson, J. Goldstein, and J. Zhou. Mtcache: Transparent mid-tier database caching in SQL server. In *Data Engineering, 2004. Proceedings. 20th International Conference on*. 177–188.

[25] LimeSurvey. 2017. An open source survey tool. (2017). https://www.limesurvey.org.

[26] P. Liu, J. Jing, P. Luenam, Y. Wang, L. Li, and S. Ingsriswang. 2004. The design and implementation of a self-healing database system. *Journal of Intelligent Information Systems* 23, 3 (2004), 247–269.

[27] Q. Luo and J. F. Naughton. 2001. Form-based proxy caching for database-backed web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 191–200.

[28] D. Matos and M. Correia. 2016. NoSQL Undo: Recovering NoSQL Databases by Undoing Operations. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*.

[29] P. Mell and T. Grance. 2011. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology* (2011).

[30] D. Nascimento and M. Correia. 2015. Shuttle: Intrusion Recovery for PaaS. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*. 653–663.

[31] G. Nascimento and M. Correia. 2011. Anomaly-based intrusion detection in software as a service. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*.

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

[33] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. 2008. Amazon S3 for science grids: a viable solution?. In *Proceedings of the International Workshop on Data-Aware Distributed Computing*. 55–64.

[34] R. Peinl, F. Holzschuher, and F. Pfitzer. 2016. Docker Cluster Management for the Cloud – Survey Results and Own Solution. *Journal of Grid Computing* (2016), 1–18.

[35] S. Pousty and K. Miller. 2014. *Getting started with OpenShift*. O'Reilly.

[36] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. 2006. 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*.

[37] M. Safronov and J. Winesett. 2014. *Web Application Development with Yii 2 and PHP*. Packt Publishing Ltd.

[38] Z. Su and G. Wassermann. 2006. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*.

[39] L. M. Vaquero, L. Rodero-Merino, and Ra. Buyya. 2011. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review* 41, 1 (2011), 45–52.

[40] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. 2008. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review* 39, 1 (2008), 50–55.

[41] J. Varia and S. Mathew. 2014. Overview of Amazon Web Services. *Amazon Web Services* (2014).

[42] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda. 2009. Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and SQL queries. *Journal of Computer Security* 17, 3 (2009), 305–329.

[43] C. D. Weissman and S. Bobrowski. 2009. The Design of the Force.com Multitenant Internet Application Development Platform. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. 889–896.

[44] J. Williams and D. Wichers. 2013. *OWASP Top 10 - The Ten Most Critical Web Application Security Risks*. Technical Report. OWASP Foundation.