UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Intrusion Recovery in Cloud Computing

David Rogério Póvoa de Matos

Supervisor:	Doctor Miguel Nuno Dias Alves Pupo Correia
Co-Supervisor:	Doctor Miguel Filipe Leitão Pardal

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

2019

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Intrusion Recovery in Cloud Computing

David Rogério Póvoa de Matos

Supervisor:	Doctor Miguel Nuno Dias Alves Pupo Correia
Co-Supervisor:	Doctor Miguel Filipe Leitão Pardal

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson: Doctor Mário Jorge Costa Gaspar da Silva, Instituto Superior Técnico, Universidade de Lisboa

Members of the Commitee:

Doctor Nuno Manuel Ribeiro Preguiça, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
Doctor Rui Carlos Mendes de Oliveira, Escola de Engenharia, Universidade do Minho
Doctor Miguel Nuno Dias Alves Pupo Correia, Instituto Superior Técnico, Universidade de Lisboa
Doctor António Manuel Ferreira Rito da Silva, Instituto Superior Técnico, Universidade de Lisboa

2019

Aos meus pais.

Acknowledgments

First I want to thank my advisers, Professor Miguel Correia and Professor Miguel Pardal, for taking the challenge of guiding me through this path.

Professor Miguel Correia accepted to advise my PhD before I was a student at Instituto Superior Técnico. He gave me the opportunity to research on a topic that I was interested in and provided me with a grant that allowed me to commit in full time to my academic work. He also gave me the opportunity to pursue an internship at the University of Texas at Austin, a very gratifying experience that I will always remember.

Professor Miguel Pardal took the challenge of advising my PhD after it had began. I appreciate that he accepted that challenge of taking an undergoing work. During the course he also encourage me to teach at Instituto Superior Técnico, an enriching experience. Besides helping with my PhD work he also gave me some valuable advice for my teaching experience.

I want to thank INESC-ID and Instituto Superior Técnico for giving me the working conditions during these four years.

Thanks to Miguel Coimbra, for every lunch, coffee break and positive thinking that helped me stay optimistic during this long journey.

Thanks to Jorge Andrade and Ricardo Maia for many lunches, companionship and friendship since before the start of this PhD.

A very special thank goes to my lovely girlfriend, Raquel, who was always there for me and encouraged me to never give up every time I had second thoughts.

Finally, I want to thank my parents for giving me the opportunity to study away from home and for giving me everything I needed to complete my academic course. They always encourage me to pursue my dreams and challenge me to surpass my expectations.

This work was partially supported by the SafeCloud project financed by the European Commission through project H2020-653884 and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and UID/CEC/50021/2019 (INESC-ID) for financially supporting my PhD.

Publications

The developed work presented in this thesis can be partially found in the following publications:

- David Matos, Miguel Correia. NoSQL Undo: Recovering NoSQL Databases by Undoing Operations. In Proceedings of the 15th IEEE International Symposium on Network Computing and Applications (NCA), Nov. 2016.
- David Matos, Miguel Pardal, and Miguel Correia. Rectify: Black-Box Intrusion Recovery in PaaS Clouds. In Proceedings of the 2017 ACM/IFIP/USENIX International Middleware Conference, Dec. 2017.
- David Matos, Miguel Pardal, Georg Carle, and Miguel Correia. RockFS: Cloud-backed File System Resilience to Client- Side. In Proceedings of the 2018 ACM/IFIP/USENIX International Middleware Conference, Dec. 2018.
- David Matos, Miguel Pardal, and Miguel Correia. μVerum: an Intrusion Recovery Approach for Microservice Applications. *In submission*.

Resumo

Os mecanismos de prevenção de intrusões em aplicações informáticas têm por objetivo reduzir a possibilidade de intrusões ocorrerem. Contudo, mais cedo ou mais tarde um atacante poderá ser bem-sucedido a explorar uma vulnerabilidade desconhecida ou a roubar credenciais de um utilizador legítimo, levando à execução de operações não desejadas. Estas intrusões podem corromper o estado da aplicação sendo necessário recorrer a mecanismos de recuperação para reverter o efeito dessas ações. Soluções simples como o uso de cópias de segurança permitem reverter os efeitos de uma intrusão na aplicação, contudo ao restaurar uma cópia de segurança perdem-se todos os dados válidos que não estão presentes nela. Portanto, os mecanismos de recuperação de intrusões têm por objetivo reverter os danos causados por intrusões sem afetar os dados legítimos que foram produzidos por utilizadores autorizados.

Esta tese teve como objetivo o estudo de sistemas de recuperação de intrusões para sistemas distribuídos baseados em computação na nuvem. Estes mecanismos têm em conta a distribuição dos sistemas, as limitações dos fornecedores de computação na nuvem e os paradigmas de desenvolvimento de aplicações adequados para estes sistemas. Dada a heterogeneidade dos vários modelos de computação na nuvem, estudaram-se diferentes sistemas de recuperação de intrusões adequados aos diferentes modelos. Para o nível da infraestrutura do modelo de computação da nuvem, nomeadamente, para o armazenamento de ficheiros propomos, o RockFS, um sistema de recuperação de intrusões para sistemas de ficheiros suportados pela nuvem. Este tipo de sistema de ficheiros é acedido remotamente permitindo que um atacante seja capaz de modificar ilegalmente ficheiros se tiver acesso a um dispositivo do utilizador. O RockFS protege as credenciais dos utilizadores através de um mecanismo de partilha de segredos, que permite distribuir fragmentos da chave em diversos dispositivos de armazenamento. A nível de recuperação, o RockFS permite anular operações não pretendidas através do uso de registos de operações e de armazenamento de múltiplas versões de ficheiros. O RockFS atua no lado do cliente e pode ser usado em sistemas de ficheiros de nuvem única e de múltiplas nuvens.

Também no nível da infraestrutura do modelo de computação da nuvem, mas para bases de dados, é apresentado o NoSQL Undo, um sistema de recuperação de intrusões para bases de dados NoSQL. Este sistema não requer qualquer alteração ao código fonte do sistema de gestão de base de dados (SGBD), podendo ser adotado por SGBDs que não tenham o código fonte disponível. O NoSQL Undo tira partido de um mecanismo que regista operações para efeitos de replicação, reduzindo a sobrecarga do sistema a nível de desempenho. O NoSQL Undo funciona no lado do cliente e pode ser usado para recuperar de intrusões sem instalação ou configuração prévia. O NoSQL Undo fornece dois algoritmos de recuperação: focada e completa. A recuperação focada corrige apenas os registos de base de dados afetados pelo ataque, enquanto que a recuperação completa reverte todos os registos da base de dados. O uso de um algoritmo em detrimento do outro depende da quantidade de registos afetados pelo ataque.

A nível aplicacional, do modelo de computação em nuvem, propomos um sistema de recuperação de intrusões chamado Rectify que permite anular os efeitos de ataques em aplicações web. É possível usar o Rectify em qualquer aplicação web que use uma base de dados SQL para guardar o seu estado. O Rectify identifica operações maliciosas na base de dados que foram gerados por pedidos maliciosos feitos à aplicação. Esta associação de operações de base de dados com pedidos de aplicação é feita recorrendo a algoritmos de aprendizagem automática. A principal vantagem desta técnica é que não é necessário fazer modificações ao código fonte da aplicação nem ao código fonte da base de dados. Rectify permite efetuar a recuperação de intrusões mantendo a aplicação disponível para os seus utilizadores.

Para dar resposta às mais modernas práticas de desenvolvimento de aplicações complexas assumindo o modelo de computação em nuvem, as aplicações de micro-serviços, propomos o μVerum, um sistema de recuperação de intrusões que adota a arquitetura de micro-serviços. Neste tipo de aplicações cada componente do sistema está distribuído em serviços independentes que interagem através da rede. O μVerum foi desenhado tendo em conta a distribuição e o carácter autónomo de cada micro-serviço. O μVerum permite propagar as operações de compensação, que revertem os efeitos da intrusão, nos vários serviços afetados. O μVerum apresenta uma arquitetura modular que permite replicar os componentes com maiores requisitos de tráfego de forma a manter o nível de desempenho da aplicação no mesmo patamar. O μVerum permite que os programadores da aplicação definam invariantes para cumprir requisitos de coerência dos dados. Estas invariantes podem ser de dois tipos: de atomicidade, em que vários micro-serviços devem ser executados em conjunto; e de ordenação, em que um conjunto de micro-serviços deve ser executado numa determinada ordem.

No seu conjunto, os contributos deste trabalho permitem tornar as aplicações baseadas no modelo de computação em nuvem capazes de resistir aos efeitos nefastos de intrusões mantendo os modelos de programação muito próximos dos atuais, e com níveis de desempenho similares na perceção dos utilizadores finais.

Palavras-chave: recuperação de intrusões, computação na nuvem, sistema de ficheiros, base de dados NoSQL, micro-serviços.

Abstract

Intrusion prevention mechanisms aim to reduce the probability of intrusions to occur. However, sooner or later an attacker may succeed in exploiting an unknown vulnerability or by stealing a user's access credentials, leading to the execution of undesired operations. These intrusions may corrupt the state of the application requiring intrusion recovery mechanisms to revert the effect of these actions. Simple solutions such as, the use of backups, allow reverting the effects of an intrusion, however, by restoring a previous backup of the system, every legitimate data that is not present in that backup is lost. Intrusion recovery mechanisms aim to revert only the damage caused by intrusions without affecting the legitimate data that was created by authorized users.

This thesis explores the problem of intrusion recovery for distributed systems running in the cloud. The presented mechanisms take into account the distribution of the systems, the limitations of the cloud services and the development paradigms for this kind of systems. Given the heterogeneity of the different cloud computing models, there were designed different intrusions recovery mechanisms for the different models. For the infrastructure level of the cloud, namely, for file storage we propose RockFS, an intrusion recovery system designed for cloud-backed file systems. This kind of systems is accessed remotely allowing attackers to illegally modify files by accessing a legitimate user's account. RockFS protects the access credentials of the user through secret sharing mechanisms. which allow distributing fragments of the access credentials through several storage devices. For recovery, RockFS allows reverting unintended actions through the combination of operation logs and a multiversioned of file system. RockFS runs on the client-side and can be used in single-cloud and cloud-of-clouds file systems.

At the infrastructure level of the cloud, but for databases, we present NoSQL Undo, an intrusion recovery system for NoSQL databases. This system does not require modifications to the source code of the Database Management System (DBMS), making it possible to be adopted by DBMS that do not provide the source code. NoSQL Undo takes advantage of the logs used by the database for replication, reducing the performance overhead. NoSQL Undo runs on the client-side and can be used to recover from intrusions without a previous installation or configuration. NoSQL Undo provides two algorithms for recovery: focused recovery and full recovery. Focused recovery only fixes the database records that were affected by the attack, while the full recovery fixes the entire database. The use of one algorithm as opposed to the other depends on the amount of affected database records by the attack.

At the application level, of the cloud computing model, we propose an intrusion recover system called Rectify that allows reverting the effects of the attack in web applications. It is possible to use Rectify in any web application that uses a SQL database to store its state. Rectify identifies malicious operations in the database that were generated by malicious requests performed by the application. This association, of database operations with application level requests, is done through machine learning algorithms. The main advantage of this technique is that it does not require modifying the source code of the application or the source coed of the database. Rectify allows recovering from intrusions while keeping the application available for users.

For modern distributed applications running in the cloud developed in a microservices paradigm, we propose μVerum, an intrusion recovery system that adopts the microservices architecture. In this kind of applications each component of the system is distributed in independent services that interact with each other through the network. μVerum was designed taking into account the distribution and self-contained characteristics of each microservice. μVerum allows propagating the compensating operations, the revert the effects of the intrusion, on the affected services. μVerum presents a modular architecture that allows to replicate the components with higher traffic demands in order to maintain the performance level of the application. μVerum allows the developers of the application to define invariants in order to fulfill the consistency requirements of the application. These invariants can be of two types: atomicity, in which several microservices should be executed together; and ordering, in which a group of microservices should be executed in a specific order.

The work presented in this thesis allows applications deployed in the cloud to overcome intrusions. The proposed systems can be deployed in existing cloud services and interfere as less as possible with the user experience.

Keywords: intrusion recovery, cloud computing, file systems, microservices, NoSQL databases.

Contents

List of Tables

List of Figures

List of Algorithms

Glossary

AES	Advanced Encryption Standard
CRUD	Create, Read, Update and Delete
DBMS	Database Management System
ENISA	European Union Agency for Network and Information Security
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IMAP	Internet Message Access Protocol
IaaS	Infrastructure-as-a-Service
MVC	Model-View-Controller
NAS	Network-Attached Storage
NoSQL	No Only SQL
OS	Operating System
PaaS	Platform-as-a-Service
RDBMS	Relational Database Management System
RPC	Remote Procedure Call
SCFS	Shared Cloud-backed File System
SHA	Secure Hash Algorithms
SIEM	Security Information and Event Management
SMPT	Simple Mail Transfer Protocol
SQL	Structured Query Language
TTTR	Total Time To Recover
UI	User Interface

1 Introduction

Intrusion recovery in the cloud poses challenges that are not present in other distributed systems. The cloud provides different computing models for different purposes, making it necessary to design a tailored intrusion recovery solution for each one. Such intrusion recovery mechanisms should be designed taking into account that it is not possible to modify the cloud systems. In this chapter we introduce the problem of intrusion recovery for cloud computing.

1.1 Cloud Computing

Cloud computing provides a convenient way to rent computing resources (Vaquero et al., 2008, 2011). This model enhances the capabilities of existing applications and systems by allowing automatic scaling with a cost proportional to the used resources. Instead of acquiring, configuring and managing a cluster of servers, organizations are now able to pay for the time they are using the computing resources, leaving the installation, configuration and maintenance to a third party, the Cloud Service Provider (CSP). The cloud has become a platform for computation that allows the execution of database instances, web applications, storage services and software accessible through the Internet.

Figure 1.1 represents the three main cloud computing models. The lowest layer of the stack, Infrastructure as a Service (IaaS), contains computing resources provided through virtualization, elastic storage and databases. In this layer there is more freedom for configuration and setup of the acquired services at the cost of complexity. The middle layer of the stack, Platform as a Service (PaaS), contains execution environments that allow organizations to deploy applications. This level limits the possible configuration and setup that can be done, however, it facilitates the process of deploying and running the applications thanks to the abstraction provided by the execution environments. The top level of the stack, Software as a Service (SaaS), contains applications that are provided for organizations without the need to install additional software in the cloud. Organizations that contract this kind of

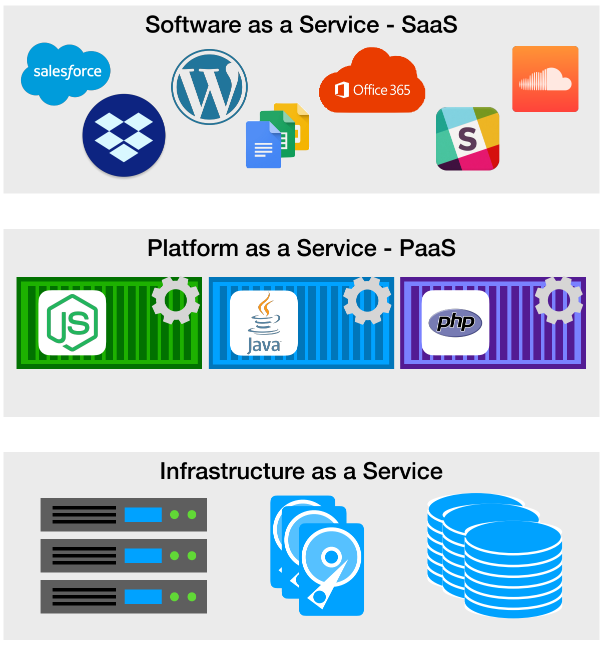
Software as a Service - SaaS
salesforce
Office 365
Platform as a Service - PaaS
JS
Java
php
Infrastructure as a Service

Figure 1.1: The cloud computing models.

applications cannot modify the software. Depending on the needs of the organization, it may acquire services in the different layers of the cloud computing model.

The convenience provided by the cloud comes with associated costs: first, most services provided by the cloud limit the configuration options with the objective of making the deployment and management as simple as possible; second, data is more exposed to unauthorized users since every cloud service is accessible through the Internet (Takabi et al., 2010; Mather et al., 2009); and third, there is a new set of system administrators that belong to the cloud organizations that have privileged access to the infrastructure and may read or corrupt user data.

1.2 Cloud Security

Organizations are aware of the cloud security risks and rely on several intrusion prevention mechanisms provided by the CSPs to reduce the probability of attacks succeeding. Some of these mechanisms are configured and managed by the CSPs while other can be modified by the organizations that acquire the cloud services. Some of these security mechanisms include: firewalls that filter inbound traffic and prevent unauthorized users from accessing private networks; access control mechanisms that prevent unauthorized users from accessing private content; and Intrusion Detection Systems (IDS) that monitor the applications for malicious activity.

A firewall (Ioannidis et al., 2000) protects a system connected to the network against unauthorized access. This kind of system regulates incoming and outgoing network traffic following a set of rules. For example, these rules can limit the traffic to a certain IP address range, only allow traffic to a specific port or exclude some servers from traffic that comes from outside of the organization. However, if a system administrator fails to configure a firewall with every necessary rule (Wool, 2004), then an attacker may be able to illegally access the application. Once an attacker has access to the application he may corrupt the users' data.

Access control mechanisms (Sandhu and Samarati, 1994; Gasser, 1988) regulate which users are allowed to use the computing resources. These mechanisms are responsible for identifying, authorizing and authenticating the users, allowing system administrators to audit and manage the access records. CSPs provide access control mechanisms that allow organizations to define user privileges and ensure that only authorized users have access to the organizations data and computing resources. Such mechanisms are only effective against attackers when they are correctly configured and managed. If an attacker gains access to a user's credentials by, for example, stealing his computer, he will be able to perform an attack and corrupt data.

Intrusion Detection Systems (Denning and Neumann, 1985; Debar et al., 1999) monitor network traffic and operation logs to identity malicious activity. If the systems suspect an attack is underway, they notify the system administrator so they can trigger the adequate counter-measures. Some activity may be falsely reported to the system administrator, the so-called false positives. Also, there may be attacks that are not detect, the false negatives. An evolution of IDSs are Security Information and Event Management (SIEM) systems, which analyze the activity alongside with other sources and filter the possible attacks from the false positives and become aware of false negatives. The combination of these mechanisms help system administrators in identifying malicious activity, i.e., attacks, and execute the appropriate counter-measures. The disadvantage of using such mechanisms is that once the alarm was triggered the attack already occurred. When this happens the system administrator needs to repair the system, which includes applying security patches, re-configuring the network and reverting the corrupted data to the most recent version prior to the attack.

The presented security mechanisms provided by the CSPs make attacks less likely to succeed. However, despite the best efforts, malicious users may still access the cloud services by, for example, exploiting a vulnerability or poor configuration, and once an attacker gains access to a cloud service he may corrupt users' data, e.g., using previously unknown (zero-day) vulnerabilities.

Data corruption may result in significant losses for the organizations. Recently we witnessed the global spread of a ransomware attack — Wannacry (Mohurle and Patil, 2017) — that successfully infected more than 230,000 systems and resulted in an estimated loss of $4 (USD) billion. According to Accenture (Varonis, 2018), the most expensive component of a cyber-attack is information loss due to data corruption, which represents 43% of the total costs of the attack. Although cyber-attacks remain the leading cause of data loss incidents — 55% of the total incidents in 2017 — overall, human error caused even more data corruption (Info Security, 2018). Therefore, it can be argued that services that are capable of reverting the effects of intentional and accidental state modifications are useful for organizations that use the cloud to run their applications.

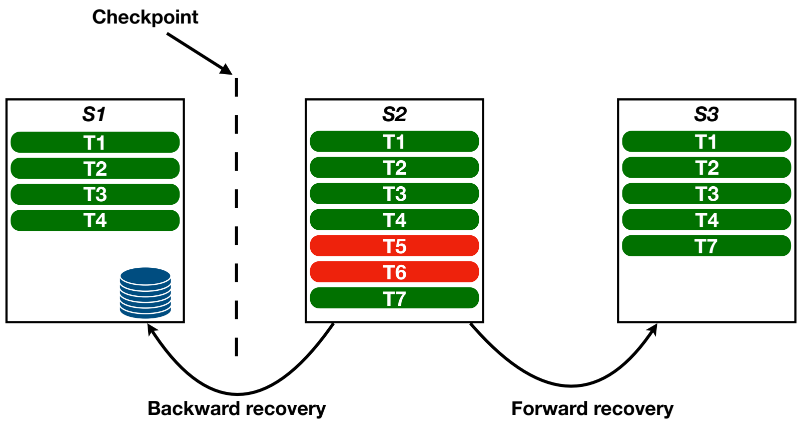

Figure 1.2: Restoring a system to a previous backup vs recovery (S1, S2, S3 are states; T1 to T7 are operations).

1.3 Intrusion Recovery

Intrusion recovery involves the tasks that need to be performed in order to revert the effects of an attack. These tasks consist in identifying the corrupted data and reverting it to the previous state prior to the attack, while keeping the legitimate data intact. To achieve this state, intrusion recovery mechanisms use a combination of periodic checkpoints with logs of the executed operations. Using only checkpoints of the state would not be ideally, since they would allow the entire state of the system to be reverted, losing the legitimate operations in the process.

Figure 1.2 presents two different approaches to revert data that was corrupt by an attack. In the figure the system state evolves from *S1* to *S2* after executing the operations: T5, T6 and T7. The problem is that operations T5 and T6 are malicious and corrupted the state of the system and need to be reverted. One possible approach, called *backward recovery*, consists in reverting the entire system state back to *S1*. This is possible since a backup was performed before the attack. This discards the effects of the malicious operations T5 and T6. The problem is that transaction T7 is discarded as well. A more preferable solution would be to only undo the effects of the attack. Such approach is called *forward recovery* and, in this example, it only discards the effects of the operations T5 and T6 while keeping the effects of transaction T7 and reaching the state in *S3*.

An intrusion recovery system can be used to achieve the desirable state *S3* of Figure 1.2, in which the effects of malicious operations are reverted and the legitimate one as kept (Brown and Patterson, 2003; Goel et al., 2005b; Korth et al., 1990; Bernstein et al., 1987; Ammann et al., 2002). Intrusion recovery systems work by recording in a log every operation that affects the state of the application. With this log, a system administrator may select unintended actions that were caused by the attack and use the intrusion recovery system to undo them. In other words, intrusion recovery mechanisms aim to revert the damage intentionally caused by attackers or accidentally by authorized users, while keeping intact data created and modified by legitimate users. These mechanisms assume that attacks already occurred and that it is necessary to revert their effects from the state of the system. This assumption is realistic given that even adopting intrusion prevention techniques will reduce the probability of attacks to be successful, attackers may always discover new ways to exploit the system.

1.4 Intrusion Recovery for Cloud Computing

There is some work in the literature regarding intrusion recovery for distributed environments (Brown and Patterson, 2003; Korth et al., 1990; Bernstein et al., 1987; Ammann et al., 2002; Chandra et al., 2011; King and Chen, 2003). Although they are capable of recovering from intrusions in different systems, they do not take into account the limitations or the features of the cloud context. This leads to two questions: is it possible to implement intrusion recovery mechanisms given all the limitations of cloud computing? And if so, would such mechanisms benefit from the features provided by cloud services?

Given the heterogeneity of the cloud computing model, it is not possible to design a single intrusion recovery system capable of recovering any system running in the cloud. Every intrusion recovery mechanism is designed with a target system in mind (Brown and Patterson, 2003). This is a requirement given that each system accepts a specific type of requests and has its own format when storing the state to storage. The intrusion recovery mechanism needs to be able to interpret the requests of the system it is protecting and interact with it in order to perform recovery. Revisiting Figure 1.1, and assuming that intrusions may occur at any level of the cloud, an intrusion that occurs in the SaaS level will require a different recovery approach than an intrusion that occurred in the PaaS or IaaS level. Therefore, the same way there are different computing options provided by the cloud there should be different intrusion recovery systems.

One challenge in adopting existing intrusion recovery systems to the cloud is the impossibility to modify the source code of the cloud platform. This is a challenge since most of the existing intrusion recovery mechanisms assume that it is possible to modify the source code of the systems and applications they are protecting(Brown and Patterson, 2003; Korth et al., 1990; Bernstein et al., 1987; Ammann et al., 2002; Chandra et al., 2011; King and Chen, 2003). For example, an intrusion recovery system designed for web applications that require modifying the PHP interpreter cannot be used in a web application running in a Platform as a Service (Akkuș and Goel, 2010). Therefore, intrusion recovery mechanisms designed for the cloud need to adopt different methods that require minimal modifications to the running platform.

1.5 Thesis Statement

This goal of this thesis is to present the design and implementation of novel intrusion recovery mechanisms designed for the cloud computing model. The presented work was developed in the Distributed Systems Group at INESC-ID in the context of the SafeCloud european project.[1]

[1]See https://www.safecloud-project.eu

The thesis statement is the following four intrusion recovery systems, each targeted at different levels/parts/components of the cloud platform:

intrusion recovery for cloud computing applications is limited by the inability to modify the source code of the cloud platform and, in some cases, of the applications themselves. The presented mechanisms show that it is possible to design intrusion recovery mechanisms without modifying the underlying code of the cloud platform and with limited modifications to the code of the applications.

This thesis assumes a system model in which distributed applications run in cloud servers. This grants some features that can be used by the recovery system, such as automatic scaling, access through the Internet and automatic deployment.

In this thesis we argue that each cloud service requires a tailored intrusion recovery mechanism. Such mechanisms should also take advantage of cloud features such as, scalability, easy deployment and network infrastructure. The cloud system model incorporates different types of systems. At the lowest level there are storage, virtualization and database systems, at a higher level there are platforms with execution environments. It is not possible to design an intrusion recovery mechanism compatible with every service provided by the cloud computing model.

It is assumed that intrusions can be tracked at the application level. This assumption is based on the existence of different intrusion detection systems available in the literature (Kruegel et al., 2005; Robertson et al., 2006; Ingham and Inoue, 2007; Nascimento and Correia, 2011).

1.6 Contributions

The contributions of the thesis are the following:

RockFS (Matos et al., 2018), an intrusion recovery service for cloud-backed file systems. This recovery system leverages existing storage services to provide a recovery mechanism resilient to client-side attacks. RockFS allows users to revert unintended actions that occurred in the client device and were synchronized to the cloud. The middleware also encrypts data stored in the client device reducing the probability of a data breach. Secret sharing algorithms protect the encryption keys allowing them to be distributed among different data stores.

NoSQL Undo (Matos and Correia, 2016), a generic recovery system designed for NoSQL databases. Unlike other recovery systems, NoSQL Undo does not require modifications to the code of the database management system. Instead, it uses the existing replication logs allowing system administrators to perform recovery even if the intrusion recovery system was not installed before the intrusion.

Rectify (Matos et al., 2017), an intrusion recovery service for web applications deployed in Platform-as-a-Service. Rectify uses machine learning algorithms to correlate user-level requests with database level operations, making it possible to track and recover the damage in the database caused by attacks in the web application.

μVerum, a system design and prototype that provides intrusion recovery capabilities for microservices applications. This system follows the architecture of existing large scale microservice applications. The proposed system allows to recover from intrusions without shutting down the application.

It also allows developers to define consistency requirements to ensure that the application remains consistency after recovery.

As a whole, the solutions proposed in this work allow applications deployed in the cloud to overcome intrusions in ways that go beyond the state-of-the-art. The proposed systems can be deployed in existing cloud services and interfere as less as possible with the user experience.

1.7 Dissertation Outline

Chapter 2 gives an overview of relevant concepts related with cloud computing, databases, storage systems and intrusion recovery.

Chapter 3 presents the state of the art in the field of intrusion recovery for distributed systems and applications.

Chapter 4 presents the design and implementation of RockFS, the intrusion recovery system for cloud storage.

Chapter 5 describes the implementation and design of NoSQL Undo, a generic intrusion recovery system for NoSQL databases.

Chapter 6 presents the design and implementation and evaluation of Rectify, an intrusion recovery system for web applications the run on Platform-as-a-service.

Chapter 7 presents the μVerum approach for intrusion recovery for microservices applications.

Chapter 8, concludes the document with the final remarks of the developed work and a discussion of possible future work challenges.

2 Background

In this chapter we present a brief explanation of some concepts relevant for the works described in this dissertation. We start by giving an overview of cloud computing. Then we present the different storage services available in the cloud, starting with block storage, object storage and file systems, and ending with a comparison between the two types of database management systems: SQL and NoSQL. Finally, we present common paradigms and architecture for web application development.

2.1 Cloud Computing

Cloud computing (Geelan et al., 2009; Knorr and Gruman, 2008; Mcfredries, 2008; Hayes, 2008) allows provisioning computational resources over the network. This paradigm allows organizations to reduce their infrastructure costs by only paying the computing resources they are using while they are needed, instead of purchasing the required hardware. For this reason, cloud is also called *utility computing* because it is as if the computing resources are readily available for consumption, in a similar way to power and water. The cloud model is even more attractive than owned infrastructure when dealing with peaks of demand. In a cloud model, more resources can be added to handle the peak load — the term *elastic cloud* deals precisely with this capability — whereas the owned infrastructure would have to be bigger to deal with the peaks and then remain underutilized for the rest of the time, leading to significant inefficiency.

Depending on the kind of computing service being provided, cloud computing can be classified in several models. The three original models are *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS) (Mell and Grance, 2011). Depending on who manages and provides the service, a cloud can be defined as a *public*, *private* or *hybrid*.

Public clouds are the ones provided and managed by third parties, meaning that users' data is kept outside of the users' domain. Private clouds are cloud infrastructures deployed and manage by the organization that uses their services. This approach allows the cloud users to maintain their information in their organization's domain. Hybrid clouds are supported by computing resources that are split between the organization that uses it and a third party. A specific example of hybrid cloud is having computation and working data on-premises but keep data archive in an external cloud storage.

2.1.1 Cloud computing models

The IaaS model is mainly used by the IT staff and consists in automatically provisioning virtualized computer resources (Bhardwaj et al., 2010), such as storage, network and virtual servers. A common use of a IaaS is to host applications that need to be accessible over the Internet. Such applications include: web applications, back-end servers for mobile applications, database servers and servers for web services. Another use for IaaS consists in renting virtual machines with different operating systems and supported by different computing configurations.

The PaaS model is targeted to developers and provides a platform that 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 communicate with browsers using the HTTP protocol. PaaS offerings normally support elastic web applications that scale horizontally, i.e., by deploying more virtual machines to handle the workload. PaaS offerings may be provided on top of IaaS offerings. The persistent state of a PaaS is usually provided by a DBMS that can be SQL or NoSQL. It is common to find PaaS clouds with different storage options like a file system or an elastic storage service, such as Amazon S3 (Palankar et al., 2008). In Section 6.1 we provide more details on the architecture of PaaS clouds designed for storage.

Finally, the SaaS model, used by the end-users, consists in providing applications over the Internet (Turner et al., 2003). The user does not need to install or configure the software on his machine since everything is hosted and being executed in remote servers. This model does not require the user to implement the software, however, he may configure it.

The three models layer themselves on top of each other. Figure 2.1 presents the common architecture of the three cloud computing models — IaaS, PaaS and SaaS. Starting from the bottom, IaaS, we can see that the platform is supported by a distributed system composed by several processes. Each one of these processes has a virtualization layer (Vaquero et al., 2011), allowing virtual machines to be instantiated. The virtual machines are controlled by the *Infrastructure Manager*. This allows users of the cloud to acquire and immediately instantiate virtual machines as services through the *Infrastructure Interface*. The middle computing model, PaaS, uses the IaaS model to instantiate the required database instances (*DBMS*) and application servers in order to provide an execution environment to the developers. A *Database Access and Replication Manager* deal with replicated databases and control multi-user access to the database instances. An execution environment is provided to the developers as a *container*, in which he can deploy one or more applications. Developers and production engineers interact with the PaaS through the PaaS APIs, allowing them to automatically deploy applications and reconfigure the containers. A *Load Balancer* is responsible for routing the users' re-

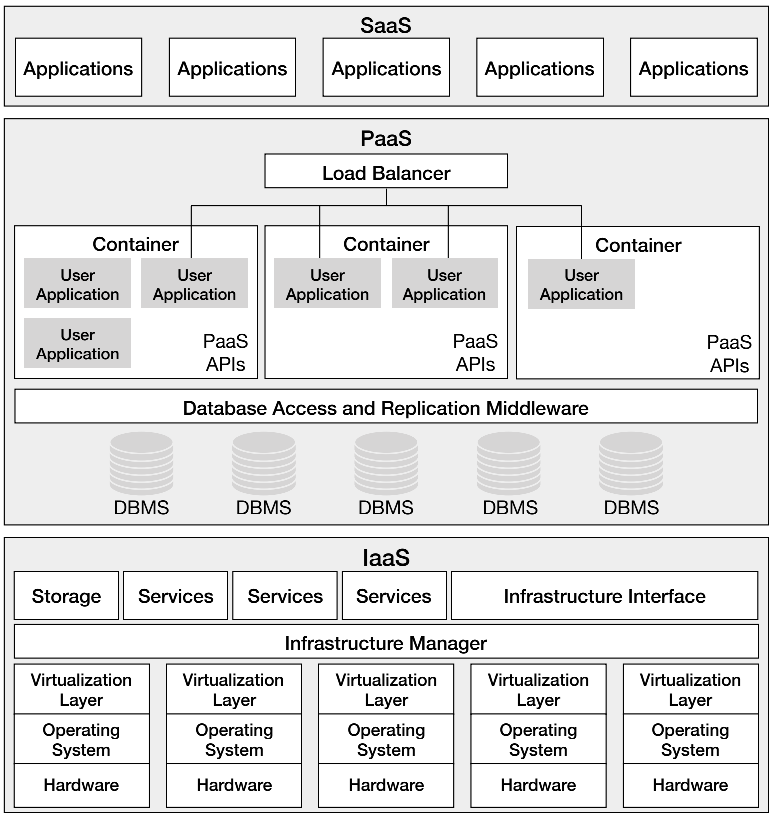

Figure 2.1: Common architecture of the three cloud computing models.

quests to the corresponding application servers. Finally, the SaaS mode can be deployed on top of a PaaS offering, providing the end-users with applications accessible though the web without requiring additional installation or configuration.

2.1.2 Cloud-of-clouds

The cloud-of-clouds approach (Slamanig and Hanser, 2012; Correia, 2013; Bessani et al., 2011) goes beyond a single cloud provider and allows users to setup a cloud service that is supported by a set of cloud offerings from different providers. This allows users to implement services that require higher levels of security and dependability that cannot be entrusted to a single cloud. To achieve this, it is necessary to implement a middleware responsible for dealing with the heterogeneity of the different cloud offerings. The advantages of using a cloud-of-clouds as opposed to a single cloud are:

- increased integrity and availability, since even if a subset of the clouds fails the remaining ones are capable of providing the service;
- *disaster-tolerance*, by distributing the cloud resources geographically;
- *vendor lock-in* prevention, because the system is built assuming different cloud providers, so they can be replaced with ease.

These advantages come with the extra monetary cost of supporting multiple clouds and the necessary development and maintenance of the middleware, which is more complex because of the need

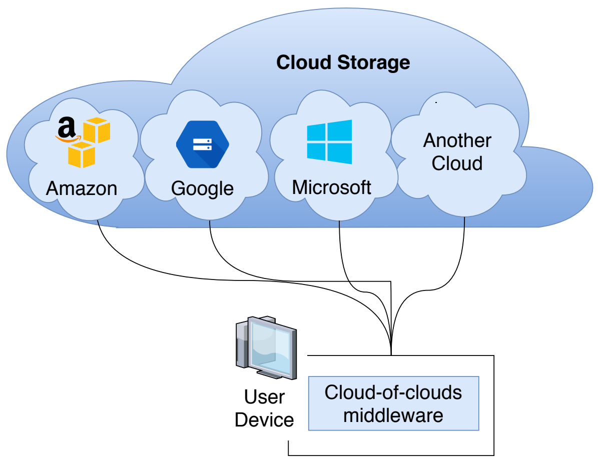

Figure 2.2: System architecture of a cloud-of-clouds storage service.

to handle different interfaces on different providers. Also, communication costs between different clouds are usually larger than communication inside the same cloud. A cloud-of-clouds model can be established to provide storage, data processing or service replication. A storage cloud-of-clouds offering allows users' data to be split into several, geographically distributed, clouds, increasing the availability and integrity levels (Bessani et al., 2011). For computation it is possible to setup a cloud-of-clouds to improve the dependability, allowing applications to scale out computations and tolerate arbitrary and malicious faults and cloud outages (Costa et al., 2017).

Figure 2.2 presents an example of the system architecture of a cloud-of-clouds storage service. The client accesses the clouds through a middleware that deals with the heterogeneity of the different clouds service providers. In this example there are four cloud providers and they all belong to different organizations, this helps to preventing vendor lock-in and, given that the different cloud organizations have their data-centers in different locations, this also provides disaster tolerance.

2.2 Cloud Storage

Cloud storage allows users to rent elastic storage capacity through the network. The cloud providers are responsible for maintaining availability of the users' data and provide more capacity on demand. Users' data is commonly split between several servers which may be distributed geographically in different data centers to provide disaster tolerance. Besides this, it is possible to achieve reduced latency by moving the users' data to the closest data center. Users access the storage service through web services. Cloud storage can be used as a standalone service with the sole purpose of storing and providing user's files, or it can be used in combination with other cloud services, allowing the applications to store their state. Cloud storage providers may also offer security features, such as data encryption, access control mechanisms and automatic backups (Burihabwa et al., 2016).

2.2.1 Cloud storage models

Cloud storage provides different types of storage: block storage, object storage and file system storage. Table 2.1 shows the differences between block storage, object storage and file systems. In the table the first column lists the main characteristics of a storage service along with the corresponding advantages and disadvantages.

	Block Storage	**Object Storage**	**File System**
Storage Unit	Block	Object (files + metadata)	File
Accessible through	SCSI, Fibre Channel, SATA	REST / SOAP	CIFS / NFS
Metadata	System attributes	Application attributes	File-system attributes
Advantages	Performance	Scalable	Hierarchical structure
Disadvantages	Lack of structure and metadata File logic in the application	Lack of structure File logic in the application	Difficult to scale Platform-dependent

Table 2.1: Comparison between different storage mechanisms.

Block storage is an abstraction of storage volumes that provides users with virtual drives accessible though the network. Each file stored in a block storage volume gets a unique identifier that will be used to access it. No more metadata is given to each file stored in a block storage unit, meaning that the structure of files inside the virtual drive is a responsibility of the application.

Object storage, rather than splitting files into raw data blocks, clumps data together as one object that contains data and metadata. Object storage can provide more context than blocks of storage about the data, which can be helpful in classifying and customizing the files. Each object also has a unique identifier, which makes quicker work of locating and retrieving objects from storage.

A *file system* manages how data is kept in a storage unit, providing users with a high level abstraction to access their data. The storage unit can be one or more hard disks in a local computer, a cluster of servers connected through the network or a cloud storage with a file system interface (Vrable et al., 2012; Bessani et al., 2014). The operating system provides an interface to allow users to interact with the file system. A widely used API to manage a file system is POSIX (Gallmeister, 1995) — Portable Operating System Interface — and it defines the access operations, namely, read, write, open, close, create and others.

2.2.2 Database management systems

Database management systems (DBMS) provide a way to store data following a specific structure. Depending on the relation between the data records, their transaction support and language used to query data a database can be SQL or NoSQL. Databases are widely used in the cloud to store the state of web applications.

Relational databases

SQL databases management systems (Date and Darwen, 1997), also known as *relational database management systems* (RDBMS), provide a language that allows users to create, read update and delete data records. SQL databases are organized respecting the relational model (Codd, 1970, 1969), which defines a structure and language where all data is represented by tuples related with each other. SQL databases adopt a fixed schema-based structure in which data is stored in tables. Each data record in a table is divided in several columns. A record from a table can share a relationship with another record from another table through a foreign key definition. SQL databases also provide the notion of transactions that allow several database queries to be executed with atomicity, consistency, isolation and durability (ACID) guarantees. In terms of scalability, most SQL databases can only scale vertically, i.e., by adding more computing resources to the same database server. Horizontal scaling, by splitting data across different database servers (sharding), is possible but requires changes to the application code to deal with it.

NoSQL databases

The term NoSQL stands for Not Only SQL and it corresponds to database management systems that do not follow a fixed schema to store data records and may have no support for transactions, at least in the strict sense of ACID transactions found in SQL databases. Their primary goal is to be able to scale more or achieve a better performance than SQL databases. To do so, they use a dynamic structure that can be in the form of documents, key/values, graph or columns. In NoSQL databases data can be registered without requiring a schema to be defined in the first place. Most NoSQL databases offer a relational model that allows data records to share a relation. Developers can implement such relations, but they will only exist at the application level. The concept of transactions with ACID properties is not present in most NoSQL databases. In terms of scalability, NoSQL databases are designed to scale horizontally, i.e., by adding more servers to the database cluster. According to the CAP theorem (Brewer, 2000), in a distributed system it is only possible to pick two out of this three properties: consistency, availability and partition tolerance. To cope with this limitation, No SQL databases typically provide a weaker form of consistency, called *eventual consistency*, that allows some users to read out-of-date records before the replication is fully concluded.

	SQL	**NoSQL**
Relational	Yes	No
Structure	Tables with records organized by columns	Documents, graph, key / value and others
Schema creation	Static	Dynamic
Scalability	Vertical	Horizontal
Transactions	ACID guarantees	Guarantees bound by the CAP theorem
Elasticity	Requires downtime	Online resizing of the database
Some examples	MySQL, PostgreSQL, OracleDB, SQLServer	MongoDB, Cassandra, HBase, Neo4J

Table 2.2: Comparison between SQL and NoSQL database management systems.

Table 2.2 summarizes the differences between SQL and NoSQL database management systems. The first column of the table lists the characteristic of database systems and the following columns present the differences between SQL and NoSQL. Given the heterogeneity of NoSQL database management systems, some characteristics may differ. For example, some NoSQL databases provide ACID transactions at the expense of a performance penalty. However, to simplify the comparison we assume the common characteristics of NoSQL databases as they are the ones most suited to the web applications targeted to applications with a large number of users.

2.3 Web Application Architecture

The common architecture of web applications has changed throughout the years. A couple of years ago it was common to include the entire code of the application in a single server. This monolithic approach required the entire server to be replicated to cope with higher traffic demands. More recently a new architecture pattern has appeared, that decomposes functionalities of the monolith in multiple parts, called microservices.

2.3.1 Monolithic architecture

The monolithic architecture consists in having every component of the application in a single server. In practice, monolithic applications are deployed in a single server that can be replicated to deal with high traffic demands. One of the most common architecture patters in monolithic web applications is the Model-View-Controller (MVC) pattern. This architecture pattern consists in dividing the application in three components: *model*, deals with the state of the applications; *view*, contains the graphical user interfaces; and *controller*, handles the logic of the application and interacts with the user's requests. The typical system architecture of this pattern assumes the existence of an application server, where the code of the application is running and a database server, where the state of the application is stored.

2.3.2 Microservices architecture

The microservices (Thönes, 2015; Dragoni et al., 2017; Newman, 2015) approach consists in having the back-end of a web application distributed in services that interact through the network, as opposed to a single monolith deployed in the same machine or in a cluster of limited size. By adopting this approach organizations are able to organize developers in small teams, each one being responsible for a self-contained component of the application. Such component is called a *microservice* (Newman, 2015; Thönes, 2015; Dragoni et al., 2017) and it communicates with other microservices through an interface accessible over the network using a protocol, such as SOAP (Newcomer and Lomow, 2005) or REST (Richardson and Ruby, 2008). Inter-service communication infrastructures, such as Linkerd (Linkerd, 2018) and Istio (Istio, 2018) are then responsible for dealing with the deployment, monitoring, communication and configuration of groups of microservices, called *services meshes*. This approach allows different kinds of applications (mobile apps, web sites and desktops) to only use the required microservices, instead of requiring a dedicated monolith for each one.

Figure 2.3 compares an example *e-commerce* application designed in the monolithic architecture and in the microservices architecture. The application is composed by: *user authentication*, *shopping cart*, *catalogue*, *billing* and *shipping* components. In the monolithic architecture every component is in the same monolith, meaning that there is a single application server running the entire application.

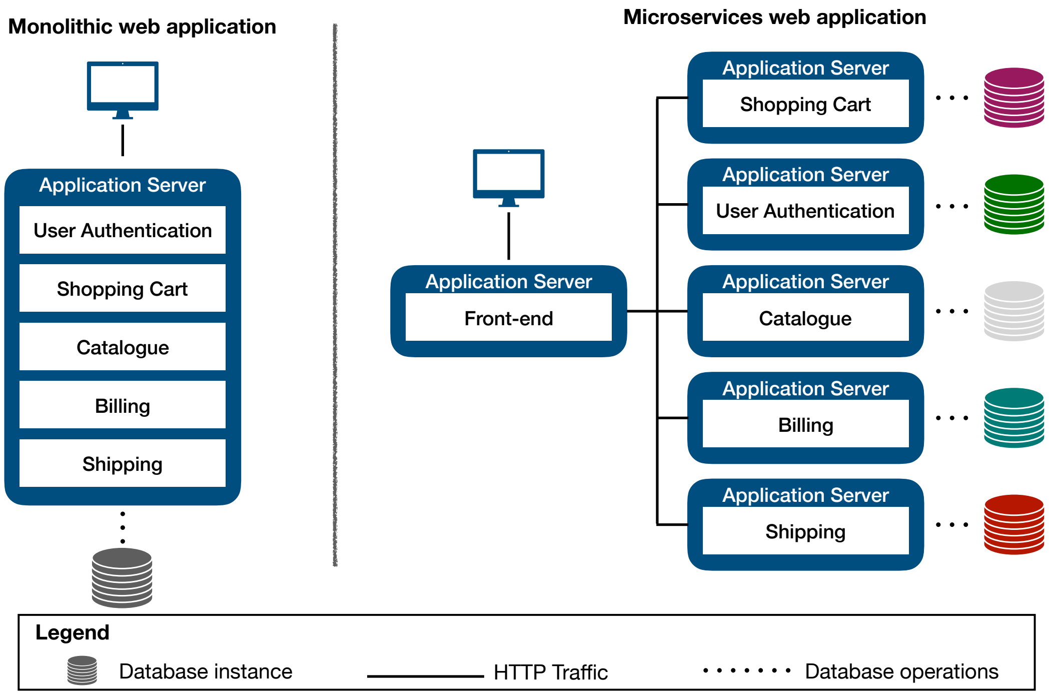

Figure 2.3: Comparison of an application with the monolithic and the microservices architectures.

Every data record is in the same database instance. In the microservices architecture there is an application server for each component — microservice. An additional component, called Front-end, deals with the requests issued by the users and interacts with the required microservices. Each microserver has its own database instance. This allows, for example, the User Authentication microservice to use a NoSQL database while the Catalogue microservice uses a SQL Database. Microservices communicate with each other using a REST API available through the network. Given their distributed architecture, it is possible to scale only specific microservices, while in the monolith in order to scale a single component it is necessary to replicate the entire application server.

2.4 Summary

This chapter gives an overview of some relevant topics related with the developed work that is presented in the following chapters. The cloud computing model can support a wide range of applications. Having a singular intrusion recovery mechanism for the entire cloud is unfeasible given the heterogeneity of each model. In order to recovery from intrusions it is necessary to design tailored solutions that take into account the architecture of the cloud platform and how the users interact with it. In the next chapter we will present the state-of-art on intrusion recovery.

3 Related work

In this chapter we give an overview of the state-of-the-art on *intrusion recovery* systems, which are organized in five groups: *generic applications*, that describe the implementation of intrusion recovery mechanisms for different systems; *virtual machines*, that take advantage of the virtualization platform to perform snapshos and log system calls; *multiversioned file systems*, that keep multiple versions of documents; *file systems with selective re-execution*, that use operation logs to recover corrupted files; and *web applications*, designed for specific database management systems;

There is another related line of research in the topic of *reversible computing* (Toffoli, 1980; Frank, 2005; Morita, 2008; De Vos, 2011) which focuses on the study of invertible primitives and physical reversibility. Some example applications in this field of study involve reversible logic circuits, reversible Turing machines and reversible cellular automata. Although this field explores the reversibility of computing operations, such mechanisms are not suitable for the cloud computing model. These works do not aim to revert corrupted state, instead they focus on generating reversible operations. Given this difference such works will not be discussed in this chapter.

The recovery approach that we are interested in, called *selective re-execution*, works by reverting the system to a previous point in time and reconstruct it by executing the legitimate operations. This approach resembles a line of work known by *operational transformation* (Sun and Ellis, 1998; Sun et al., 2004; Davis et al., 2002; Sun and Sun, 2009) (OT). This technique aims to maintain data consistency in collaborative applications, allowing several users to simultaneously work on common data records using distributed computers connected through the network. This technique allows, among other operations, to update, delete, lock edits and undo operations. The undo operation has a similar goal to the one used to recover from intrusions, which is to revert the effects of a previous operation. However, when intrusions occur they may propagate their effects to different data records, requiring a more delicate undo operation capable of tracking and reverting every data record.

The works presented in this chapter aim to support the reversal of corrupted state of different kinds of distributed applications without discarding legitimate data. This form of recovery, also known as *forward recovery*, contrasts with a different approach, *backward recovery*, that aims to only revert the state to a point in time prior to the intrusion (Figure 1.2 illustrates the differences between both approaches). In this dissertation we only explore the forward recovery approach since backward recovery would discard valid data. In terms of how recovery is performed, there are different approaches but in general all of them are supported in the existence of a combination of *checkpoints* and *message logging*. Checkpoints are copies of the state of the application that can be used to revert the application back to a previous point in time. Logged messages are previously recorded operations that can be re-executed on a checkpoint to reconstruct the state of the application until present time. For file systems, there is an alternative approach that allows several versions of each file — *multiversioned file systems* — that allow corrupted files to be reverted to previous versions without affecting the remaining file system.

3.1 Rewind, repair, replay: three R's to dependability

Intrusion recovery consists in the technique of reverting, or undoing, the effects of unintended actions from the state of the system. The mechanisms presented in the following sections assume that users interact with the system by executing operations which, in turn, modify its state. Users with malicious intent may execute illegal operations that will modify the state of the application, by exploiting vulnerabilities or by accessing the system in other users' behalf. These illegal operations are intrusions that the system administrator wishes to revert. Normal users may also accidentally execute unintended operations and, although they are not intrusions, in the sense they were not purposely executed, they should be undone from the state of the system.

3.1.1 The three R's approach

The three R's to dependability (Brown and Patterson, 2002) consists in a technique that offers a system-level undo operation, offering the possibility to revert unintended actions performed by human operators, viruses, hacker attacks and unpredictable problems that are detected too late to be contained. The motivation behind this work is the fact that human operator error has been the leading cause for outages (Brown and Patterson, 2001; Enriquez et al., 2002; Oppenheimer et al., 2003) and the fact that it takes a considerable amount of time to revert the effects of the error and restore the service. The authors of the article defend that future systems should be designed from the start with recovery mechanisms similar to the undo operations found in widely used applications, such as word processors and spreadsheets.

The three R's approach allows undoing unintended operations without losing user data. This approach poses some challenges: dealing with the external consistency of the system when the recovery process reverts the effects of operations that were already observed by external users; defining what can be recoverable in a system; and building an undo system that is compatible with the multiple levels of the system: distributed system, single computer and user.

The mechanism works in three steps: rewind, repair and replay. In the *rewind* step, the system state is reverted to a previous backup prior to the error. In the *repair* step, a system administrator applies the required corrections to the system: applying a software patch to the system, omitting erroneous operations or fixing a bug in the code. In the *replay* step, the undo system re-executes every user operation since the backup, allowing the corrections added in the repair step to be executed as well.

Recovery happens after the effects of an intrusion are recorded in the state of the system and made available for external users. It is not possible to ensure that no user saw the effects of the intrusion before recovery completes. This problem of external insconsistency, as discussed in the literature (Elnozahy et al., 2002), does not have a solution, but it is possible to manage such situation. One solution proposed by the authors consists in using compensating or explanatory actions to inform the user about the inconsistencies he may experience.

The three R's approach is capable of undoing human operator errors. For that it is necessary to record executed operations at the user level, instead of recording state changes. This makes preserving the state and implementing the corrections in the repair phase easier, since the system administrator will have to reason about user actions instead of state changes. Given that the target systems of this approach provide their interface through standardized protocols, such as SMTP, IMAP, JDBC/SQL and XML/SOAP, it is possible to collect user intent actions by recording protocol level operations. The proposed architecture consists in having a proxy intercepting every user-lever operation so they can be recorded in an operation log. A *time-travel* storage unit is then responsible for making periodic backups in order to allow rewinding the system to a previous point in time.

3.1.2 An implementation of the three R's approach

The three R's approach is a generic mechanism that allows system administrators to revert unintended actions from the state of the system. An implementation of this mechanism in a distributed system is Operator Undo (Brown and Patterson, 2003). It offers the undo operation in e-mail systems so that a system administrator can revert unintended actions. In the paper the authors present the design of Operator Undo as a generic system, allowing it to be adopted by many enterprise and Internet service applications. The authors describe the implementation of Operator Undo in an e-mail server with the SMPT/IMAP protocols for message delivery and retrieval.

The system architecture is similar to the one presented in (Brown and Patterson, 2002) with the addition of a Control UI that serves as an interface for system administrators. Figure 3.1 presents the system architecture of Operator Undo. In the figure, the *time-travel storage* and the *service application* (the blue components) belong to the application (with some minor modifications) that is being protected by Operator undo.The *undo proxy*, *control UI*, *undo manager* and the *timeline storage* (the green components) are from Operator Undo. The proxy, used to intercept user's requests, is specific to the application it is wrapping. This allows the proxy to identify and register operations at

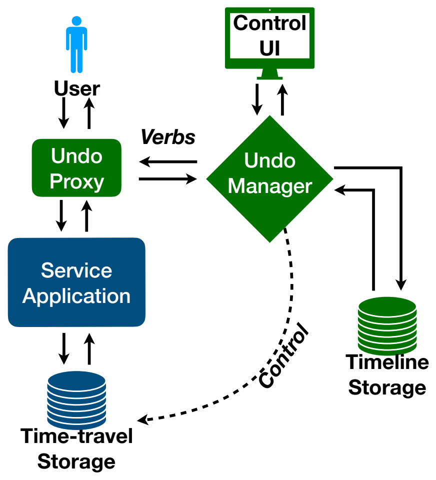

Figure 3.1: System architecture of Operator Undo.

the user level, making it possible for the administrator to select and undo user operations, instead of storage level operations. While the proxy adopts the same protocol as the application, to identify the user level operations, the remaining components of the system assume a generic interpretation for the operations. This allows porting Operator Undo to different kinds of systems by requiring only the proxy to be modified.

Each operation provided by the application has a corresponding *verb*, an object understandable by Operator Undo, with an interface that provides some functions required in order to keep consistency. The operations added by the verb can be grouped in two sets: sequencing and consistency-management. The first set, sequencing, helps the undo system components to order operations when they arrive concurrently, otherwise the proxy might record the concurrent operations in a different order than the one executed by the application. To cope with this situation, the proxy serializes operations that should be executed in a specific order and adds this ordering parameters to the meta-data of the operation, allowing non-dependent operations to be executed concurrently. The second set, consistency-management, has three functions: a comparator that compares the output of the operation with the output of the replay action; a compensating operation that is executed with the differences returned by the comparator; and a function used to order the execution of operations when they participate in a chain of inconsistent verbs.

The implementation of Operator Undo in an e-mail service was made without code modifications to the e-mail server; instead the components of Operator Undo are setup to wrap the e-mail server. In the implementation the authors identified the operations provided by the email server and implemented, for each of the operations, a tailored verb. To do so it was necessary to identify which operations change the state, interact with external state and which ones are asynchronous. The proxy required an *UndoID* to be added to each message header so that Operator Undo is capable of ordering and correlating messages. The compensation operations that deal with inconsistencies observed by the user, such as missing e-mail messages, are solved by informing the users that maintenance was done. The *time-travel* storage performs automatic backups hourly, which are converted in daily backups at the end of the day. This allows to rewind the system to, at most, a day prior to the intrusion, before repairing and replaying the verbs.

In another work (Brown et al., 2004), the authors present an evaluation of human-assisted recovery processes and tools. In their experiments the use of a recovery tool, like Operator Undo, reduces the downtime of an application due to recovery from around 10 to 3 minutes.

3.2 Virtual Machine Intrusion Recovery

With virtualization it is possible to create virtual computer hardware, storage, network and execution environments. Some intrusion recovery mechanisms take advantage of the capabilities of virtualization to log system level operations and provide recovery at the system level. The following works explore the problem of intrusion recovery in servers that run in *virtual machine appliances*, i.e., pre-configured virtual machines that run on a *hypervisor*.

3.2.1 Tracking intrusions through the operating system

A system administrator needs to analyze how an intrusion propagates through the operating system in order to perform the adequate recovery measures. If done manually, this task can be an arduous and time consuming requiring administrators to explore the logs of operations and received network packets. A tool that deals with this problem is BackTracker (King and Chen, 2003). It assists system administrators in identifying the sequence of steps that occur in an intrusion. BackTracker works backwards from the detection point, i.e., it starts from the state in the file system that alerts the administrator of the intrusion, and identifies a chain of events that could have led to the erroneous state. During runtime BackTracker collects logs of events in the system. When a system administrator wishes to analyze a detection point, BackTracker creates dependency graphs with the chain of events that cause the state modification.

BackTracker observes events, such as system calls, and the objects affected by them: files, filenames and processes. An event induces a dependency relationship between two objects when one object affects the state of another object. A dependency relationship is characterized by three parts: a *source object*, a *sink object* and a *time interval*. The source object is, for example, the file that was read before its data was written in another file — the sink object. The time interval is a counter of events, not related with the time of the system clock. BackTracker identifies three types of dependencies: *process—process* dependency, occurs when one process interacts with another by creating it, sharing memory with it or signaling it; *process—file* dependency, happens when a process affects or is affected by data from files; *process—filename* dependency, occurs when a process affects of is affected by the name of a file. The component of BackTracker responsible for deriving a graph from the registered dependencies is denominated GraphGen. GraphGen asks the system administrator to identify a detection point in the system, for example a file the administrator knows was corrupted, and a time at which the administrator suspects the attacker corrupted the file. With this information, GraphGen reads the log of events and follows the dependencies backwards, starting from the detection point to the beginning of the log.

The component in BackTracker responsible for logging events and objects during runtime is called *EventLogger*. BackTracker runs the target operating system (Linux-based) in a virtual machine and schedules the virtual machine monitor to report events to the EventLogger. The log of events is stored in the machine hosting the virtual machine, to ensure that intruders cannot access it.

The dependency graph can be simplified to make it easier for the system administrator to analyze the succession of events that lead to the detection point, i.e., the intrusion. BackTracker simplifies the dependency graph by applying five rules. First, it ignores certain objects that would be present in any graph of dependencies. For example, when the bash shell initiates it reads the bash history file and loads it into memory, this event, if not ignored, will be in the root of every dependency graph and it has little to do with the intrusion. Second, it filter some events, such as low-control events (for example, changing a file's access time or creating a filename in a directory). These first two rules can be deactivated since, in some attacks, they may be relevant. Third, it hides files that have been read but not written in the time interval given by the system administrator. Fourth, it filters helper processes that take input from one process, execute a simple task and return an output to the main process. These helper processes cause cycles in the dependency graph and do not help analyzing the effects of the intrusion. And fifth, BackTracker allows the system administrator to choose several detection points, which are used to calculate the intersection of the dependency graphs of those detection points.

In conclusion, BackTracker does not provide recovery mechanisms to revert the effects of intrusions, but it offers a forensic tool that helps system administrators analyzing the trace of events of an intrusion in the system. This tool can be used together with a file system recovery mechanism to effectively eliminate the effects of an attack.

Some system administrators would benefit from an automated recovery system that, besides tracking the effects of intrusions, would also perform recovery. Bezoar (Oliveira et al., 2008) is an OS and application independent intrusion recovery system that is capable of recovering full system state while imposing a low overhead to the virtual machine. The system works by performing periodic checkpoints of the system, logging all architectural non-deterministic events and tracking how data network propagates in the memory. A memory monitor registers the trace of network data in memory. When an attack is detected, the system rolls back the state to the latest backup prior to the source of the attack and replays the legitimate operations, while ignoring the ones caused by the attack. The Beazoar system relies on a memory monitor, a detection service, a recovery unit and a storage for the logs and checkpoints.

The memory monitor tracks operations that affect the memory and uses a symbolic memory space to map the propagation of network data from a specific source (IP address and port number) in the memory of the system. For every new frame that enters the system from the network card, a new entry in the symbolic memory is created with the source address and an integer key. These keys will follow every memory operation and will be stored in the symbolic memory along with the data they touch, allowing it to be traced. A garbage collection mechanism cleans data sources that do not have memory entries associated with.

Logging of operations is done by a modified version of ExecRecorder (Oliveira et al., 2006) to allow recovery by removing the re-execution of malicious events. The periodic checkpoint includes the complete system state but it uses a copy-on-write approach to reduce the required storage. This is achieved by executing a fork operation that duplicates the VM process along with a *SIGUSR1* signal that suspends the parent process. This allows the parent process to become the checkpoint while the child process continues normal execution. When recovery needs to be done, the father process is woken up by a *SIGUSR1* signal and the legitimate logged operations are re-executed.

To start the re-execution of legitimate operations and recover the system, the administrator needs to identify the source of the network by selecting the network source identification in the memory unit. Then, for each operation in the log, Bezoar compares the nework source of the operation with the one pointed out by the administrator. If they are different then they are executed since they did not originate from the network intruder. If they are equal then the operation is not executed and the system starts a *semi-replay* phase. In this phase the algorithm presented in (Brown and Patterson, 2002) is executed, i.e., a selective re-execution of the legitimate and non-tainted operations. The difference between Bezoar and (Brown and Patterson, 2002) is that Bezoar introduces the notion of a semi-replay phase which performs replay concurrently with the normal mode, allowing interrupts and inputs events to be processed with the exception of network traffic. When every valid operation in the log was executed the system enters a new valid state in which the intrusions have never occurred.

3.2.2 Online recovery with quarantined objects

Bezoar provides online recovery by performing recovery concurrently with the normal execution of the system. This technique maintains availability during recovery, however, it fails to contain the propagation of the intrusion. A system that recovers the effects of an intrusion, avoiding having it infecting other legitimate files in the system is SHELF (Xiong et al., 2009).

SHELF is an *on-the-fly* intrusion recovery system that provides business continuity, availability and recovery accuracy. SHELF maintains a cumulative clean state of the system for applications and files affected by the intrusion, allowing them to resume execution with the most recent version prior to the attack. SHELF uses taint tracking techniques to assess the damages caused by the intrusion and quarantine methods to contain infected files, allowing uninfected process and files to be available to the users. SHELF provides three unique features. First, preservation of the accumulated useful states for applications and data, allowing to revert to a previous version in case of an intrusion. Second, backward and forward taint tracking techniques that allow to track the damages caused by an intrusion. Third, *on-the-fly* recovery that allows uninfected files and applications to be available while infected ones are being recovered.

SHELF was implemented on top of a light-weight virtual machine, performing most of its functionality at the Hypervisor layer. This provides a transparent execution environment for the users and imposes a minimal interference to the guest system.

SHELF runs in three phases: in the *normal* phase, SHELF does periodic state recording for each object in the system and logs operations that will allow it to track interactions and analyze dependencies between objects; in the *damage assessment* phase, SHELF determines infected objects based on an intrusion source and the dependencies recorded during the normal phase; in the *recovery* phase, the infected objects are quarantined, i.e. deactivated to avoid having them infecting other objects, and reverted to the latest, non-infected, version.

SHELF runs at the hipervisor layer and the host layer. It has a similar system architecture to the one presented in (Brown and Patterson, 2002) and has the following components:

- *state recording and restore module*: responsible for keeping accumulated state of each object in the system;
- *logging and reconstruction module*: tracks dependencies between objects during the normal execution of the system. It works in normal execution logging operations and auditing system calls;
- *dynamic damage assessment engine*: performs a backward / forward taint tracking that iterates through the logged operations in order to determine how the intrusion propagated in the system;
- *quarantine enforcer*: applies a deny policy that regulates access to suspect objects;
- *recovery engine*: recovers quarantine objects by reverting them to the latest accumulated state prior to the attack source and re-executes the logged operations.

3.3 Intrusion Recovery in Multiversioned File Systems

Intrusion recovery in a file system can be done by reverting the affected files to the latest legitimate version. This technique imposes two requirements: one, the file system keeps multiple versions of files; and two, attackers cannot tamper with the previous versions of the files. The following systems concern mutliversioned file systems that were designed to recover from intrusions.

3.3.1 Recovering from intrusions in a multiversioned file system

Legacy file systems did not allow user to revert files to previous versions. Once a file was deleted or updated the previous version would become unavailable. A system that adopts an alternative approach is the Elephant File System (EFS) (Santry et al., 1999). This file system retains every version of each file allowing users to revert unwanted operations by recovering a previous copy of the affected files. By keeping every single version of any file in a secure storage, a user is able to revert any type of attack that corrupts his data. However, no storage system has an unlimited capacity and some policy has to be followed to decide the number of versions of each file that should be kept. Each type of file may require different levels of versioning. The authors of EFS identify the following file types: *read-only*, such as executables, have no version history; *derived*, like object files that can be later regenerated from their original source, requiring no versioning; *cached*, for example web browser cached files, can be discarded; *temporary* have short lifetimes and may benefit from short-term histories but it would not be useful to keep a long-term history of them; and *user-modified*, require a history of versions but each file may require a different levels of versioning.

In order to recover corrupted files, the file system should fulfill two requirements: first, users should be able to undo changes to recent versions, and second, it is important to keep a long-term history of important files. In order to keep the history of short-term user's files with the limited storage space, a time policy can be implemented. For example, keeping every version of a user's files for one hour, a day or a week. For long-term files a different approach based on landmarks can be adopted. In this approach, users adopt a format similar to version control systems, such as Git or SVN, that allows them to mark important versions that can be later recovered, while the intermediate versions cannot.

In practice EFS provides four policies to keep previous versions of files: *keep one*, acts like a standard file system, keeping only the most recent version; *keep all*, maintains every version of the file; *keep safe*, maintains short-term history of files allowing users to undo recent versions; and *keep landmarks*, that allows users to keep short-term and long-term history of the files by defining checkpoint versions.

EFS allow users to define landmarks of versions, but it also implements an automatic mechanism to do so. This strategy protects users against their own mistakes. The automatic mechanisms works as follows: in the short-term, every version of the file is kept, then an automatic routine cleans up versions that should not be necessary for the user because they are too similar. In other words, in a short time frame every version is important for the user, but the longer it passes the less important closer versions are to the user. For older versions only the most recent version of each batch of updates should be relevant for the user. This allows the automatic routine to propose intermediate versions to be discarded.

EFS allows intrusion recovery by keeping every version of the stored files, but it was not designed with the purpose of intrusion recovery. Instead, it was designed to allow users to revert their mistakes. One can argue that the same mechanism can be used with intrusions, however, in an intrusion the malicious user is not available to inform the administrator about the files that were tampered. For intrusion recovery, a file system capable of diagnosing and repairing the system from attacks is required.

S4 (Strunk et al., 2000) is a self-securing storage server that allows system administrators to analyze the effects of intrusions and recover from them by reverting files to previous versions. It uses a log-structured object system to keep versions of the stored objects and a journal-based structure for the versions of the metadata.

S4 provides two features to the system administrator useful to overcome intrusions: diagnosis and recovery. Diagnosis allows the system administrator to assess the damages caused by the attack. S4 performs analysis in three phases: detection, discovery of the vulnerability and determination of the damages. Recovery reverts the damages caused by the attack by restoring the previous, not corrupted, versions of the affected files.

An *history pool* keeps previous versions of files which can be restored through a *copy forward* operation that copies a previous version to the current one. Old versions of the files are only kept for a specific window of time called *detection window*, after which old versions are garbage collected and cannot be recovered. Attackers may try to fill the history pool in order to force the garbage collection to discard previous versions and, as a result, make recovery impossible. To cope with this, S4 applies monitoring mechanisms that detect abuses and purposely impose some latency in the system, allowing system administrators to deal with the attack while keeping the system available for the legitimate users. The history pool employs an access control mechanism that prevents unauthorized users to delete previous versions of files. Only the users, owners of the files, and the administrator can revert files to previous versions, and only the administrator can delete previous versions. Access to previous versions is done using the same interface to access files with an extra time parameter.

S4 is a network-attached storage (NAS) for objects accessible to users through a RPC interface. Users store objects, i.e. files, in a abstraction of a folder called drive. The only metadata they can append to a file is a name. The Remote Procedure Call (RPC) interface provides the CRUD (Create, Read, Update and Delete) operations. The interface is similar to other network-attached storage servers, except for the read operations that allow a time parameter to access the correct version of the file.

The S4 system uses a *client daemon* that translates file system requests to S4 requests. This allows S4 to be deployed in an existing file system without modifying the code of the file system. The *audit log*, an append-only audit log records all requests for each object. For each operation the audit log saves the operations arguments and the client that issued the request.

S4 provides a multiversioned object storage feature to existing network-attached storage servers. The recorder logs during normal execution allow the system administrator and the users to analyze and revert unintended actions. The recovery approach is based on reverting the affected files to previous versions, as opposed to re-executing legitimate operations.

3.3.2 An intrusion recovery plugin for existing file systems

In some organizations it is complex and costly to update or completely substitute existing systems. This can happen because of: licensing, lack of know-how or complexity of the infrastructure. This is a problem to system administrators who want to be able to recover their systems from intrusions. One solution for this problem is the Repairable File Service (RFS) (Zhu and Chiueh, 2003).

RFS adds a repair feature to an existing shared file service. It was designed for network file servers with intrusion tolerance techniques that speed up the process of repairing from security breaches and human errors. RFS provides roll-back features that allow any file to be reverted to previous versions and keeps track of inter-process dependencies to determine the damages caused by an intrusion.

RFS is not a file system *per se*; instead it is a framework that protects existing network file systems by reverting the effects of undesired operations. During normal operation of the file system, RFS records write operations and inter-process dependencies to a log. When undesired operations need to be undone, RFS selectively rolls back the affected and contaminated files. The design of RFS provides three key features: first, it does not require software modifications to existing file servers, second, it interferes as less as possible with the performance of the file system, third, its design is modular allowing it to be implemented in different file access protocols.

The RFS framework is composed by a Request Interceptor, responsible for recording user's operations to a *Log Collection*. The Mirror NFS Server interacts with the clients and offers the exact same interface as the protected NFS Server, acting as a proxy. The Contamination Analysis and Repair Engine performs the repair according to the propagation of the damages. Syscall Loggers run on the clients to log client logs, as an addition to the write operation logs captured by the Request Interceptor. Both logs complement each other to gather information, such as process id and system calls related to the process creation, that will alow the Contamination Analysis module to find and track dependencies. There are some similarities between this architecture and the one presented in (Brown and Patterson, 2002), namely the Request Interceptor, the Log Collection and the Contamination Analysis and Repair Engine which have a similar purpose to the Undo Proxy, the Timeline Storage and the Undo Manager respectively.

RFS provides two possible recovery approaches: forward recovery, which works by rolling back the system state to the most recent cleaned snapshot and selectively re-executing legitimate operations; and backward recovery, which works by undoing contaminated operations until the system is cleaned. RFS uses backward recovery, this way it does not require periodic snapshots to be performed. To do so it calculates *undo records* which are operations that revert the file to the previous version.

RFS considers a process contaminated if it is a child of a contaminated process, if it reads contaminated files or if it reads or writes attributes from contaminated files. When undo operations are executed they are also logged, meaning that they can be undone in the future.

3.4 Intrusion Recovery in File Systems with Selective Re-execution

In the previous section we presented works that explore multiversioned file systems. These systems allow administrators to recover from intrusions by reverting affected files to previous versions. Another technique that can be used in file systems consists in adopting the three R's approach, i.e., reverting the state or affected files to a previous point in time, repair the system and replay every legitimate operation. The following works perform intrusion recovery in file systems using this approach.

3.4.1 Tracking intrusions using taint tracking techniques

One technique used to track the effects of intrusions in the system is by marking the affected files and propagating the mark throughout every file that shares a relation with it. For example, when a marked file is read by a process and then the process writes to another file. This technique is called *tainting*, and the marked files are called *tainted*. In a file system, in order to apply the taint tracking technique it is necessary to monitor files and processes and keep track of the operations in which they interact. One system that does this is Taser (Goel et al., 2005b).

Taser is an intrusion recovery system designed for file systems. Taser uses taint tracking techniques to mark processes and files when they are read and written, creating a dependency graph. Taser faces two main challenges: first, it is not trivial to isolate intrusion operations since both malicious and legitimate users' operations affect the same files of the system; second, it is in the best interest to keep the effects of legitimate users' operations in the system even if they depend on tainted operations.

Taser operates in three modes: *auditing*, *analysis* and *recovery*. The *auditing* mode uses Forensix (Goel et al., 2005a) to audit files, processes and sockets and relate them to the file system's operations. It also allows to replay previous operations. The *analysis* mode tags files, processes and sockets as tainted based on the information gathered by the auditing mode. The *recovery* mode allows to revert the effects of tainted operations by selectively replaying only the legitimate operations. Taser allows the system administrator to chose one of two possible sets of rules to taint operations — a more conservative or a more optimistic. The more conservative set of rules will taint any operation that interacts with a tainted object, this will also mark legitimate operations as tainted. The more optimistic rules will allow the administrator to ignore certain operations which will keep their effects in the system after recovery. The conservative rules will facilitate the recovery process, while the optimistic rules will make recovery more difficult due to the conflicts between the legitimate and malicious operations. To cope with this, Taser provides an automatic conflict resolution mechanism that separates file system's operations into name, content and attribute operations, allowing specific recovery approaches for each type.

The Taser system is composed by three components: the auditor, the analyzer, and the resolver. The auditor runs during normal execution of the file system and logs executed operations. It tracks three kinds of objects: files, processes and communication sockets. The analyzer runs when the system administrator want to recover from an intrusion. It uses the logs collected by the auditor in order to determine the set of tainted file systems that were affected by the intrusion.

Like the analyzer, the resolver also runs when the administrator wants to recover from an intrusion. It uses the operations logs collected by the auditor and the tainted objects calculated by the analyzer and uses this information to revert the affected files to the previous version, prior to the intrusion, and selectively re-execute legitimate operations.

A dependency exists when information passes from one object to another via a system call operation. Rules define how dependencies are identified. A process to process dependency is caused by operations between processes, for example a *fork*. A process to file dependency happens when a process write to a file. A file to process dependency happens when a file is executed or read by a process. A process to socket dependency exists when a process writes to a socket. Finally, a socket to process dependency happens when a process reads from a socket. Taser follows these rules in order to taint dependent objects when the source is tainted. The tainting algorithm uses the logs, the dependency rules and a detection point to derive a set of tainted objects. The detection point is the source or result of the attack that was pointed by the administrator. The tainting algorithm works in two phases: a *tracing phase* and a *propagation phase*. The tracing phase processes the sequence of operations in a reverse order, starting from the detection point to the start of the log. As a results, the tracing phase will present the administrator a set of objects denoted as possible attack sources, allowing the administrator to select the ones responsible for the attack. The propagation phase starts with the attack sources and follows the sequence of operations forward, from the attack source to the detection point. In each operation it follows the dependency rules to taint the affected objects. The process stops when there are no more operations to mark as tainted.

The resolver aims to undo the effects of tainted operations while preserving the legitimate, non-tainted, ones. To do so it requires a file system snapshot prior to the intrusion, a set of tainted objects and the log with the executed operations. The recovery mechanism follows a selective re-execution approach that executes the legitimate operations on top of the snapshot version of the file. Only write operations that succeeded during normal execution of the system are executed. Failed operations and read operations are ignored. Taser provides two recovery algorithms: a simple redo algorithm and a selective redo algorithm. The simple redo algorithm executes the legitimate operations on top of the snapshot of the entire file system. Legitimate operations are defined taking into account the dependency rules of Taser. This algorithm can be costly, in terms of time, if there are many operations to execute. The selective redo uses a smarter approach. It first checks if the non-tainted objects are legitimate; if so it only reverts the affect files to the version present in the snapshot and only re-executes the legitimate operations that affect these objects. Another optimization consists in separating operations by their nature: file name operations, content operations and attributed operations. This allows Taser to discard every operation before the last legitimate one, given that the last one dictates the current state of the object.

3.4.2 Quarantining untrusted files

While Taser is capable of tracking the effects of intrusions and recovering from them, it does so at the cost of keeping logs with every executed operation in the system. This can be costly in many systems and given that only some processes may be responsible for intrusions, keeping logs with every operation is a waste of space. For example, processes from applications downloaded from the Internet are more likely to cause intrusions and, therefore, should be logged. A different approach would be to keep logs only of the operations that can be performed by an attacker. One way to achieve this it by having isolation environments for suspect files and processes. This way the logger would only monitor operations from the suspect, and isolated, environment. One system that adopts this approach is Solitude (Jain et al., 2008).

Solitude is an application-level isolation and recovery system that, besides simplifying the intrusion recovery process, also limits the effects of attacks by using a restricted privilege isolation environment to run untrusted applications. This strategy is different from common file systems that share the same namespace between every user and process in the system. Instead, Solitude uses its own file system, called IFS, that uses a copy-on-write technique to provide an isolation environment for each untrusted application. If a user detects that an application is malicious, he can discard the application's IFS environment without losing valid legitimate data from other users and applications in the file system. To avoid limiting the functionality of applications or making the system more difficult to use, Solitude provides support for restricted privileges to run server applications and policies for explicitly sharing files between an isolated environment and the remaining files stored in a trusted base file system. It may happen that a user misjudges an application and configures a sharing policy that allows files from a malicious application to be shared with the base file system and, with that, contaminates the user's files in the base file system. To cope with this Solitude implements a taint propagation layer between all applications of the file system that allows to track the effects of files shared by an isolation environment and the base file system by recording how other applications access those files. A system administrator can then use Solitude to analyze the intrusion and recover from it

The IFS isolation environment provides applications with a transparent view of the file system, restricting file changes with a copy-on-write policy. Using a isolation environment provides some benefits: in case of an intrusion the user can simple discard the isolation environment without losing his files, the propagation of the intrusion is limited to the isolation environment, and it allows untrusted applications to read files in the base file system without requiring them to copied to the isolation environment.

An isolation environment is created by mounting the copy-on-write IFS in a directory of the base file system. Then, the untrusted application is confined to the mounting point by executing a *chroot*[1] operation. IFS provides sharing modes that limit the capabilities of the applications in reading files — by allowing or denying to read or write files in the base. Before files are synchronized from the IFS to the base file system — *commit* — a *snapshot* is done to copy them to the log.

Some of the policies provided by IFS can be poorly configured by the users, resulting in untrusted applications being able to affect user's files in the base file system. To cope with this Solitude uses a modified version of Taser (Goel et al., 2005b) to track how applications in isolation environments access files that are committed or write shared and, with a taint propagation method, log their actions. In each isolation environment there is a monitor that performs the file operations of behalf of the processes.

The taint monitor synchronizes files from IFS to the base during a commit. Solitude marks the monitor as tainted, then the taint propagation algorithm tracks every modification in the base file system that depend on the monitor's changes. Only the base file system is subject to the tainted algorithm since, in case of an intrusion, it is the base file system that will be recovered. The algorithm follows three rules: taint a process when it reads or executes a tainted file; taint a file when a tainted process modifies the file; taint children processes of a tainted process.

Solitude logs two kinds of operations: first, operations caused by a tainted object; second, operations caused by the commit process. The logs are stored and analyzed in a different system so they do not get corrupted. These logging policies differ from the ones present in Taser, which logged

[1]chroot is a Linux command that changes the root directory of the current process and its children, so that they cannot access files outside of that directory. This is also called a *jail root*.

every operation in the system since it assumed that all operations were untrusted. After an attack, the copy-on-write isolated environment can be discarded. The operations that occur in the isolated environment do not need to be logged and, as a result, create a smaller log.

When the user detects that a process shared a malicious file from the isolated environment to the base file system, he selects the file from the IDS and the synchronization commit as a starting point for recovery. Then, the recovery process generates the tainted dependency graph. The affected files are manually verified to ensure the correctness of recovery. Then the files are rolled back to an untainted state, which was marked in a snapshot right before the commit from the IFS. Finally, every operation that affected the files is re-executed until the before moment the commit, marked by the user as the intrusion, was done.

3.4.3 Configurable recovery algorithms

A recovery algorithm can undo the effects of intrusions in different ways. By following the three R's approach, using a multiversioned file system or using a refined algorithm that reverts affected files without modifying the legitimate ones. *Back to the Future* (Hsu et al., 2006) is an intrusion recovery framework that provides two different recovery algorithms for the system administrator.

Back to the Future allows malware removal and, as a result, repairs the system by undoing the effects of the malware while keeping every valid effect caused by legitimate operations intact. Although this work focuses on malware removal, it can also be applied to recover from intrusions, as both attacks modify data. Malware is implemented and sent to a target system by a user will malicious intent and, when successfully executed, it damages the system state by executing unintended actions. The framework, as the name suggests, works by reverting the state of the system to a point in time prior to the intrusion and then bringing the trusted processes back to their pre-recovery state. The framework uses sandbox environments to bound the scope of untrusted processes in the system. These untrusted processes are defined by the user. If he misclassifies a trusted process as untrusted there is no harm except for a performance degradation.

The framework is composed by three components: a monitor, a logger and a recovery agent. This architecture is similar to the one presented in (Brown and Patterson, 2002, 2003). The monitor is responsible for intercepting system operations and sending them to the logger, which records them in stable storage. When an intrusion is detected, the recovery agent triggers and coordinates the recovery process. The monitor has also an auditing function that detects when an untrusted program violates the integrity of the system and, if so, it automatically invokes the recovery agent to restore the system integrity. Although this detection feature is part of the framework, it deals with a different problem than the one explored in this work, so we will not go into much detail.

The integrity model followed by the framework is inspired by Bibas's integrity model (Bishop, 2003). This model states that no process can read objects of lower integrity level, and no process can write objects of higher integrity levels. The authors define two integrity levels for Back to the Future: *trusted* and *untrusted*. By following the model, trusted processes cannot read untrusted data and untrusted processes should not be able to write to trusted data. To avoid having the system staled unnecessarily because an untrusted process needs to write trusted data that will not be read by a trusted process, the framework adopts a relaxed integrity model, called lazy Biba's model, that does not enforce integrity until the point where untrusted data will flow into trusted processes.

According to the authors, a good approach to intervene when a trusted process is about to read untrusted data is to preserve the consistency of the process by either allowing the process to fulfill its purpose or completely block it, and to run the process as long as possible until it cannot preserve consistency of the system. To fulfill this intervention approach, the authors propose three options to preserve system integrity: deny the operation to read untrusted data; allow the read operation by terminating the untrusted process and reverting the affected data to the latest trusted state; allow the read operation by marking the trusted process as untrusted.

Back to the Future provides two approaches to recover from intrusions: a basic approach and a refined approach. The basic approach works by monitoring every operation in the system and, after an intrusion is detected, undoing all the logged write operations of every process (trusted and untrusted) and redoing all the logged operations for the trusted processes. The problem with this approach is that it requires every trusted process operation to be logged and reverted, which takes time and requires more storage space for the log.

The refined approach avoids undoing and redoing write operations that do not need to be recovered. The authors present two examples to justify the refined approach: when a trusted process writes an object before and untrusted process and vice-versa. According to the authors, it should only be necessary to undo untrusted operations when they occur after trusted operations, since in the other case the trusted process legitimates the file by writing on it. This also changes the way logging is done. If an untrusted process writes to trusted data then the log should record the previous value and mark the file and operation as untrusted. When a trusted process writes a file, the log marks it as trusted and does not log the operation.

3.4.4 Intrusion recovery using selective re-execution

RETRO (Kim et al., 2010) is an intrusion recovery system for desktops and servers. During normal operation, RETRO logs user's operations in an *action history graph*, which describes in detail the system execution of operations triggered by users. When a system administrator detects an intrusion he selects the faulty operation (or operations) issued by the attacker, for example, a TCP connection or an HTTP request, and starts recovery. RETRO performs recovery using *selective re-execution*, in other words, it rolls back the state of the system to a point in time prior to the intrusion, then it re-executes every valid operation in the log. The system administrator is responsible for dealing with any conflict that arises during recovery. This way the effects of the intrusion are wiped out of the system while every valid state modification remains intact.

The action history graph used by RETRO allows it to not only undo the attacker's actions but also repair legitimate users' actions that were performed as a consequence of the attack. For example, if the attack consisted in modifying a collection of file that were later changed by legitimate users, then, during recovery, RETRO will discard both the attacker's actions as well as the users modifications, that should not have been done if the attack did not occur in the first place.

RETRO faces the challenge of undoing faulty actions while preserving the state so that legitimate user's actions are not lost or corrupt. To do so it follows these four strategies:

- the action history records objects in the system, such as files and process, and the actions that cause state modifications to these objects. *Refinement* allows representing each object in different levels of abstraction to gather information specific to files that is not present in processes;

- RETRO re-executes actions respecting their dependencies in the action history graph. This means that if an legitimate user's action was performed based on erroneous information generated by an intrusion, then the legitimate actions should be executed with the corrected data;

- RETRO uses *predicates* to reason about which actions should be re-executed. This allows recovery to avoid re-executing actions that result in the same state modifications before recovery;

- RETRO uses *shepherded re-execution* to stop the re-execution process when the process state reaches the state of the original execution (like when the process executes an identical *exec* call).

RETRO has a system architecture that is similar with other intrusion recovery systems discussed in this chapter. Besides the log, repair controller, managers and checkpoint storage, RETRO also appends a module to the kernel of the operating system and some extra libraries to the processes. The checkpoint storage stores a period snapshot of the system. The log records two kinds of objects: *data objects* — files — and *actor objects* — processes. Each action in the log is correlated based on their dependencies with other actions. Garbage collection mechanisms discard old logs to reduce the required space for RETRO. Discarded log operations cannot be recovered, so a trade-off must be calculated to select the appropriate required storage and the longevity for recovery. Like Operator Undo, the repair controllers of RETRO provide a generic API allowing it to deal the heterogeneity of the different processes. Also like Operator Undo, RETRO deals with external inconsistencies by applying compensating actions when possible or asking the administrator to provide a solution when necessary.

After rolling back the system state to a point in time previous to the intrusion, the recovery starts. First, malicious operations in the action history graph are replaced by benign ones. For example, a faulty operations that consists in appending data to an existing file then is replaced with a similar operation that appends zero bytes to that file. Second, RETRO re-executes the operations in the action history graph, ignoring those that maintain the same state after execution.

3.5 Intrusion Recovery in Web Applications

Most web applications store their state in databases and cloud storage offerings. Intrusion recovery systems for web applications monitor HTTP requests, which are then used to undo the effects of unintended actions. In this section we present works regarding intrusion recovery in web applications.

3.5.1 Intrusion recovery through transaction support

Ammann et al. (Ammann et al., 2002) explored the problem of recovering a database that was partially damaged by an attacker, requiring a special recovery approach capable of undoing the effects of the attack while keeping every benign transaction intact. In their paper the authors propose a framework designed to be built on top of an off-the-shelf Database Management System (DBMS). The proposed framework consists of: the *Damage Assessor*, that tracks the corruption caused by the detected transaction; the *Damage Repairer*, that fixes the located damage with specific operations; the *Damage Confinement Manager*, binds data items that were marked as untrusted by the Damage Assessor; the *Policy Enforcement Manager*, with two distinct responsibilities, works as a proxy for normal users of the database, and enforces intrusion tolerance policies.

The recovery framework has two responsibilities: *damage assessment* and *damage repair*. The main challenge in repairing a database is, as the authors state, the phenomenon of *damage spreading*, that consists of having malicious transactions creating corrupted data that once read by legitimate transactions will be propagated throughout the database state. To overcome the attack, the recovery mechanism may affect the availability of the database by requiring a *coldstart*, in which the system is halted while the damages are being reverted, or a *warmstart*, in which the system is not halted during recovery but there is a significant performance degradation. A *hotstart* would allow recovery in a transparent way by not halting nor degrading the performance of the system. The goal of the proposed framework is to locate every record of corrupted data and revert the damaged records. The framework provides two recovery mechanisms that offer a coldstart semantic and another that offer a warmstart semantic.

By definition, recovery violates the principle of durability in transactions, since it reverts committed transactions that should be kept in stable storage after successful execution. To overcome this violation of the durability property, the authors propose the concept of an *undo transaction* that revokes the effect of previously committed transactions, allowing the system to satisfy the durability property. In practice the concept works as follows. A top level transaction, called *malicious activity recovery transaction* (MART), is placed at the beginning of the history of transactions, making every subsequent transaction a child of MART. This nested transaction model allows, by the properties of database transactions, MART to undo or compensate the any child transaction as long as MART is not committed. The problem with this approach is that eventually MART needs to be committed, otherwise it would grow indefinitely. Aborting MART would result in reverting the entire state of the database to the point in which MART was defined. To overcome the memory problem, MART is committed when the system is recovered. In other words, after intrusions are removed from the system, it is assumed that every transaction in the history is now clean, so a new MART can be started from that point while the previous MART is committed.

A transaction T_2 is dependent on another transaction T_1 if it reads a data item that was created by T_1. The transitive relation also applies, meaning that any other transaction T_2 that depends on T_2 is indirectly dependent from T_1. This dependencies are taken into account by the recovery algorithm, which marks every transactions with some dependencies to a bad transactions also as bad. This dependency rules help the recovery process in two ways: first, it does not require good transaction that do not depend on any bad transaction to be undone and re-executed; and second, only the effects caused, directly or indirectly, by the intrusion are undone.

For *coldstart recovery* the authors propose two algorithms: two pass repair algorithm and repair algorithm based on separate read log. The two pass algorithm, as the name suggests, works in two steps. In step one, the algorithm scans the log forward, starting from the entry caused by the intrusion, and locates every bad and suspect transaction. In step two, the algorithm goes backwards to undo all bad and suspect transactions. The problem with this algorithm is that the default database logs do not include read operations which would require some modifications to the existing database systems.

The repair algorithm, based on separate read log, overcomes the problem of the previous algorithm. It separates the read operations from the write operations in two distinct logs, instead of modifying the database to record every operation in the log. It uses a mechanism to schedule the order of operations in both logs, then it uses the two pass algorithm to repair the database.

The previous algorithms only allow recovery by halting the system while the damage transactions are being undone. The on-the-fly repair algorithm allows *warmstart recovery* and was designed for database systems and applications with high availability requirements. In this algorithm the *Repair Manager* analyzes the logs to mark any bad or suspect transaction and submits the undo transaction to the *Scheduler*. The Scheduler receives operations from both the *Repair Manager* and the users of the database. The *Recovery Manager* executes operations received from the Scheduler and logs them. The on-the-fly repair algorithm also needs to deal with the confinement of corrupted data created by bad transactions. The algorithm marks every suspect transaction concurrently with the user's requests, then it should prioritize undoing bad transaction over the execution of user's operations.

Akkuş et al. (Akkuş and Goel, 2010) present a recovery system for web applications that use databases to store their state. One of the key challenges of undoing unintended application requests is the difficulty in tracking the corrupted data and its dependencies. As the authors explain in the paper, at the database level there can be different dependencies that cause corrupt data to be propagated in the database: a chain of queries that read a record and update another record, dependent transactions and foreign keys that correlate record in different tables are some examples. Although other works in the literature deal with the problem of corrupted data being propagated in a database, they lack mechanisms that take into account the several layers of web applications. The recovery system proposed by the authors applies two methods to analyze dependencies: a combination of application replay with offline taint analysis for deriving application-level dependencies, and a fine-grained field-level dependencies as opposed to row-level dependencies.

The presented recovery system was designed for web applications that follow a layered architecture, with a presentation layer that interacts with users, a logic layer with the required code to process user's request and interact with the database, and a database level where the application's state is stored. The system is composed by a proxy that logs application level requests and two components that analyze data before recovery. Unlike other intrusion recovery systems for web application that only track operations at one level, in their system the logger captures requests at all three layers.

Logging is done at the three levels of the web application: presentation, logic and database. The database logger converts every query in a transaction and serializes the transactions ensuring that they can be reordered in the future. Analysis is done in a sandbox and uses data collected during logging to calculate three types of dependencies: at the database level, the application level and user level. At the database level the system assumes that two queries are dependent if one reads the same record produced by another. This allows to create a graph of dependencies in which the queries are the node connected by edges that represent the database rows records. When an administrator selects one or more initial requests that cause the intrusion, the analyzer traces the dependency graph. At the application level, dependencies are analyzed by replaying the target requests in the sandbox environment. By using a taint-based interpreter the system is capable of associating the requests with the corresponding database record, this way it is possible to correlate the received application requests with the corresponding database queries. At the user level, the system collects information available in the session cookies which allows the administrator to identify which user caused which actions. It also makes it possible to group requests by user and, if necessary, undo everything caused by a specific user — the attacker.

Recovery is done by executing compensating operations to the application state. The compensating transactions are calculated by computing a reverse query that when executed reverts the database row to the previous value. These compensating transactions are executed in reverse order in the database undoing, version by version, every undesired update.

The classic system model of web applications assumes the existence of a database in which the entire state of the application is stored. Data in the database is produced by statements that are generated by an application server that receives HTTP requests. In other words, attackers perform malicious HTTP requests that will result in one or more malicious database queries that need to be undone. One system that adopts this model is Warp (Chandra et al., 2011).

Warp is a recovery system for web applications that use a database to store their state. Unlike other recovery systems, Warp does not require the administrator to detect and identify the intrusions from the application's requests logs. Instead it allows the administrator to retroactively apply software patches to fix vulnerabilities which, after recovery, mitigate the effects of the attacks that exploit them. Warp works by logging every application request and database operation. With this information, Warp is able to construct dependency graphs that correlate the requests and operations.

Warp provides three innovative techniques: first, it does not require an administrator to select the original request caused by the intrusion, instead it allows to retroactively apply software patches to fix vulnerabilities, undoing every intrusion that exploited them; second, Warp used a *time-travel database* that correlates database queries allowing it to perform partial recovery, which only reverts the part of the system that was targeted by the intrusion, without requiring the entire database to be reverted; third, Warp uses a client-side browser extension that records and uploads events to the server and performs replay.

Recovery in Warp starts when the administrator applies a security patch to the application's code. Then Warp rolls back the system to a checkpoint prior to the time the administrator wants to apply the patch and replays every action that happened after the checkpoint. Replay conflicts created during recovery, such as having legitimate user edit operations in pages created by the adversary, have to be solved by the administrator. The Warp system reuses two components from Retro (Kim et al., 2010): the action log and the repair controller. Besides these components, it also includes a browser with an extension, a versioning database, an HTTP log, an application runtime and a manager that is attached to the repair controller.

During normal execution of the application, Warp logs user's requests and database queries, collecting three types of dependencies: first, Warp collects an input dependency for the HTTP request and an output dependency for the HTTP response; second, for each read and write in the database, Warp collects input and output dependencies; third, it collects dependencies between the HTTP request and the source code files of the application that respond to that HTTP request. To deal with non-deterministic functions, such as timestamps and random number generators, Warp records the result of the execution of those functions, avoiding to replay them during recovery.

To start repair with a retroactive patch, the administrator needs to identify the affected source code file, or files, and a time indicating when the patch should be applied. Warp then registers the patch in the log in the correct time frame, uses the dependencies graph to partially revert the state of the application and re-executes the operations in the graph.

Re-execution of the operations is done in the same way as the original execution with two exceptions: Warp does not replay operations that do not affect the state of the application and non-deterministic functions are executed in a different way, which reuses the calculated values from the original execution.

The use of a time travel database has two goals: allows re-executing only a portion of the total number of database queries in the log, and it allows recovery while the application is online serving users. To reduce the number of queries to be re-executed, Warp performs fine-grained rollback at the level of rows in the table. This means that instead of reverting the entire table it is only necessary to revert the affected rows. Another technique is the use of table partitions which are the fractions of the table that are affected by queries. A partition can be identified by the columns specified in the *where* clause of a query. The time travel database stores every version of every row in the database. This allows to rollback the affected rows and re-execute queries in the same version as the original execution. There is a detail that needs to be taken into account: queries that use a *where* clause with a criteria to filter rows need to be modified to ensure that during re-execution the exact same set of rows is affected. This is done by modifying the original criteria for a criteria that uses row IDs (for example, *where acount_balance > 150* is substituted with *where row_id in (1,54,32)*).

Having a time travel database that adopts an append-only approach is costly in terms of space, so periodic routines perform garbage collection by discarding old versions of the database records.

To allow concurrent recovery with normal operation of the application, Warp adopts a *repair generation* approach which consists in creating a fork of the current version of the database — *current generation* — which will be identified as *new generation*. User operations are forwarded to the current generation while recovery is being processed in the new generation. If users execute operations that affect the part of the database being recovered in the current generation, those operations are scheduled to be executed after replay. Once recovery finishes, every change in the current generation not present in the new one is copied and the new generation takes the place of the current one.

Using the browser to re-execute operations allows Warp to correlate client requests with server operations. The browser extension does this by generating a unique ID to each user (*clientId*) and an unique ID to each request performed by the user (*visitId*). This way, Warp can correlate user requests by ordering them. To correlate the HTTP requests issued by the browser with the ones processed by the server, Warp adds a HTTP *requestId*. The three generated IDs are appended to the HTTP request header and logged by the server. To avoid having malicious users trying to fill the log with garbage (for example, by executing a large amount of requests) to trigger the garbage collection mechanism and corrupt the log, Warp keeps a log with an individual storage quota for each user. Replay of HTTP requests is done in the server by a browser with a re-execution extension. This execution is done in a sandbox that only grants access to a specific client's cookie. Alternatively it is possible to re-execute requests in the client side by an administrator. This is an useful feature to undo some unintended action caused by an administrator by accident.

3.5.2 Propagate recovery to external web services

The previous works presented in this section assume a system model in which the web application is running in an isolated environment. Although this may be true for many web applications, in some cases this system model does not apply. Some web applications need to interact with external web services during their execution. In this kind of system, intrusions may propagate through several services requiring the recovery process to repair every affected service. One system that assumes this system model is Aire (Chandra et al., 2013).

Aire is an intrusion recovery system designed for applications composed by interconnected web services. Aire runs in each web service gathering information about the received and sent requests in order to track dependencies across services. Aire performs recovery locally by rolling back the state and selective re-executing every valid operation. Once it finishes, it propagates the recovery actions to the dependent web services. Aire performs repair in a distributed manner, as opposed to having a central coordinator managing the web services. This is because of the distributed nature of this kind of application, as web services do not have a strict hierarchy that allows the definition of a coordinator. Another reason is that during recovery some services may be down, which would delay a centralized recovery process. Instead, each service executes recovery immediately and queues the recovery messages to be propagated when the target services are online. With this strategy it may happen that some services are repaired while others are still recovering, which lead the system to a partially repaired state. To cope with this the authors of Aire propose a contract with the developers that states that recovery requests should be indistinguishable from concurrent requests issued by the users of the application.

The Aire system is composed by the following components: logging proxy, repair log, versioned database, repair controller and a request proxy. The proxies collect and send to logs the received operations. The repair controller coordinates the recovery process in each web service. The request log stores every executed request. Moreover, a versioned database, also found in Warp (Chandra et al., 2011), stores every version of a database record, allowing the affected records to be reverted before repairing them.

Each web service that can be recovered by Aire provides an external interface for recovery that allows the administrator and other web services to trigger recovery requests. The operations provided by this interface are: *replace*, to substitute a previous request with a new one; *delete*, to remove a previous request; *create*, to execute a request in the past; and *replace_response*, to replace the response of a previous request. The delete and replace operations, when executed in a web service, undo the effects of previous requests and, if necessary, propagate a replace or delete request to other web services. The absence of a global time for every service brings a challenge for the administrator to specify a timestamp for the new request. To cope with this, the client that executes the create operation must provide the web service with two, previously executed, requests indicating that new requests should execute them. Repairing a response is not straightforward since the server usually does not know the address of the client, as it is the client that initiates the connection. Furthermore, the client usually authenticates the server through his certificate, not the other way around, which means that the server cannot authenticate the identity of the client. Aire solves this by requiring the client to specify a URL that should be used by the web service to send a token identifying the new response. Once the client receives the notification of a replacement for a previous response it starts a connection with the web service using the posted token, allowing the web service to provide the correct response. In order for this to work there must be a way for clients to identify previous requests unequivocally. To do so, Aire appends metadata to the HTTP headers that will allow clients to identity each request.

To avoid having a local web service from being stalled waiting for other web services, Aire employs a strategy that consists in throwing a *timeout exception* when a repair requires an external web service. This should not be a problem since applications should be prepared to deal with this kind of situations. Once the remote web service finishes repair it will execute a *replace_response* that will allow local web service to finish recovery. Requests that need to execute in other web services are queued.

Aire provides access control to the repair mechanisms in order to avoid creating an interface that allows malicious users to attack the system. This is an interesting feature given that most of the other recovery services presented in this chapter assume that the recovery mechanism cannot be accessed by unauthorized users. This access control requires two functions to be implemented by the application: *authorize*, that verifies the credentials of a request, and *notify*, the notifies the application when an request fails to due unauthorized credentials or timeouts.

By executing repair requests asynchronously, Aire exposes a partially repaired state to the users which may appear invalid for the users of the application. Although this could be problematic in tightly coupled distributed systems, with web services this is something that the application developers already expect, given the loosely coupled nature of such applications. One example of a web service application that handles concurrent operations in a similar manner is Amazon S3 (Robinson, 2008). In S3, if two concurrent operations access the same object, a last-writer-wins semantics is employed. Some applications require stronger invariants and cannot rely in partially repaired states. To cope with this, Aire provides a feature similar to Git branches (Torvalds and Hamano, 2010). When a past request needs to be repaired a new branch is created, allowing the repair to take place while the current value remains intact.

3.5.3 Recovering web applications in the cloud

The cloud computing model facilitates how administrators deploy their applications, but also limits the functionality of the application servers. More specifically, in the Platform-as-a-Service model, the system administrator has access to an execution environment with automatic scaling capabilities. This allows him to immediately deploy an application in a container without requiring him to install the application server, database and other services. However, the administrator cannot modify the execution environment. This complicates the process of setting up an intrusion recovery mechanism, since most of them require heavy and system specific modifications in order to work. An intrusion recovery system that was build with this limitations in mind is Shuttle (Nascimento and Correia, 2015).

Shuttle is an intrusion recovery service designed for web applications deployed in PaaS offerings. Shuttle works by logging HTTP requests that are issued by the users of the application. Periodic checkpoints of the state of the web applications allows it to be reverted to a moment prior to the intrusion so that it can be recovered by selectively re-executing valid HTTP requests.

The system model defined in Shuttle assumes that intrusions can be caused by two kinds of situations: *software vulnerabilities*, that allow malicious users of execute invalid state modifications to the application's state, for example, by performing a SQL injection attack; and *faulty requests* that are issued by authorized users, for example, when a hacker obtains the access credentials of an authorized user. Shuttle works in two modes of operations: *normal operation* and *recovery*. In normal operation the application serves the users while Shuttle collects operation logs and performs periodic checkpoints of the application's state. These checkpoints are created without interrupting the application. To do so Shuttle employs a *copy-on-write* strategy in which an object is only replicated when it is being written. In recovery, Shuttle creates a branch of the application loaded with a snapshot of the application created before the intrusion. It then re-executes the valid requests in this branch while the application still continues to serve users. When recovery finishes the branch replaces the main application.

The Shuttle system is composed by the following components: *proxy*, *load balancer*, *application servers*, *database instances*, *Shuttle storage*, *manager*, *replay instances*. The proxy, Shuttle storage and manager are components with the same function as the ones present in the previous systems discussed in this section. The load balancer is part of the PaaS and it allows to route requests to the corresponding replica the application servers, which are also part of the PaaS. The database instance contains a database proxy that logs database operations so that it is possible to gather dependencies between operations. The replay instances are HTTP clients that will be used by Shuttle to re-execute the valid operation during recovery.

To recover from intrusions, Shuttle provides the administrator with the following actions: fix the application software, identify the tampered records in the database, modify the logged requests, and launch a cleaned database and application server. If the administrator opts to change the code of the application he must keep the same interface, in order to ensure that no error arises during the re-execution of requests. If he chooses to select tampered records in the database, then Shuttle will calculate the dependency graph that will ensure that no valid information is lost or corrupted.

The dependency graph consists of interconnected nodes that represent requests, and edges that represent the relations between the requests. Dependencies are established based on two rules: first, a request R_B depends on another request R_A if there is a data object x that was read by R_B and last modified by R_A. Second, dependencies are transitive. Dependencies are calculated periodically in background. Like other works presented earlier in this chapter, there is a garbage collection routine that discards graph dependencies prior to the last backup. As pointed by the authors, one problem with this approach is that it may lead to false dependencies. This happens if the value read by R_B is never used. False positives in detecting dependencies may delay the recovery, but do not corrupt the state of the application. The alternative would require changing the code of the application interpreter (Python, PHP, etc.), which would limit the adoption of Shuttle to few applications. False negatives in the graph dependencies may also occur. This may happen if some object y is deleted by R_C and later another operation R_D tries to access y, which will result in R_D falling. If during recovery, the administrator chooses to keep y will not be dependent on R_C. To avoid this Shuttle records every operation in the system, including those that fail.

Shuttle performs replay in clusters of requests which are executed concurrently. Requests were originally executed concurrently, i.e., when a request starts executing before the other one finished, are replayed concurrently. Concurrent operations may result in non-deterministic situations. This happens when concurrent operations access and modify the same object. To cope with this, Shuttle allows developers to use an interface to solve conflicts. To avoid generation of non-deterministic values, such as random IDs, Shuttle provides a deterministic random number generator in its API based on the current timestamp. This way, during replay it is possible to generate the same random numbers as the ones generated during normal execution.

For replay Shuttle allows the administrator to select from *full replay* and *selective replay*. Full replay requires every request in the log that occurred after the last backup to be re-executed. This can take a considerable amount of time so it should only be used if the intrusion happened recently, at most a few days. Selective replay requires the administrator to mark the malicious requests. Then, Shuttle gathers every tainted request, i.e., every request issued by the same user or any request that uses accesses the data modified by it. With these requests it is possible to mark the affected data and restore it from the latest valid backup. Finally, every request that affected the data is re-executed, undoing the effects of the intrusion. To deal with inconsistencies observed by external users, Shuttle provides an API that allows developers to apply compensating operations. This API is composed by three operations: *preRecover*, *handleInconsistency* and *postRecover*. Replay can be done in runtime, meaning that it does not require the application to be offline while recovery is being processed.

3.6 Summary

This chapter presented some of the related work in the field of intrusion recovery for different kinds of applications. Each one of the systems was designed for a specific end, such as file systems, virtual machines, web applications, databases and email servers. It is not possible to adapt these intrusion recovery systems to different applications. However, these papers motivated and inspired the work presented in this thesis.

In Table 3.1 we summarize the recovery strategies adopted by each recovery system. In the table, the intrusion recovery mechanisms appear in the first column, the target system for the recovery mechanism is in the second column, the third, fourth and fifth columns correspond to the adopted recovery strategy employed by the mechanism. The sixth column presents how the mechanisms deals with external inconsistencies observed by the user and finally, the last column indicates if the mechanism is capable of recovering the system without requiring it to be offline.

Recovery System	Designed for	Selective re-execution	Compensating operations	Multiversioned	External inconsistencies	Online Recovery
Operator Undo (Brown and Patterson, 2003)	Email systems	X			Compensating operations	No
Backtracking (King and Chen, 2003)	Virtual Machine	X			Not mentioned	No
Bezoar (Oliveira et al., 2008)	Virtual Machine	X			Not mentioned	No
SHELF (Xiong et al., 2009)	Virtual Machine	X			Not mentioned	Yes
EFS (Santry et al., 1999).	File Systems			X	Not mentioned	Yes
S4 (Strunk et al., 2000)	File systems			X	Not mentioned	Yes
RFS (Zhu and Chiueh, 2003)	File Systems		X		Not mentioned	Yes
Taser (Goel et al., 2005b)	File Systems	X			Compensating operations	No
Back to the Future (Hsu et al., 2006)	File Systems	X			Not mentioned	No
Solitude (Jain et al., 2008)	File Systems	X			Compensating operations	No
Retro (Kim et al., 2010)	File systems	X			Compensating operations	No
Amman et al. (Ammann et al., 2002)	Databases		X		Not mentioned	Yes
Ekin et al. (Akkuş and Goel, 2010)	Web Applications		X		Not mentioned	No
Warp (Chandra et al., 2011)	Web Applications	X			Compensating operations	Yes
AIRE (Chandra et al., 2013)	Web Applications		X		Parallel recovery (like Git)	Yes
Shuttle (Nascimento and Correia, 2015)	Web Applications	X	X		Compensating operations	Yes

Table 3.1: Recovery strategies for each system.

Figure 3.2 presents a diagram that relates the ideas of the presented papers with the developed work. In the figure, the light blue boxes represent the papers presented in this section, the obround boxes represent the ideas or approaches that inspired the developed work, represented as dark blue boxes in the center. For simplicity, some ideas and works appear repeated, so that the lines connecting the ideas with the developed work would not overlap.

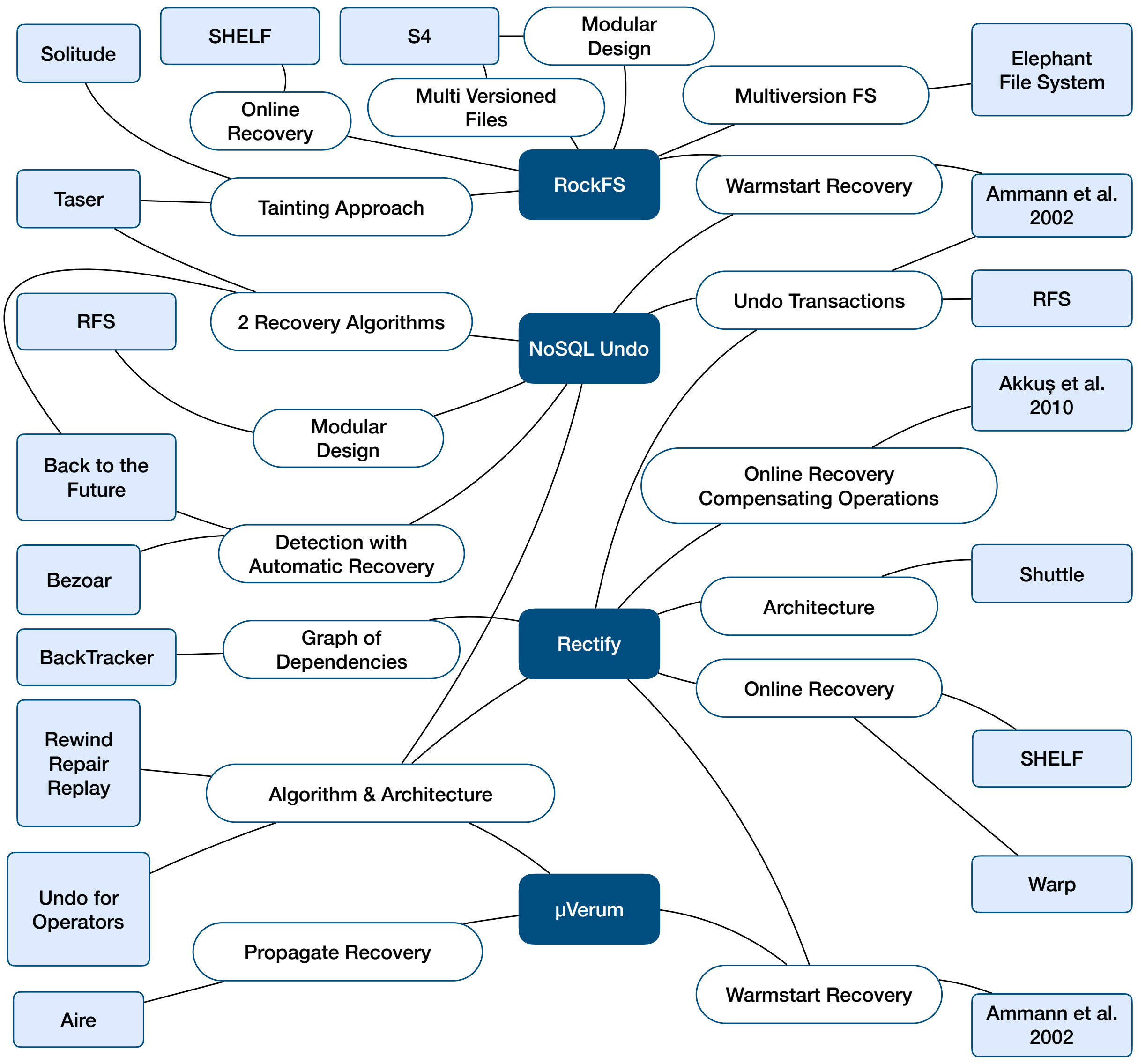

Figure 3.2: Relation of the state-of-the art with the developed work.

4 RockFS: Cloud-backed File System Resilience to Client-Side Attacks

In this chapter we present RockFS (RecOverable Cloud-bacKed File-System), a cloud-backed file system framework that aims to make the client-side resilient to attacks.[1] RockFS adds protection mechanisms to data in the client device and allows undoing unintended file modifications. Several solutions have been proposed to enhance *cloud storage security* (Kamara and Lauter, 2010; Abu-Libdeh et al., 2010; Bessani et al., 2011; Vrable et al., 2012; Bessani et al., 2014; Zhao et al., 2015; Dobre et al., 2014; Pereira et al., 2016). An interesting approach is to encrypt and store data in several clouds, forming a *cloud-of-clouds*, allowing to improve data availability, while ensuring data integrity and confidentiality, even if some of the clouds fail (Abu-Libdeh et al., 2010; Bessani et al., 2011, 2014; Zhao et al., 2015). Some of these systems, and others that are not so concerned with security and dependability, may be designated *cloud-backed file systems*, in the sense that they provide client-side software that provides a POSIX file system interface (Gallmeister, 1995). This approach allows users to use seemingly local files that transparently store data in the clouds (Rizun, 2018; Vrable et al., 2012; Bessani et al., 2014).

Despite all these benefits, a weakness remains: *the client device*. These devices store user credentials that if stolen or compromised may lead to violations of confidentiality (files are disclosed), integrity (files are modified), and availability (files are made inaccessible).

Both file systems that store data in a single cloud and in a cloud-of-clouds assume that the client-side — the user's device or proxy used to access the cloud — is secure (Abu-Libdeh et al., 2010; Bessani et al., 2011, 2014; Rizun, 2018; Vrable et al., 2012; Zhao et al., 2015). This is a strong assumption considering that in many occasions users leave their personal devices unattended, use weak passwords,

[1]RockFS was developed in the context of the european project SafeCloud and can be found at: https://github.com/inesc-id/SafeCloudFS.

have outdated systems with known vulnerabilities, or fall prey to social engineering (Stanton et al., 2005; Cloud Security Alliance, 2012). Once an attacker gains access to a user account, he can read, modify or delete any file that the user has access to. Even if the file system stores several versions of the files, the attacker may delete the old versions or revert a file to the version prior to the attack, discarding a significant amount of work. Moreover, there is a reasonably recent surge of *ransomware* that encrypts device data, potentially making the user unable to use his credentials and access the files stored in the cloud(s) (Kharraz et al., 2015; Yalew et al., 2017).

The *RockFS framework* is a set of components that allow improving client-side security of cloud-backed file systems. RockFS provides two sets of security mechanisms to be integrated with the client-side of a file system: a *recovery service* capable of undoing unintended file operations without losing valid file operations that occurred after the attack; and *device data security mechanisms* to safely store encryption keys reducing the probability of having the credentials compromised by attackers and to protect cached data.

In relation to device data security mechanisms, backup and encryption schemes are widely adopted for protecting credentials in endpoints, but not secret sharing that both protects and allows recovering credentials (Shamir, 1979; Blakley, 1979; Schoenmakers, 1999), and not for cloud-backed storage. This chapter presents new protections and demonstrates their usefulness by leveraging an existing cloud-backed file system. Specifically, the RockFS prototype is based on the Shared Cloud-backed File System (SCFS) (Bessani et al., 2014), which itself is based on the DepSky Byzantine fault-tolerant cloud-of-clouds storage service (Bessani et al., 2011).

Figure 4.1 recapitulates the cloud architecture presented in Chapter 2. In the figure we can see that RockFS is running in the lowest level of abstraction model, IaaS, more specifically in the storage services. Therefore, RockFS allows recovering files stored in the cloud and, as a result, recover applications that use cloud storage to keep their state.

4.1 RockFS

RockFS improves cloud-backed file systems in the two following ways: it provides logs that allow an administrator to analyze the usage of the file system and to recover user files stored in the clouds by undoing unintended operations; it secures user data that is stored on the client-side, namely, access credentials to the cloud service providers and cached files by encrypting this data with a key that is distributed using secret sharing.

4.1.1 Threats

We assume attackers cannot access the cloud back-ends, however they may gain total access to the user's device, therefore RockFS focuses on three security threats not addressed by current *cloud-backed file systems*, as they target the client device:

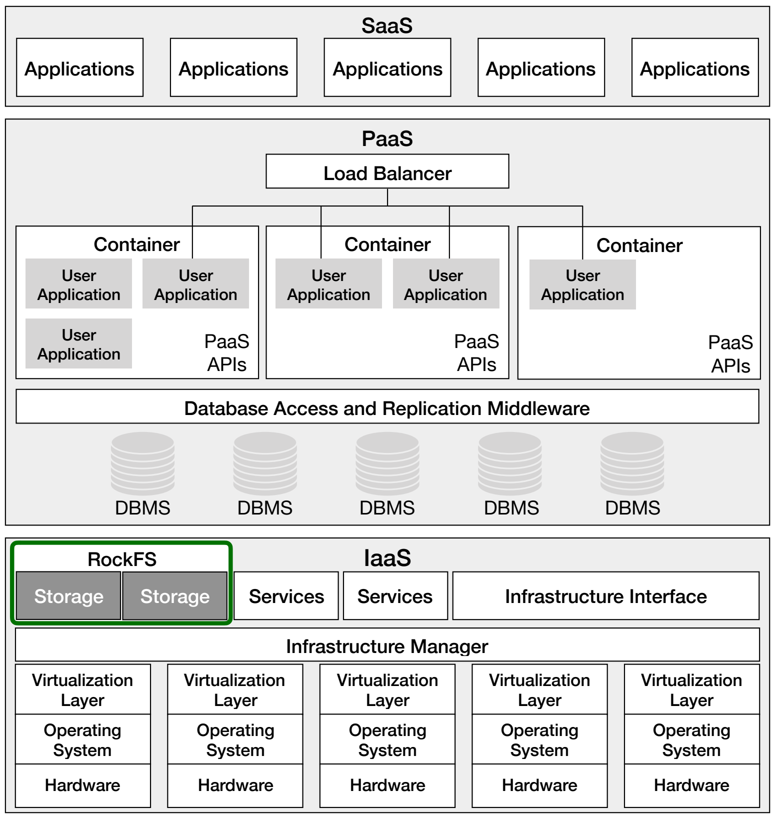

Figure 4.1: RockFS in the common cloud architecture.

- **Threat T1:** *Adversary illegally modifies files in the cloud*, e.g. with a ransomware that overwrites the data files in the client device, which eventually are synchronized to the cloud.
- **Threat T2:** *Adversary prevents a user from accessing the cloud*, by corrupting or deleting the access credentials stored in the client.
- **Threat T3:** *Adversary accesses the locally cached files*, when he gets access to the user's device, as the client cache is not encrypted.

4.1.2 System model

We assume that the client device may be compromised by an attacker that may get access to any data that is on the disk. On the contrary, we assume that adversaries have no access to data in memory. In relation to the services running in clouds, we make the same assumptions as cloud-backed file systems, e.g., that the cloud is not compromised (Rizun, 2018) or that no more than a threshold of clouds is compromised (Bessani et al., 2014).

We assume that communication over the Internet is done using secure channels that ensure communication authentication, confidentiality, and integrity (e.g., using SSL/TLS). We make the standard assumptions about cryptography, e.g. that encryption cannot be broken in practice, that hash functions are collision-resistant, and that signatures cannot be forged.

In RockFS there are two main actors:

- *Users:* contract cloud services to store files and access them remotely using a personal computer or a mobile device.

- *Administrators*: operate RockFS, i.e., maintain the coordination service, monitor the usage of RockFS, and trigger recoveries.

The authentication of both users (PU_U, PR_U) and administrators (PU_A, PR_A) relies on *asymmetric keys* (PUblic, PRivate). We assume that only the owner of a key pair knows the private key, whereas public keys are known and accessible to every user.

The access control to the cloud storage relies on *access tokens* (t_l, t_u). Access tokens are temporary credentials generated by cloud storage providers that authorize users and applications to use specific actions of their services without sharing passwords. In RockFS we use an access token for the log (t_l) that only authorizes to append data, and another one (t_u) that authorizes users to read and modify files but not the logs. These access tokens will typically include the identifier of the user, an expiration date, an identifier of RockFS and a signature to ensure integrity. We assume that these tokens are generated by the cloud storage providers in a non-predictable manner and that it is not possible for an attacker to re-use expired tokens. RockFS requires several tokens and keys to ensure integrity and confidentiality of data. The list is presented in Table 4.1.

4.1.3 System architecture

RockFS can be deployed using a single cloud — Figures 4.2 , or using a cloud-of-clouds — Figure 4.3. On the top left of each figure there is a cloud storage service, where the data is stored (e.g., Amazon S3, Google Drive, etc.). On the top right of each figure there is a *coordination service*, which is responsible for storing logs and other metadata. It is possible to deploy each replica of the coordination service in a distinct cloud (cloud-of-clouds architecture), improving the availability of the service. RockFS may be deployed with coordination services like ZooKeeper (Hunt et al., 2010) or DepSpace (Bessani et al., 2008). The client agent, on the bottom, is a middleware component that runs on the client-side and communicates with the storage cloud and the coordination service, trans-

Entity	Notation	Description	Generated by	Stored in
User	PU_U PR_U	Public key of user U Private key of user U	User U in setup	Shared: user's device, coordination service and external storage
	A_i B_i	ith log entry secret key ith log entry secret key	RockFS agent	Coordination service
	t_l t_u	Access token for log entries Access token to manage files	Cloud providers	Shared: user's device, coordination service and external storage
	SC_i CC_i	Cloud storage service credentials Coordination service credentials	User and admin in setup Admin in setup	Shared: user, admin, coordination service and external storage
	S_U	Session key of user U for local cache	User U	Shared: user's device, coordination service and external storage
Admin	PU_A PR_A	Public key of admin A Private key of admin A	Admin A in setup	Shared: admin's device, coordination service and external storage
Clouds	PU_{SC_i} PR_{SC_i}	Cloud storage service public key Cloud storage service private key	Admin in setup	Each cloud storage service
	PU_{CC_i} PR_{CC_i}	Coordination service public key Coordination service private key	Admin in setup	Each cloud hosting a coordination service replica

Table 4.1: Keys used in RockFS with description, who generated them, and where they are stored.

parently for the file system user.

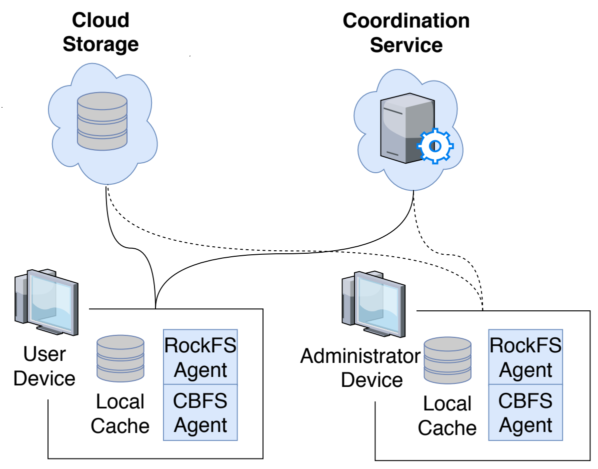

Figure 4.2: RockFS architecture with a single cloud storage provider and coordination service.

Both RockFS' users and administrators access the cloud storage and coordination service, but with different privileges: users only access their files; administrators access the files and also the logs used to store recovery data. The users interact with the file system by invoking the POSIX operations (e.g. open, read, write and close). The administrator accesses logs to analyze usage and recover from unintended actions.

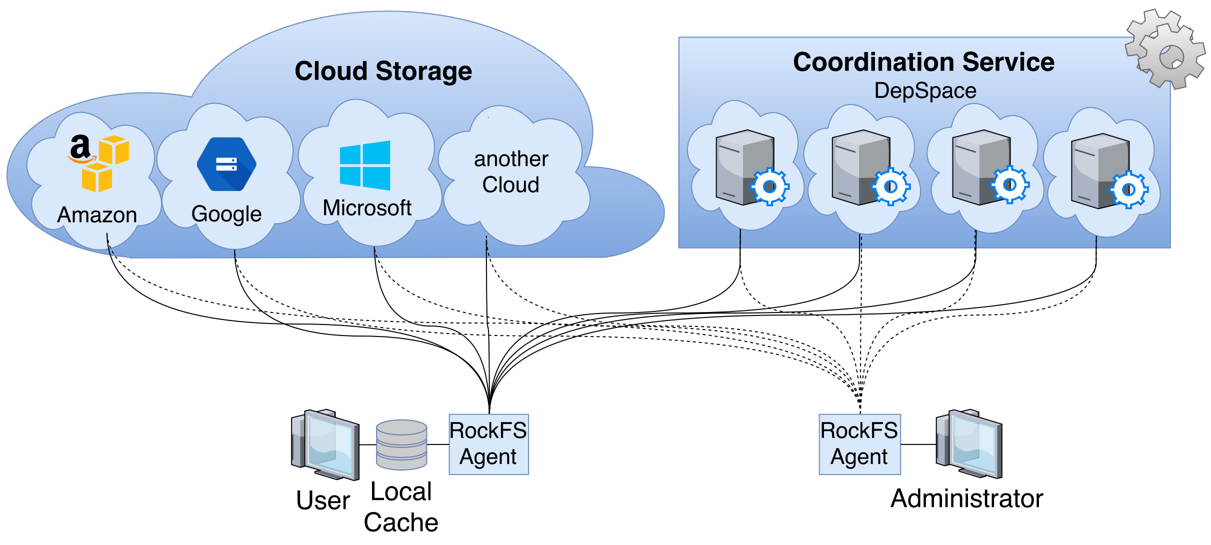

Figure 4.3: RockFS architecture with four different cloud storage providers and four replicas of the coordination service.

The interaction between the client, the cloud storage services and the coordination service is mediated by the RockFS agent and the cloud-backed file system (CBFS) agent, that are responsible for intercepting file system operations with the aid of the FUSE library[2]. The RockFS agent performs several encryption and encoding tasks.

[2]Filesystem in Userspace, https://github.com/libfuse/libfuse

4.1.4 Client architecture

The user device contains the RockFS agent, the local cache and the keystore, including the private key of the user (PR_U). In order to access the cloud(s), the RockFS agent needs the access credentials of each cloud storage provider (SC_i) and cloud used by the coordination service (CC_i). These credentials are stored in a file called *keystore*. The keystore is kept in persistent storage. RockFS splits the keystore in shares and stores them in separate places so that, even if an attacker accesses one of the shares, he can neither read nor delete the keystore. This is achieved using secret sharing (Shamir, 1979; Blakley, 1979).

When the user performs a login, the RockFS agent needs to combine some of the shares, e.g., 2 out of 3 shares, to obtain the credentials, which it keeps only in volatile memory. By default, it uses the share kept in the coordination service and the share in the client device. The share in the client device is protected by the user account access control mechanisms.

For recovery there is one or more additional shares stored in an external memory, like a USB flash drive, or a smart card, which must be kept at a secure location. The use of secret sharing is further detailed in Section 4.3.1.

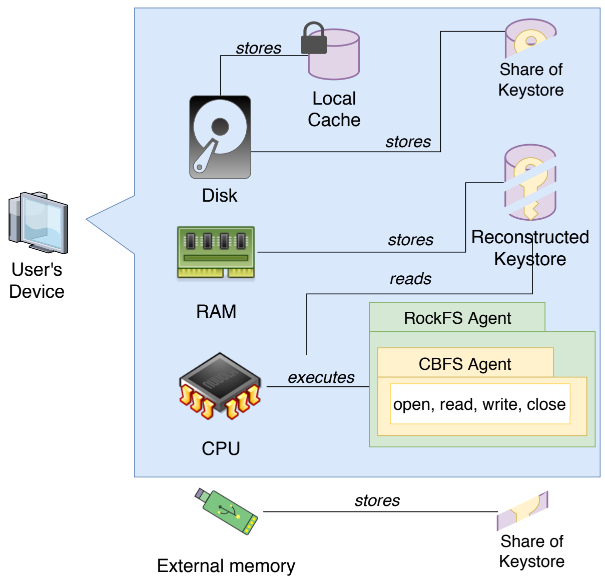

Figure 4.4: User device with client-side components and external memory with one of the shares of the keystore.

The user device architecture is represented in Figure 4.4. The Disk stores the encrypted local cache and one of the shares of the keystore. The RAM stores the keystore that is reconstructed with secret sharing. This ensures that even if the user's device is stolen, the adversary cannot obtain the access credentials from the disk. The CPU executes the RockFS agent.

4.1.5 Logging architecture

Figure 4.5 represents the architecture of the subsystem used to log file system operations in order to support recovery. The operations are logged in a coordination service (top-right of the figure). This coordination service will contain the metadata corresponding to the log entry, i.e., a timestamp, file identifier and user identifier. The log entry data is stored in the cloud storage. The figure also represents what happens when the user closes a file after writing into it: the file and log data are uploaded to the cloud storage services. Then, a log entry is created in the coordination service. Finally, in some cloud-backed file systems (e.g., SCFS), the file metadata will be updated in the coordination service. This step must be performed at the end to ensure that all data — the file, log data and metadata — is stable before notifying the user that the operation was completed successfully. This step is done at the end rather than at the beginning to prevent other users from seeing data files that were not yet uploaded to the cloud or may not be uploaded if the client crashes during the operation.

Each log entry has two parts: data and metadata. These parts are kept in different systems for two reasons: first, the data part requires more storage space, so it is more effectively stored in a cloud storage service; second, the metadata needs to be queried by the administrator, something that is supported by coordination services, but is harder to do with cloud storage services (typically the data would have to be downloaded before processing the query).

4.2 Storage Recovery

This section details how RockFS handles threat T1: an adversary illegally modifying files as if he was their owner (see Section 4.1.1). When an adversary hijacks a user's session, e.g., by stealing his computer, he can tamper with the user's files. Every action executed in the user's device that modifies a file is eventually synchronized to the storage cloud(s), so the effects of an attack that

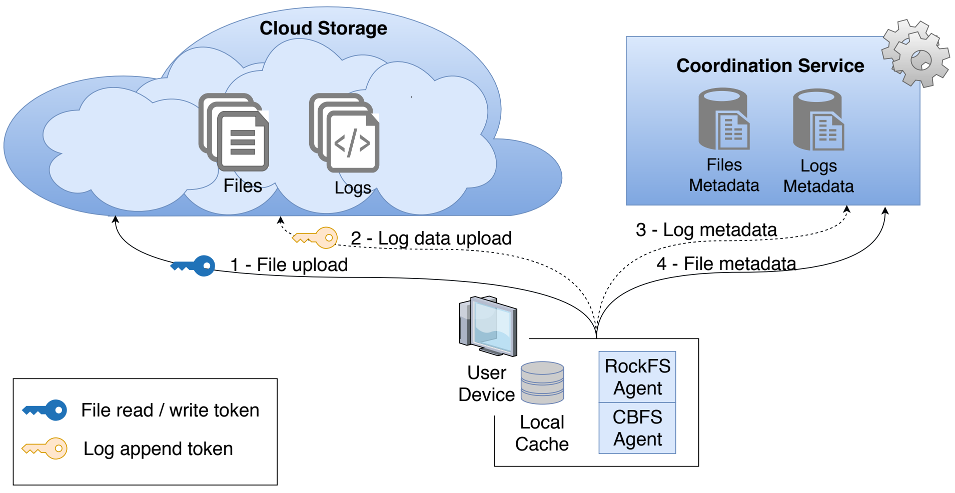

Figure 4.5: RockFS logging of operations in the file system.

occurred on the client-side are propagated to the cloud-side. When this happens, the cloud(s) will have tampered files. To cope with this vulnerability we propose a recovery approach that enables the administrator to identify malicious operations and undo them.

4.2.1 Recovery and logging threats

The log is the resource that allows the administrator to recover from attacks. As such, it becomes a target for an attacker that wants to prevent the administrator from recovering the file system. After modifying user files, an attacker may also try to modify the log entries corresponding to his actions in order to make the effects of the attack permanent. More exactly, in RockFS an adversary may try the following attacks:

- **A1:** attacker illegally modifies user files in the cloud storage services;
- **A2:** attacker illegally modifies log entries;
- **A3:** attacker illegally modifies both the user files and corresponding log entries;

If an attacker succeeds in **A1** it is still possible to revert what he has done by recovering the affected files, as explained further in Section 4.2.3. To cope with **A2** we propose a log integrity check mechanism explained in Section 4.2.2. In **A3**, the attacker is not able to modify the log given that RockFS only allows to append new log entries. Furthermore, any unauthorized log entry is detected by the integrity mechanisms and discarded before recovery.

4.2.2 Logging

To enable the administrator to undo the faulty operations performed by adversaries, all modifications done to a file have to be registered in a log. The RockFS agent is the component responsible for recording these operations. The log is stored alongside with the files in the cloud or cloud-of-clouds, with the protections provided by the cloud-backed file system (more precisely by its cloud-access subsystem, such as the one available in DepSky (Bessani et al., 2011)). The metadata that is logged in the coordination service for an operation, lm_{fu}, contains a timestamp, the user identifier, the file name, the version identifier and the operation (create, update or delete).

The log data of the user's files, ld_{fu}, is stored encrypted in the cloud or cloud-of-clouds and consists of the differences between the new version of the file and the previous one or, if the differences are larger than the file, the file itself.[3]

Logging is triggered when a file is closed. Specifically, when the POSIX operation close is invoked on a file f_u, the following operations are executed by the RockFS agent:

[2]From time to time it is necessary to create a snapshot of the file system in order to clean old log entries (for instance, by moving them to cold storage, e.g., Amazon Glacier (Robinson, 2008)) and save some storage space.

[3]More sophisticated policies might be devised but we simplify by considering that the best is to store whatever is the smallest between the two: differences or whole file. Nevertheless, we expect the differences file to be typically much smaller than the whole file, except for small files.

- Compute ld_{fu}, a file with the differences between f_u and its previous version that is stored; if ld_{fu} is larger than f_u, then consider ld_{fu} to be f_u (a flag indicates which is the case);
- ld_{fu} is encrypted using a random secret key S_{fu};
- If a cloud-of-clouds is being used, the encode function is used to split ld_{fu} in n shares, where n is the number of clouds used;
- ld_{fu} is sent to the cloud storage service or services (one share per service in the latter case) following the same scheme used in the cloud-backed file system;
- The key S_{fu} is stored in the same way the cloud-backed file system does, e.g., in the coordination service or by splitting it in shares using secret sharing and storing one per cloud (as in SCFS (Bessani et al., 2014));
- Both the file and log data are uploaded to the cloud storage back-end;
- Finally, log metadata lm_{fu} is stored in the coordination service after its corresponding data was uploaded to the storage back-end.

Both the log entries and files are uploaded to the cloud or cloud-of-clouds by the agent. If an adversary gains access to the user's device, he could try to erase or modify the log entries in order to prevent the administrator from recovering the file. To prevent this from happening, RockFS uses two distinct access credentials for the cloud storage: the user's files token, t_u, and the logging token, t_l. These tokens are generated by the cloud storage providers and they restrict access in a way that t_l cannot be used to modify the user files and t_u cannot be used to tamper with the log entries. Most cloud storage services provide such control access mechanisms (Amazon, 2015; Calder et al., 2011; Krishnan and Gonzalez, 2015). Figure 4.5 shows how RockFS agent uses these credentials.

By using the logging token, t_l, we ensure that even if an adversary gains access to the user's device he cannot tamper with the existing log entries. He can only create new log entries in order to try to interfere with the recovery process. To ensure the recovery is done correctly, RockFS has to provide *forward-secure stream integrity* (Ma and Tsudik, 2009), a property that ensures that logs, which are considered to be streams of sequential entries, can be verified for integrity. The scheme we leverage in RockFS is the forward-secure stream integrity scheme (FssAgg) presented in (Ma and Tsudik, 2009) that provides the following guarantees:

- *Forward Security:* the secret signing key used in the scheme is updated by a one-way function, making it computationally unfeasible for an attacker to recover previous keys from a current, stolen, key;
- *Stream Security:* the sequential aggregation process preserves the order of the log entries and provides stream-security;
- *Integrity:* illegal insertions, modifications and deletion of log entries are detected.

This scheme provides the following functions:

- FssAgg.Kg: generates pairs of asymmetric keys;
- FssAgg.Asig: generates an aggregate signature for a log entry. Takes as input the previous aggregate signature, the log entry and the current secret key.
- FssAgg.Upd: updates the secret key used to authenticate each log entry;
- FssAgg.Aver: verification algorithm that takes as input a log entry and the corresponding signature.

These protections are relevant for the security goals of RockFS and the system implements them. The forward-secure stream integrity scheme (Ma and Tsudik, 2009) allows the integrity of the entire log to be checked. This requires that, during setup, two random symmetric keys, A_1 and B_1, are securely exchanged between two parties. We assume the administrator himself is responsible for providing these keys to the cloud storage servers (stored in shares using secret sharing) and the coordination service. For every new log entry L_i that is created, RockFS agent will generate a hash value based on the previous ones:

$\mathcal{U}_i = H(\mathcal{U}_{i-1}|mac_{A_i}(L_i))$

where $\mathcal{U}_i$ is a FssAgg MAC of the i^{th} entry of the log. It is calculated by the hash of the concatenation of two values: the previous $\mathcal{U}_{i-1}$, and the MAC of the current log entry with the current symmetric key A_i. A_i evolves in each new log entry by using an hash function ($A_i = H(A_{i-1})$).

The new hash values are then uploaded to the cloud storage servers and the coordination service. A more detailed explanation can be found in (Ma and Tsudik, 2007).

4.2.3 Recovery

Before recovering a file, the RockFS agent needs to verify the integrity of the log entries using the forward-secure stream (Ma and Tsudik, 2007) verification algorithm. This is done by taking the symmetric key A_1 and computing the corresponding hash values (A'_i) for each log entry (L_i). Then RockFS compares the computed hash values of each entry ($A'_i = A_i$). If they match, then the log entries are valid and the recovery process can initiate. Otherwise, the invalid log entries are removed.

Recovering a file, as opposed to rolling it back to a previous version, allows users to erase illegal modifications while keeping the valid ones that occurred after the attack. To do so it is necessary to reconstruct the file by executing each valid action. This technique has been proposed in a different context and named as *selective re-execution* (Kim et al., 2010). It can be executed from the first operation that created the file or by obtaining the latest valid version of a file. From this point, RockFS will apply every valid operation to it until the present time. In more detail:

1. the administrator fetches the first version of f_u and its corresponding log entries, LD_{fu} (LD_{fu} is an array with several ld_{fu});
2. the function decode is used to join the shares of both the f_u and every ld_{fu}
3. the administrator selects the malicious modifications from LD_{fu} and discards them;

4. for each ld_{fu} in LD_{fu}, the function *patch* is invoked to apply ld_{fu} to f_u;

5. file f_u is shared by the encode version and sent to the cloud storage services.

Steps 2 and 4 are only necessary if the file system is backed by a cloud-of-clouds offering. If the file system is backed by a single cloud, then there is no need to share the log entries and the file.

In relation to step 3, notice that we assume that there is some way of knowing which modifications have been malicious. This is a problem of intrusion detection to which there are many solutions (Kruegel et al., 2005; Robertson et al., 2006), so we simply take that for granted.

It is worth mentioning that the recovery operations will be logged as well. This means that, if a file gets corrupted a second time after it was already recovered, then the RockFS will execute the same operations on the file that were executed in the first recovery. This happens because we want to ensure integrity of the log entries and by not allowing even the administrator to erase or modify log entries, we are also avoiding adversaries to do so.

4.3 Securing the User Device

In this section we propose solutions for the threats T2 (adversary corrupting access credentials) and T3 (adversary accessing files cached in the user device) presented in Section 4.1.1. The user device stores two types of sensitive information: the credentials for accessing the cloud(s), and the local cache with the most recently accessed files. By corrupting the access credentials an attacker may violate the availability of the service. If the local cache is accessed by an attacker both the integrity and confidentiality of the user files are at risk.

4.3.1 Securing user credentials

The credentials are protected with *secret sharing* (Shamir, 1979). The idea is that, even if the user's machine is compromised and the keystore is corrupted, the user may still recover his credentials and resume the use of the cloud-backed file system.

In a secret sharing scheme there is a special entity, the dealer, that splits a secret among n parties. Each party gets a share of the secret, meaning that if an attacker succeeds in obtaining one of the shares, he cannot reconstruct the secret. With such a scheme, $k < n$ shares of the secret are required to reconstruct it; therefore, an attacker would need to be get k shares to recover it. More specifically, in this work we use a secret sharing scheme called *publicly verifiable secret sharing scheme* (PVSS) that allows verifying if the shares are corrupted (Schoenmakers, 1999).

In RockFS, the PVSS scheme is used to break the keystore with the credentials in shares. The client acts as a dealer, sharing, combining and verifying the shares. PVSS provides the following functions:

- share, invoked by the client to obtain the shares of the keystore;

- combine, invoked by the client to reconstruct the keystore using k shares;

- verifyD, invoked by the servers to verify if the received share is legitimate;
- verifyS, invoked by the client to verify if the shares sent by the servers are legitimate.

The shares of the keystore are generated during *setup*, in the following way:

- the RockFS agent asks the user for how many shares of the keystore (n) should be generated, and how many are needed to reconstruct the keystore (k);
- the RockFS agent invokes the share function of the PVSS with parameters n and k;
- one of the shares is sent to the coordination service while the remaining shares are given to the user so he can choose where to store them,
- the shares given to the user are erased from disk and RAM, and the setup is complete.

By default, the user keeps one secret share on his device, for making it simple to log in RockFS (assuming a scenario with $n = 3$ and $k = 2$) and another share in the coordination service (so the $k = 2$ that are needed are available online). However, the PVSS allows the user to choose a different way to split the secret (different parameters n and k) and different devices where to store them, for added security. The user's smartphone can be used for this purpose, or other more elaborate password stores (Shirvanian et al., 2017).

A user recovers the keystore in two situations: every time he logs in, and when his device was compromised, and he needs to recover the keystore using the share kept in external memory. For both cases the recovery works as follows:

- the user provides $k - 1$ shares (the remaining one is located in the coordination service);
- RockFS agent fetches the remaining share from the coordination service;
- RockFS agent executes the PVSS function verifyS to verify if all the shares are legitimate;
- RockFS agent executes the PVSS function combine and loads the keystore into memory. In any case the keystore is on the disk.

This scheme protects the keystore from deletion or ransomware encryption, unlike the alternative of encrypting the keystore with a user-provided password (which would be converted to an encryption key). It also protects the keystore from being read at the disk, unlike the alternative of backing it up. In relation to the alternative of both encrypting and backing up the keystore, the solution still provides the benefit of not requiring the user to introduce a password.

4.3.2 Securing the device local cache

In RockFS we propose mechanisms to verify the integrity and ensure confidentiality of the local cache on the device, which stores the files recently accessed by the user. The existence of this cache in a cloud-backed file system is not mandatory, but in practice it is essential for the performance to be acceptable (otherwise, e.g., the client would block for long periods waiting for reads or writes in the storage clouds).

Figure 4.6 represents the extension of the POSIX file operations open and close that have to be modified for protecting the cache.

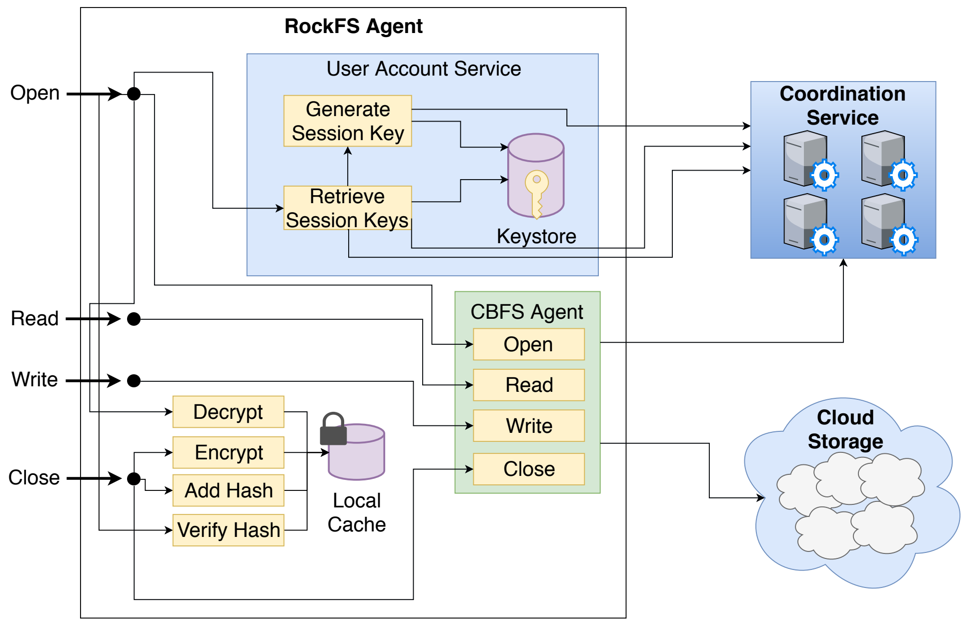

Figure 4.6: RockFS operations, the `read` and `write` operations from the cloud-backed file system remain unaltered. The operations `open` and `close` decrypt and encrypt the local cached files to ensure data confidentiality.

Confidentiality

Local cache files are encrypted using a session key, S_U, to ensure confidentiality. This key has a short validity (configurable by the administrator) and is associated with an entry in the coordination service, preventing an attacker from reusing old session keys. On open, the RockFS agent:

- Checks if S_U is still valid, if not the local cached file is discarded and a new S_U is generated and exchanged with the coordination service;
- Decrypts the opened file and loads it.

The file close operation:

- Creates a log entry for the file update, and uploads the log entry to the cloud;
- Uploads the file to the cloud;
- Fetches S_U from the keystore;
- Removes the log entry from the disk or memory;
- Encrypts the file that was closed.

Besides encrypting the locally cached file, the close operation also logs the modifications, as detailed in Section 4.2.

Integrity

The integrity of the local cache is ensured using cryptographic hash functions, which are encrypted with S_U together with the rest of the file. When a user invokes the open operation on a file f_u, the RockFS agent:

- fetches h_{fu}, the hash value correspondent to f_u;
- computes a hash value h'_{fu} of f_u;
- compares both hash values, h_{fu} and h'_{fu}. If they match the file is opened, otherwise the file is discarded, and a valid version of the file is fetched from the cloud.

When the user invokes the close operation on a file f_u:

- a new hash value, h_{fu}, is calculated and stored in the local cache alongside f_u;
- the file f_u and h_{fu} are encrypted.

4.4 RockFS Prototype Implementation

In our implementation of RockFS we used a modified version of SCFS as cloud-backed file system (Bessani et al., 2014). Most modifications were made to implement the log of operations and extend the compatibility of SCFS with more cloud storage back-ends by using the JClouds[4] library (Apache, 2018). SCFS works with a single cloud or with a cloud-of-clouds, but we considered only the latter configuration, as it provides stronger security and dependability assurances for the server-side (i.e., the cloud). The underlying protocol, DepSky, deals with the heterogeneity of the different clouds and is compatible with most commercial cloud storage. As coordination service we used DepSpace, also supported by SCFS. We start by presenting these components before presenting the implementation of RockFS itself.

4.4.1 DepSky cloud-of-clouds storage

DepSky is a dependable and secure cloud-of-clouds storage system that addresses four limitations of existing cloud storage services: loss of availability, loss and corruption of data, loss of privacy, and vendor lock-in (Bessani et al., 2011). It works by using a collection of cloud storage services that are managed remotely by a client library. The user files are stored encrypted in public clouds. It also uses erasure-codes (Cachin and Tessaro, 2006; Goodson et al., 2004; Hendricks et al., 2007; Halalai et al., 2017) to reduce the required storage for the user files. This way, instead of having n copies of a file distributed in n cloud providers, only a fraction of n (usually half) is needed. DepSky provides a consistency level similar to the weakest consistency level given by the cloud providers. For each user file in the clouds, there is the file itself and a signed metadata file that stores the version number of the file and verification data.

[4]JClouds is a Java library that provides a abstraction layer for different cloud providers.

DepSky supports two protocols: A and CA. *A*, which stands for availability, and *CA*, which stands for confidentiality plus availability. The *A* protocol replicates the file in each cloud storage. This improves availability at the cost of the extra storage (n times the size of the file). The *CA* protocol encrypts each file with a symmetric key before sending it to the cloud. Then the key is split in n parts using secret sharing, requiring at least $k+1$ parts to reconstruct the key. The file itself is also split in n shares using erasure-codes, which will require $k+1$ shares to reconstruct it. The *CA* protocol requires less storage thanks to the erasure-codes ($\frac{n-k}{n}$ as opposed to n).

RockFS uses the CA protocol for two reasons: first, it requires less storage space, and, second, it encrypts the files and secures the encryption key with secret sharing, which fits our system model.

4.4.2 SCFS cloud-of-clouds file system

SCFS (Bessani et al., 2014) is a distributed file system supported by DepSky, when configured for cloud-of-clouds (the case we consider). It provides consistency-on-close semantics (Howard et al., 1988). Its architecture is more complex than DepSky because it requires a coordination service to keep the logical structure of the file system and to coordinate concurrent accesses from multiple writers.

SCFS provides confidentiality of data stored in cloud storage providers through the mechanisms of DepSky: the user's files are encrypted and the secret keys are protected using secret sharing. These encryption mechanisms ensure confidentiality of data in the cloud. However, the data stored in the user's personal device is not encrypted since it is assumed to be trusted. Metadata of SCFS is stored in the coordination service for three reasons: the coordination service offers consistent storage; it uses replication to cope with faults; and because it can be used to cope with faults and to implement synchronization operations, such as file locking.

4.4.3 DepSpace dependable coordination service

DepSpace (Bessani et al., 2008) is a Byzantine fault-tolerant distributed coordination service. It was designed to run on a cluster of commodity machines, making it possible to deploy in any infrastructure cloud service. Unlike other coordination services like ZooKeeper (Hunt et al., 2010), DepSpace tolerates arbitrary faults. It uses Byzantine fault-tolerant protocols (Avizienis et al., 2004) to ensure correctness in the event of up to f arbitrary faults (also designated Byzantine faults), requiring $3f+1$ replicas, possibly running in different cloud services, for that effect. DepSpace provides a *tuple space* which can be used to implement locks, timed tuples, partial barriers and secret storage.

One of the limitations of the original DepSpace was the fact that it maintained its state in volatile memory, making it impossible to recover if more than f replicas failed by crashing. An enhanced version was proposed in (Bessani et al., 2013) and is used by RockFS. It applies check-pointing and logging mechanisms that use external storage to keep recovery data. These mechanisms also allow the migration of a DepSpace cluster to a different cloud provider, thereby avoiding vendor lock-in.

4.4.4 RockFS implementation

RockFS was implemented in Java SE 8. We chose this programming language since all the components of SCFS were written in Java and every cloud storage service used in this work also provides Java APIs. Also, since Java is a multi-platform language, it is possible to deploy in distinct execution environments, making it more robust against operating system specific vulnerabilities.

The size of the keys as well as the algorithms used to generate them were chosen taking into account their security in the long term. According to ENISA, the recommended values are: SHA-512 for hash values, 512-bit elliptic curves for public keys and AES-256 for symmetric keys (European Union Agency for Network and Information Security, 2014).

The keystore (depicted in Figure 4.6) is a text file with the access credentials for cloud services, the access credentials for the coordination service, the sessions keys (S_U) and the private key of the user (PR_U). This file is never stored on persistent storage (disk). It is split in, at least, three shares: with one of the shares in the coordination service, another in the user's device and an extra one in an external storage (USB memory or smartcard), especially for recovery. To recover the keystore it is not enough to reveal the secrets since this file is also encrypted (with AES-256), requiring a user password to decrypt it. Once it is decrypted it is loaded into memory (RAM).

The RockFS agent uses a variation of the UNIX *diff* command (Hunt and Szymanski, 1977) called JBDiff [5]. This command is used to calculate the log entries of each file operation. Each log entry is in fact the difference between the old version and the new version of a file. Recovery is done by reconstructing a file, i.e., executing the corresponding *patch* command sequence.

4.5 Experimental Evaluation

The performance of the protections for the credentials (threat T2) and cache (threat T3) was found to be below tens of milliseconds, so their cost can be considered negligible in the overall cloud-of-clouds solution. Therefore, our evaluation focused on T1, i.e., on the costs of the *recovery* scheme presented in Section 4.2. With the experiments performed we wanted to answer the following three questions: (a) What is the cost, in terms of performance, of having the RockFS agent log every file operation? (b) What is the cost, in terms of storage, of saving every file modification? (c) How long does it take to recover files depending of the number of modifications they suffered? The answer to this last question is relevant to assess how effective our solution is against *ransomware* attacks.

In the experiments we wanted to simulate a realistic scenario in which RockFS would be deployed in at least two different clouds. This way it would be possible to ensure that metadata and data are stored in different locations (logically and geographically) ensuring that even if one cloud gets compromised, it is not possible to read the users' files. We set up RockFS using two different clouds: Amazon S3 (Amazon, 2015) for the cloud storage services; and Google Compute Engine (Krishnan and Gonzalez, 2015), for the coordination service. Regarding the Amazon S3 storage services, we set up

[5] Java Binary Diff https://github.com/jdesbonnet/jbdiff

4 storage buckets in the Ireland data centers. In the Google Compute Engine we created 4 instances (for the 4 replicas of DepSpace) with 1 vCPU and 3.75GB of memory for each machine. All 4 replicas were located in the Belgium data center.

For the client machine, we created an instance, again with 1 vCPU and 3.75GB of memory, in the London data center. This additional instance serves as a client of RockFS and will execute the RockFS agent code. We chose to execute the client on the cloud for two reasons: first, it provides a stable Internet connection for the experiments; and second, this machine is as simple as possible, meaning that no extra software running in the background interferes with the execution of the experiments.

4.5.1 Latency overhead of log operations

To calculate the latency of logging operations we created a workload that consisted in creating files and then updating these files with an extra 30% content. We vary the size of the files between 1 and 50MB, according to statistics from (Agrawal et al., 2007). Given that SCFS offers two different approaches for file synchronization (blocking and non-blocking), we performed the experiments with both configurations. Each test was repeated 10 times and the values presented in the graphs correspond to the *average* values.

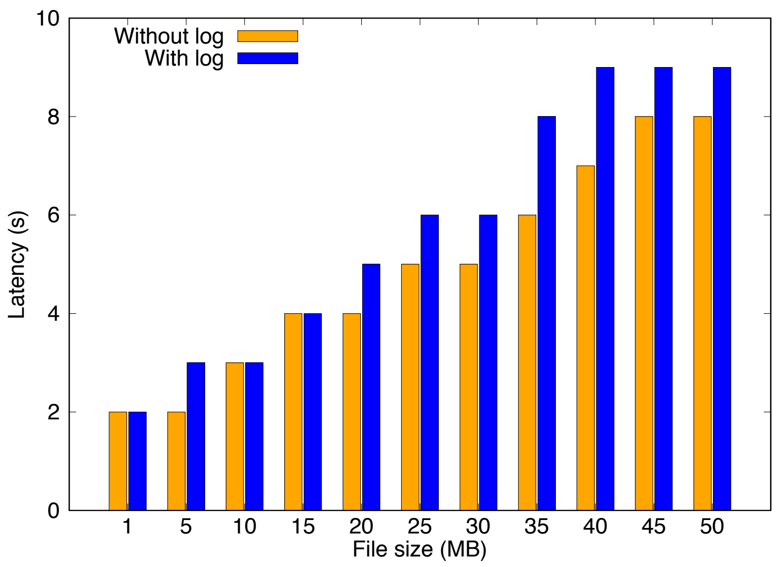

Figure 4.7: Latency of using RockFS with and without the log.

Figure 4.7 presents the average latency of logging file operations in RockFS. The latency is the time it takes since the user finishes updating the file, i.e., invoking the POSIX close function, and the time the coordination service finishes recording the file operation. The latency values without log were collected executing the workload on SCFS. The latency values with logging are, on average, 20% higher than the ones without logging. This overhead is expected, since it takes time for the RockFS agent to compute the log entry (differences between versions) and to upload these differences to the cloud. Several optimizations were performed to reach this value. The two most important ones were (1) both the logging and the file operation are processed in parallel by the coordination service, and (2) the file and log entry uploads are also done in parallel. This 20% overhead can be reduced by improving the network bandwidth (for instance, by using the same data center for the storage services and the coordination services) and by improving the computing hardware of the client (to process the differences quicker).

In a different experiment we used file system microbenchmarks from FileBench (Filebench, 2017) to execute different workloads. Table 4.2 shows the results of three different workloads: *sequential write*, that appends data to the end of a file; *random write*, that modifies a random section of a file; and *create files*, that creates new files without modifying them afterwards. Each workload was tested in both SCFS and RockFS in two different modes: non-blocking (NB) which synchronizes files to the cloud in the background allowing the user to proceed his work, and blocking (B) which blocks the application until the modified file has been completely uploaded to the clouds. We tested these three workloads with both SCFS and RockFS to understand how much it costs, in terms of performance, to log file operations. Unlike the original experiments in SCFS (Bessani et al., 2014), that executed several workloads of read operations, in this case we are only interested in testing operations that modify files, since these are the only ones being logged by RockFS. The overhead of using RockFS according to the results shown in Table 4.2 is non-negligible but can be considered acceptable, especially in the non-blocking mode which is the recommended configuration.

4.5.2 Storage overhead of log operations

We did experiments to find out how much more storage RockFS needs to keep all the logs of several file operations. To do so we executed 1, 10 and 100 file updates in files with sizes again varying from 1MB to 50MB.

Experiments show that the storage overhead of the log is significant. Every time a user appends 10MB to a file, an extra 10MB are added to the log. In this system model we are using the *CA* protocol of DepSky (described in Section 4.4.1) which uses erasure-codes to reduce the required storage to 2 times (as opposed to 4 times in the *A* protocol) i.e. a log entry of 10MB will occupy 20MB overall in the clouds.

It is also worth noting that a file that is modified several times will create a log history greater than a file that is created once and subsequently is not modified. In these experiments we wanted to evaluate how much storage does it take to store the log entries of files that are rarely updated (1 version), moderately updated (10 times) and intensively updated (100 times). Each modification to the file was in 30% of the original size of the file (e.g. a file with 10MB was updated with more 3MB every time).

Figure 4.8 presents the storage overhead of logging different files with different versions. The

Micro-benchmarks	# Operations	File size	SCFS		RockFS		Overhead	
			NB	B	NB	B	NB	B
write	1	4MB	1.63	1.71	1.90	2.12	17%	24%
create	200	16KB	197.60	236.76	219.00	298.20	11%	26%

Table 4.2: Latency (in seconds) of Filebench micro-benchmarks for SCFS and RockFS.

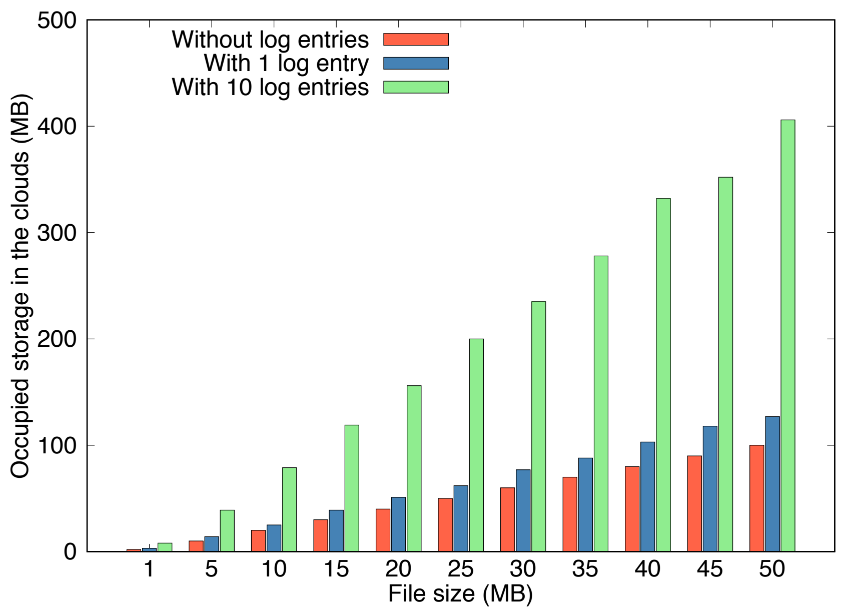

Figure 4.8: Required storage for the files and logs in the cloud storage services.

storage growth is linear. The red bars, labeled as "without log entries", represents the total storage occupied the file itself in the clouds. It is worth noticing that each file occupies twice its size in the clouds storage due to the use of several clouds and erasure codes. The increment in the total size for the blue bars (with 1 log entry) is only marginally superior to the red bars because it represents the size of the file plus the log entries, which only contains the delta of the modifications. For the 10 versions we can see that the required storage for the log is greater than the file itself. This motivates the adoption of a future *snapshot* mechanism to create backup versions of the files in order to discard log entries. The log size values for the 100 versions file are not in the chart. The sizes vary from 60MB (for the 1MB file) to around 3GB (for the 50MB file). It is worth noticing that existing cloud-backed file systems already employ a multi version approach to prevent data loss. In such systems the amount of storage required to save every version of the file would be greater than the RockFS logs, since it just stores the delta encoding of each version.

It is possible to calculate and approximation of the total required storage s for a file in version n with a percentage of modifications δ by computing the following equation:

$$s_n = 2 \times (s_{n-1} + \delta \times s_{n-1}) \tag{4.1}$$

The storage overhead is considerable, but it is not detrimental of the adoption of RockFS for two reasons. First, cloud users and providers seem not to be too eager to minimize the storage space used as, for example, Microsoft OneDrive by default keeps in storage 500 versions of each file. Second, most cloud providers offer cheaper cold storage to which older versions can be moved, e.g., Amazon Glacier. Compression techniques could also be used to reduce the overall storage required by RockFS. This is a problem we intend to explore as future work.

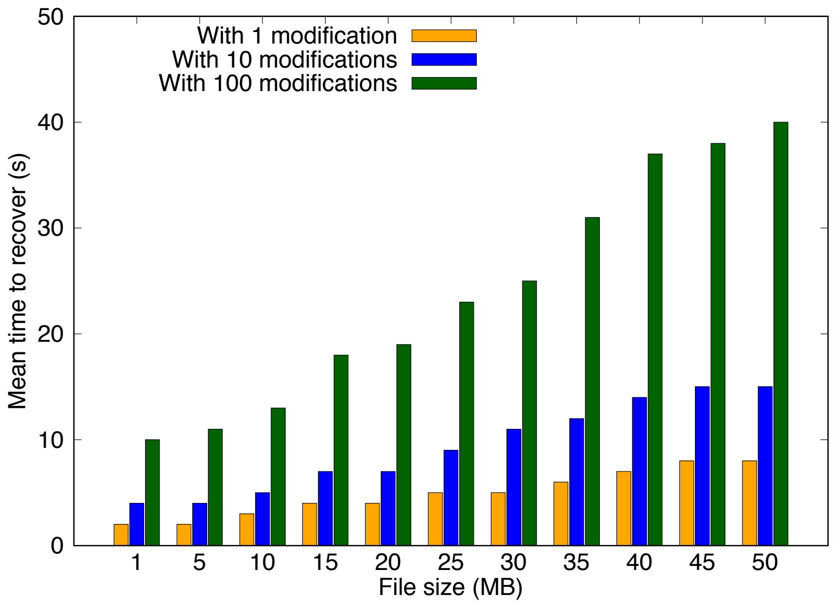

Figure 4.9: Mean time to recover files with 1, 10 and 100 versions.

4.5.3 Mean time to recover files

The MTTR (Mean Time to Recover) a specific file varies according to the number of versions of that file. A file that was only modified once before being attacked can be recovered by executing a *patch* operation (i.e. applying the differences in the log to the original version), while a file that was modified 100 times requires 100 *patch* operations to be executed.

Here we are recovering the file system from a *ransomware* attack. In this type of attack, every file in the file system is corrupted (encrypted). First, we did experiments to measure the recovery of a single file. Then we did experiments with the recovery of the set of all files.

To evaluate the MTTR of different files we took the files and log entries from the previous experiments and recovered each file 10 times (to reach an average value). Again, a file with 1 version means that was created and modified just once while a file with 100 versions means that after its creation, it was modified 100 times.

Figure 4.9 presents the MTTR of several files with different versions. Although the MTTR grows linearly with the file size, in the 100 versions files the growth is steeper. The time varies from around 2 seconds (for 1 version file with 1 MB) to around 40 seconds (for the 100 versions file with 50MB). We optimize the recovery process by downloading every log entry of the file to be recovered at once, instead of downloading each entry at a time.

To evaluate how RockFS recovers a complete file system from such attack, we created 16KB files (from 10 to 10,000) and modified them several times (from 1 to 100 versions with each modification being a 4KB write in the file). Then RockFS recovered every file of the file system. The results are presented in Figure 4.10. The mean time to recover grows exponentially with the number of files in the file system. In the worst case of the experiments (10,000 files with 100 versions each) it took around 2 hours and 5 minutes to recover every file in the system. Considering the hindering effects

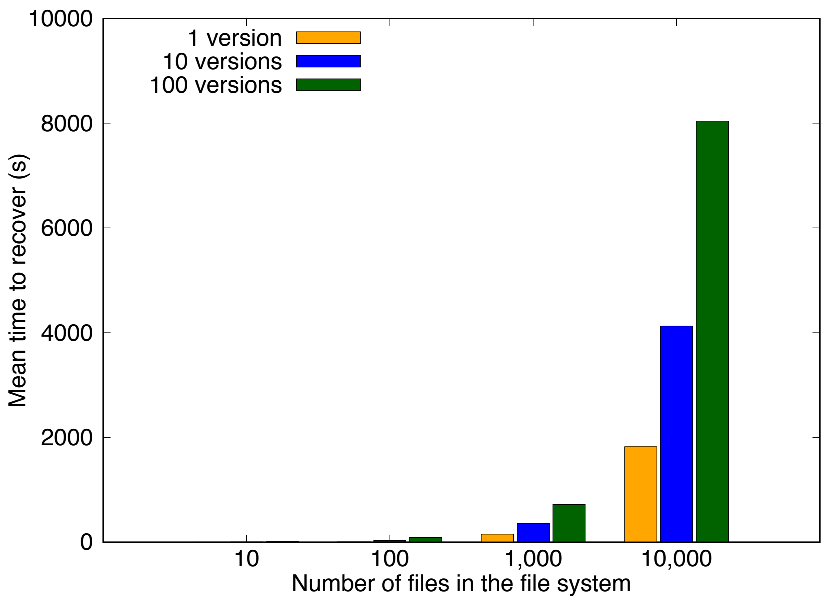

Figure 4.10: Mean time to recover a file system compromised by a ransomware attack varying in the number of files and versions of each file.

of a ransomware attack, the full recovery time is acceptable. This is still a considerable time but once RockFS starts the recovery process, files become gradually available for the user. Because of this property of the system, the recovery can start with the most urgently needed files, as specified by the user. And assuming that a regular user does not access every file in the system at once, this allows him to resume working while RockFS continues the remaining recovery process in the background.

4.5.4 Network overhead

There is a network overhead every time a user modifies a file and when an administrator recovers a file. It is possible to calculate how much network traffic will be spend by computing a model for the upload and a model for the download. Moreover, this model allows the users of RockFS to calculate how much more will it cost, in terms of monetary costs, to use RockFS as opposed to a single cloud storage without recovery capabilities.

In the following models we assume a network overhead of σ that is depends on the size t of the file, the delta encoding percentage of the file δ, the number of clouds n and the number of versions of the file v.

Network overhead while logging

Every time a user updates a file it is automatically uploaded to the cloud (or clouds). RockFS will also append the delta encoding of the modifications to be stored as the log entry of that file operation. Given that a cloud-of-clouds configuration requires data to be sent to multiple clouds, such cost must be multiplied by the number of clouds. In the end we can divide the total cost by 2 given that we use erasure codes to fragment data. The following equation can be used to calculate the total cost of using RockFS to log every file operation.

$$\sigma = \frac{((t + \delta \times t) \times n)}{2} \tag{4.2}$$

For example, by computing this model with a fixed $\delta = 30\%$ and $n = 4$, uploading a 1MB file will result in 3MB traffic while a 50MB file will result in 130MB.

In terms of monetary costs, most cloud storage services do not charge for uploading data. Because of this, network-wise logging every file operation will not have an impact in the service bill.

Network overhead while recovering

When the administrator recovers a file, RockFS will download the first version of that file and every log entry. The following model can be used to compute the total network traffic required for recovering a file.

$$\sigma = \frac{((t + \delta \times t \times v) \times n)}{2} \tag{4.3}$$

For example, by computing this model with a fixed $\delta = 30\%$ and $n = 4$, recovering a 1MB file with only 1 version will result in 3MB traffic while a 50MB file with 100 versions will result in 3.1GB. As of April 2018, Amazon S3 charges about 9 cents of a US dollar for GB downloaded from their cloud. Other cloud storage services charge a similar price for their services. This means that the 50MB file with 100 versions will cost the user around 27 cents of US dollars to recover, while the 1MB file with only 1 version would cost less than a cent.

4.6 Summary

This chapter presents RockFS, a recoverable cloud-of-clouds file system resilient to client-side attacks. RockFS provides recovery mechanisms for the access credentials of the users and for files stored in the cloud storage services. RockFS improves on SCFS and other cloud-backed file systems by protecting against client-side attacks and allowing for recovery of unauthorized changes, in particular, recovery from ransomware attacks. The experimental evaluation results show that it is possible to recover intensively modified files (with 100 updates) in around 40 seconds. Using RockFS to log file system operations imposes a performance overhead in the order of 20%, a cost that can be further reduced by improving the computing characteristics of the cloud servers used by the file system.

5 NoSQL Undo: Recovering NoSQL Databases by Undoing Operations

This chapter presents NoSQL Undo, a recovery approach and tool that allows database administrators to automatically remove (“undo”) the effects of faulty operations in NoSQL databases. NoSQL Undo is a client-side tool in the sense that it does not need to be installed in the database server, but runs similarly to other clients. Unlike recovery tools in the literature (Brown and Patterson, 2003; Chiueh and Pilania, 2005; Kim et al., 2010), NoSQL Undo does not require an extra server to act as proxy since it uses the built-in log and snapshots of the database to perform recovery. It also does not require extra meta-data or modifications to the database distribution or to the application using the database. The tool offers two different methods to recover a database: a *full recovery* that performs better when removing a large amount of incorrect operations; and a *focused recovery* that requires less database writes when there are just a few incorrect operations to undo. NoSQL Undo supports the *replicated* (primary-secondary) and *sharded* architecture of NoSQL databases. The tool provides a graphical user interface so that a database administrator is able to easily and quickly find faulty operations and perform a recovery.

Currently there are many NoSQL databases.[1] Some of the best known are: Cassandra (Lakshman and Malik, 2010), MongoDB (MongoDB, 2018), Hadoop HBase (White, 2009), Couchbase (Brown, 2012), DynamoDB (Sivasubramanian, 2012), and Google BigTable (Chang et al., 2008). These databases vary mainly in the format of stored data, which can be key-value (Sivasubramanian, 2012), columnar (Vora, 2011; Lakshman and Malik, 2010; Chang et al., 2008), or document oriented (MongoDB, 2018). In terms of scalability, all these systems can be deployed in large clusters and have the ability to easily extend to new machines and to cope with failures.

[1]When there is no ambiguity we sometimes abuse the nomenclature and use “databases” for short to database management systems.

To evaluate NoSQL Undo, we integrated it with MongoDB and conducted several experiments using YCSB (Cooper et al., 2010). The latter is a benchmark framework for data-servicing services that provides realistic workloads that represent real-world applications. It allows configuring the amount of operations, records, threads and clients using the database. With the experimental evaluation we wanted to compare both approaches to undo incorrect operations (focused recovery and full recovery), and how both methods perform with different sets of operations. The experimental results show that it is possible to undo a single operation in a log with 1,000,000 entries in around one second and to undo 10,000 incorrect operations in less than 200 seconds.

Figure 5.1 revisits the cloud architecture diagram presented in Chapter 2. In the figure we can see that NoSQL Undo operates at the PaaS level, more specifically the Database Access and Replication Middleware. This means that NoSQL Undo is capable of recovering stand alone database instances that are used by different sorts of applications as well as web applications deployed in the PaaS and applications offered to the user as SaaS.

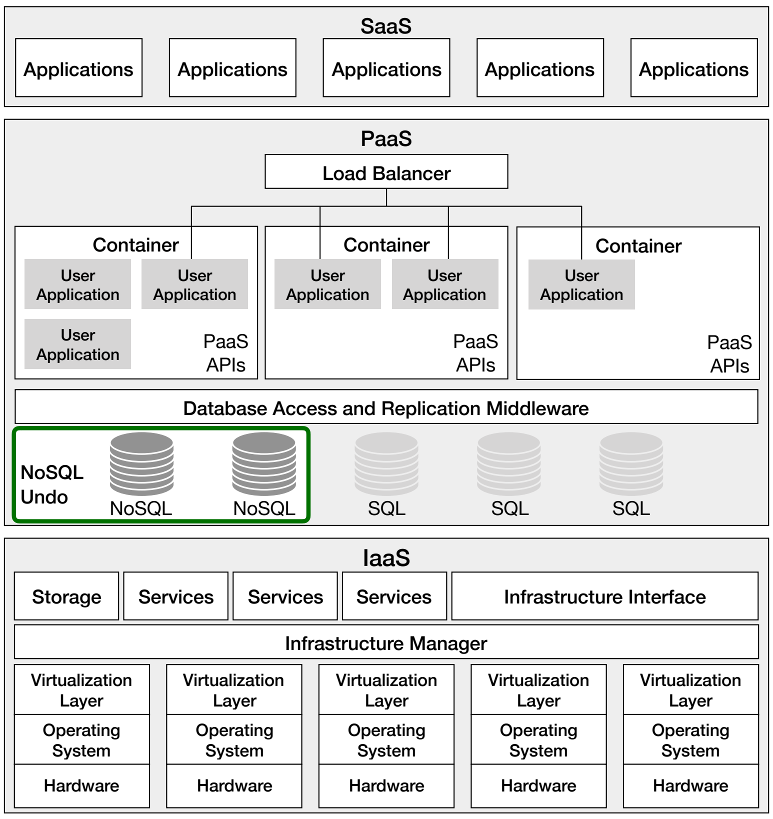

Figure 5.1: NoSQL Undo in the common cloud architecture.

5.1 NoSQL Databases

Most NoSQL databases aim to provide high performance for large-scale applications (Cattell, 2011; Li and Manoharan, 2013). Their ability to split records and scale-out horizontally allows them to maintain performance when dealing with high traffic loads and peaks. In comparison with traditional rela-

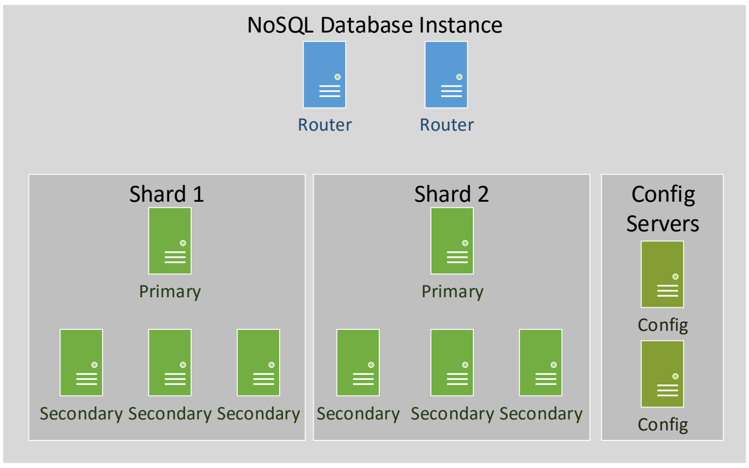

Figure 5.2: Example of NoSQL instance with two shards.

tional databases, NoSQL databases offer better performance and availability, over strong consistency, relational data and ACID properties (Cattell, 2011). These characteristics make NoSQL databases a good choice for applications with high availability and scalability requirements, but no need of strong consistency and complex transactions.

There are different NoSQL databases, but there are many common elements in their architectures. In some databases several elements can be incorporated in the same server (Brown, 2012), whereas others require that each component of the system is placed in a dedicated server (Lakshman and Malik, 2010; MongoDB, 2018). Some databases (Lakshman and Malik, 2010; Vora, 2011) may require additional components, such as Zookeeper (Hunt et al., 2010) to manage group membership. Despite their differences, NoSQL databases that provide replication and horizontal scaling tend to have a similar architecture.

5.1.1 Architecture

Figure 5.2 represents the architecture of a common NoSQL database instance with two *shards*. In this configuration each piece of data is divided into slices that are stored in the shards. Each shard is a collection of servers that store the same data, i.e., that are *replicas*. This redundancy provides fault tolerance (no data is lost in case a replica fails) and more performance (any of the servers can respond to read operations). In each replica a server acts as *primary* and coordinates the replication actions of the remaining, *secondary*, servers. The primary server is responsible for keeping data consistent inside a shard.

In order to correctly split data in shards, a special server (or a collection of servers, depending on the complexity of the instance) is responsible for redirecting the requests to the correct shards and divide the records. These servers are usually called *routers*. The routers are the components of the system that interact with the application. If there are no routers alive then the database is inaccessible. To prevent this usually there are several routers. Besides increasing availability of the database, having several routers also increases the performance since it reduces the bottleneck of having a single server responding to every request of the application.

Some NoSQL databases have extra servers that are responsible for recording configuration information of the instance (*configuration servers* in the figure). This allows separation of concerns: while some server are only responsible for storing data, other are responsible for storing metadata and configuration parameters of the instance.

This architecture is very similar to a distributed MongoDB instance. The main difference is that the configuration servers in a Mongo database are grouped in a replica set (primary with a set of secondary servers) which means that all the configuration servers store the exact same information. Cassandra also has a similar architecture except that the configuration servers are not present. The metadata and configuration parameters are stored in the data servers. HBase has a slightly different architecture. Master servers are in a group and manage how data is partitioned to the slave servers (also called region servers). This approach does not separate the servers by data partitions, but instead separates the servers in two groups: master servers and region servers. In Couchbase it is possible to aggregate every component in every server. This way every server performs the same tasks. It is also possible to deploy a Couchbase database having each server responsible for a specific task. This way the architecture of a Couchbase database would be like the one in Figure 5.2.

5.1.2 Logging mechanisms

Most databases have a logging mechanism that records database requests and take periodic snapshots, allowing the recovery of the system in case of failure. NoSQL databases also have logging mechanisms and take snapshots to allow the recovery of an individual server. However, these logs are specific to an individual server, so if the entire database (all servers) fails it is difficult to recover it using the local logs of each server. Besides the local log of each server, most NoSQL databases also have a global log that is used to maintain consistency across all the servers. This global log is not intended to recover the database, but instead to guarantee that all the servers receive the same set of operation in the correct order.

Local logs

Any cluster should be prepared for single servers failing unexpectedly. After a failure, a server has to perform fail-over, i.e., to take the required actions to come back to work without interfering with the remaining servers. In order to do that the server must have a diary in which it records every operation it writes to disk, so in case of failure the server only needs to repeat every operation and it should reach the state right before the failure. In some cases this diary is not sufficient since it only contains recent operations, so it is necessary to use a snapshot (a full copy of the server in a previous moment in time) of the server and complete the missing information with the entries in the diary. The diary in which the server stores the operations is a local log and the information it contains is usually non-human readable and specific to the server itself. A simple query can be decomposed in several local log entries corresponding to all the disk writes necessary to execute that query.

Global logs

In order to keep data consistent across servers the requests have to be delivered in total order, i.e., all requests delivered in the same order to all servers. On the contrary, simply sending the requests to all the servers might not work since some messages could be lost in the network or delivered out of order. To prevent this most databases use a global log in which they store every request they receive. This log is then used to propagate the operations to all the servers. If a server fails to receive an operation it can later consult the global log and execute it. Some databases have a fixed storage limit for this global log, i.e., it is implemented as a circular array (older operations are overwritten by new ones to prevent the log from growing indefinitely). The way the operations are stored in the global log is usually similar to the way data is stored in the database, meaning that is is possible and fairly easy to perform normal queries over the global log. Random values are converted into deterministic values to guarantee that every operation in the log is idempotent. Besides the operation itself, each log entry contains also a numeric value that allows ordering the executed operations. This numeric value guarantees that every server in the instance is able to reach the same state, since they all execute the operations in the same order.

In most NoSQL databases the global log has a format that is close to the format of the requests, contains the executed queries, is in a human readable form, and stores the operations in the order they were executed. Therefore, in NoSQL Undo we use this log to perform recovery.

DBMS	**Data Record Unit**	**API**	**Operation Log**
Hadoop / HBase (Vora, 2011)	Column families	Java	Yes
Cassandra (Lakshman and Malik, 2010)	Column families	Java	Yes
Hypertable (Khetrapal and Ganesh, 2006)	Column families	Java and others	Yes
MongoDB (MongoDB, 2018)	Document	C+, Java	Yes
Elastic (Kononenko et al., 2014)	Document	REST API	Yes
CouchDB (Brown, 2012)	Document	REST / JSON	Yes
DynamoDB (Sivasubramanian, 2012)	Key Value / Tuple Store	REST / JSON	Yes
Berkeley DB (Olson et al., 1999)	Key Value / Tuple Store	C, C++ and Java	Yes
Voldemort (Feinberg, 2011)	Key Value / Tuple Store	Java	Yes

Table 5.1: Comparison between different NoSQL databases.

Table 5.1 lists some well known NoSQL databases. In the table the NoSQL databases offer different structures to store data: column families, document and Key Value. All of them offer a Java API or an API that is compatible with Java, such as REST web services in JSON. The Master / Slave replication approach is present in every database. Although, some of them have a different name for the replication schema, such as Primary / Secondary or Coordinater / Worker. For simplification purposes we use the Master / Slave terminology. Replication is achieved by the use of Operation Logs, which can be used for intrusion recovery.

5.2 NoSQL Undo

NoSQL Undo is a client side tool that only accesses a NoSQL database instance when the database administrator wants to remove the effect of some operations from the database, e.g., because they are malicious. The client does not need to be connected to the server in runtime since it uses the database built-in logs to do recovery.

5.2.1 Undo vs rollback

Every database provides rollback capabilities, meaning that if an incorrect operation is detected and needs to be removed then it is possible to revert the entire database to a previous point in time prior to the execution of that incorrect operation. The problem with this approach is that every single correct operation executed after the point in time to recover is lost. Figure 5.3 represents the state of a database with three different documents (D1, D2 and D3). All three documents were updated 6 times, i.e., there are 6 versions of each of these documents in the log of the database. Two of these documents (D1 and D2) were corrupted by malicious operations (red dots). D1 is later updated with valid operations meaning that part of the document may be corrupted while the other part is valid. To use a traditional rollback, the administrator needs to select a point in time prior to any malicious operation, which in this case is when all documents were in version 2. After the rollback the database is clean, however all the documents were reverted to older versions. Every correct operation executed after version 2 is then lost. By using any of the algorithms of NoSQL Undo, the administrator is able to clean both D1 and D2 and still keep every correct operation that was executed after, since both algorithms correct the corrupted documents by removing the effects of the malicious operations, instead of replacing them with older versions.

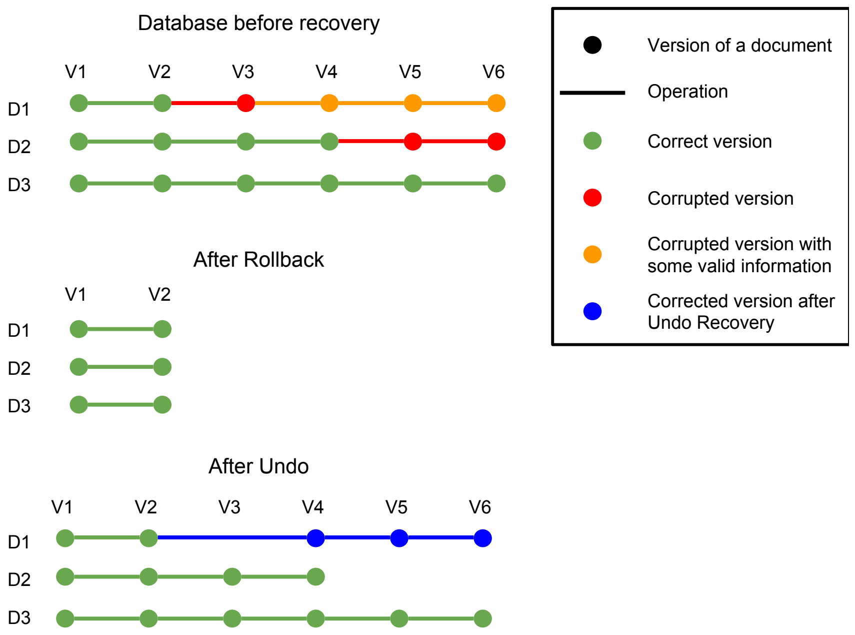

Figure 5.3: Comparison of using rollback to recover a corrupted database with using undo to revert incorrect operations.

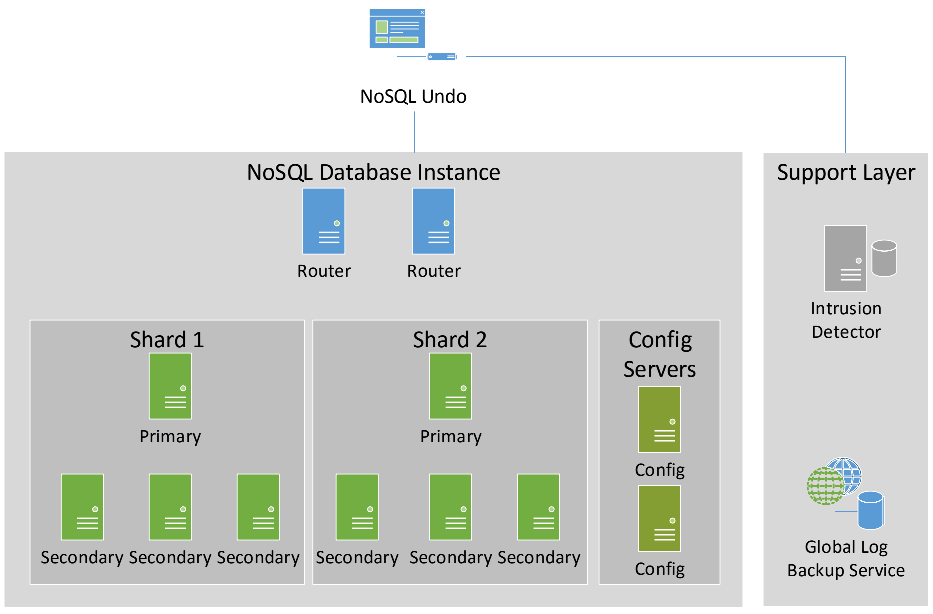

Figure 5.4: A NoSQL database with NoSQL Undo.

5.2.2 Architecture

Figure 5.4 presents an example of a NoSQL instance with NoSQL Undo. The architecture is logically divided in two layers: the database layer in which the database runs without any modification to the configuration or to the software; and the support layer, which includes optional modules that can also be deployed to improve the capabilities of NoSQL Undo.

In order to undo undesired operations, there has to be a log with every executed operation and snapshots with previous versions of the database. Most recovery systems, such as Operator Undo (Brown and Patterson, 2003) and Shuttle (Nascimento and Correia, 2015), implement a proxy that intercepts every request to the database and saves these requests in a specific log, independent from the DBMS. That approach may impact performance since every operation must pass through a single server (Nascimento and Correia, 2015). The proxy may also be a single point of failure; if it fails, the clients become unable to contact the DBMS. It is possible to circumvent this limitation by replicating the proxy, however this introduces more complexity in the system. NoSQL Undo handles this issue using the built-in logs and snapshots to do recovery, so it does not require an additional server (proxy).

Since NoSQL Undo only accesses the built-in log to perform recovery it does not have control of when log entries are discarded. A database administrator may define a high storage threshold for the log, but an unpredictable peak of traffic may be enough to exhaust that limit. To guarantee that any operation can be undone, the log has to be saved regularly. NoSQL Undo does this using a service that runs along with the database instance and listens to changes in the log, the *Global Log Backup Service* (Figure 5.4). This service is a daemon that is constantly listening to the database for changes and keeping a copy of every log entry in an external database. Every time an operation is executed in the database instance, a new log entry is created, and the global log backup service is notified, then it copies the global log record to another database that should be stored in a different server for availability purposes. This backup operation executes concurrently with the clients' operations and does not require the database to lock until the record was successfully stored, so it does not interfere with the original database functioning.

The last component of the architecture is the Intrusion Detection System, which provides assistance with the process of identifying operations that need to be undone. We postpone the explanation of this component to Section 5.2.4.

5.2.3 Recovery mechanisms

NoSQL Undo uses two methods to undo the effects of incorrect operations leaving the database in a correct state: full recovery and focused recovery. Both methods take as input a list with operations to undo.

Full recovery

The full recovery algorithm is the simplest recovery method among the two. It works by loading the most recent snapshot of the database, then it updates the state by executing the remaining operations, which were previously recorded in a log. The algorithm takes as input a list of incorrect operations that it is supposed to ignore when it is executing the log operations.

Algorithm 1 Full recovery algorithm.

```
1: $recoveredDatabase \leftarrow snapshot$
2: $logEntries \leftarrow getLogEntries(snapshot)$
3: for $logEntry \in logEntries$ do
4:     if $logEntry \notin incorrectLogEntries$ then
5:         $recoveredDatabase.execute(logEntry)$
6:     end if
7: end for
8: return $recoveredDatabase$
```

Algorithm 1 shows the full recovery procedure. The algorithm takes as input the most recent snapshot before the first incorrect operation and a list with the incorrect operations to undo. In line 1 a new database instance is created using that snapshot. The log entries are fetched from the global log (line 2) using the *getLogEntries* method. This method returns an ordered list with every log entry after $snapshot$. Correct operations are executed in line 5. When the algorithm finishes (line 8), $recoveredDatabase$ is a clean copy of the database without the effects of incorrect operations. This algorithm is simple and effective but is not efficient when there are a small number of operations to undo, since it requires every correct operation in the log after the snapshot to be re-executed.

Focused recovery

The idea behind focused recovery is that instead of recovering the entire database just to erase the effects of a small set of incorrect operations, only compensation operations are executed. A compensation operation is an operation that corrects the effects of a faulty operation. The algorithm works basically the following way. For each faulty operation the affected record is reconstructed in memory by NoSQL Undo. When the record is updated, NoSQL Undo removes the incorrect record and inserts the correct one in the database. On the contrary of Algorithm 1, this algorithm only requires two write operations in the database for each faulty operation.

Algorithm 2 describes the process of erasing the effect of incorrect operations. The algorithm iterates through every incorrect operation (line 1). For each incorrect document it fetches every log entry that affected this record (line 3). For simplicity the pseudo-code assumes that there is no older snapshot and that every operation executed is in the log, therefore the recovered document is initialized empty (line 4). If there was a snapshot, then the recovered document would be initialized as a copy of the incorrect document in the snapshot. Then the reconstruction begins and every correct operation is executed in memory in the recovered record, not in the database (lines 4 to 6). Once every correct operation is executed, the record is ready to be inserted in the database. First the incorrect record is deleted (line 10) and finally the correct one is inserted (line 11).

Algorithm 2 Focused recovery algorithm.

```
 1: for $incorrectOperation \in incorrectOperations$ do
 2:     $corruptedRecord \leftarrow incorrectOperation.getRecord()$
 3:     $recordLogEntries \leftarrow getLogEntries(corruptedRecord)$
 4:     $recoveredRecord \leftarrow \{\}$
 5:     for $recordLogEntry \in recordLogEntries$ do
 6:         if $recordLogEntry \neq incorrectOperation$ then
 7:             $recoveredRecord \leftarrow updateRecord(recoveredRecord, recordLogEntry)$
 8:         end if
 9:     end for
10:     $database.remove(corruptedRecord)$
11:     $database.insert(recoveredRecord)$
12: end for
```

Comparison of the two recovery schemes

Both methods are capable of removing the effects of undesirable operations, but there are differences. Focused recovery does not require a new database to be created ($recoveredDatabase$) because it does compensation operations in the existing database. For each record affected by incorrect operations, it does two write operations in the database: one to remove the corrupted record, and another to insert the fixed record. On the contrary, with full recovery every correct operation executed after the last correct snapshot is executed in a new database instance. In terms of writes in the database, focused recovery is much lighter if the number of incorrect operations is reasonably small. On the contrary, if the number of incorrect operations is greater than the number of correct operations in the log, then using the full recovery will be more efficient because there are less write operations to execute in the database (see Section 5.4.2).

Although the algorithms leave the database in a consistent state, a user that has read a corrupted document and suddenly reads the corrected document may believe that the state is inconsistent. To solve this problem, the tool can be configured to leave a message to the users so that they understand why the state suddenly changed (Brown and Patterson, 2003).

Both algorithms are able to remove the effects of faulty operations, but they require the database to be paused, i.e., not executing operations while recovering. If the database keeps serving clients, then data consistency after recovery cannot be guaranteed.

5.2.4 Detecting incorrect operations

NoSQL Undo provides an interface for administrators to select which operations should be discarded during recovery. This interface provides searching capabilities making it easier to find incorrect operations. An interesting case to use this tool is to do recovery from intrusions.

One of the problems in recovering faulty databases is to detect when the database became corrupted in the first place. Detecting the incorrect operations in a log with millions of operations can be a difficult and error prone task. Searching for regular expressions of possible attacks may not be sufficient since a database administrator may not remember every search pattern to all possible malicious operations.

To cope with this, it is possible to deploy alongside with the NoSQL database an Intrusion Detection System (IDS). This IDS permanently listens to the requests to the database and if they match a certain signature, it logs this operation as suspicious. The most conspicuous case of requests that match signatures are attempts of doing NoSQL injection attacks, which are similar to SQL injection attacks but target NoSQL databases (OWASP, 2014; Scambray et al., 2011). Later, when NoSQL Undo is being used, it first consults all the suspicious operations and suggests them to the database administrator who then decides if they should be removed from the database. This automates the identification process and reduces the time to recovery, which can be critical in a highly accessible database. An example of an IDS that can be deployed in this manner is Snort (Roesch, 1999). Different solutions to detect malicious operations in the log can be used, such as (Lee et al., 2002; Hu and Panda, 2004, 2003).

5.2.5 Recovery without configuration

An interesting feature of NoSQL Undo is that it does not have to be configured *a priori* to be able to recover a database. If an incorrect operation is detected soon enough, i.e., while it is still present in the log, then it is possible to remove the effects of this faulty operation without any previous configuration of NoSQL Undo. This is interesting as many organizations do not take preventive measures to allow later recovery.

This approach however has some limitations in comparison to the full-fledged recovery scheme presented in the previous sections. When a recovery service uses specific logging mechanism it is able to store additional information useful for later recovery (e.g., dependencies between operations, origin of the operations and versions of the affected documents). With this extra information it is possible to improve the recovery process as well as provide more information to the database administrator to help him find the faulty operations.

5.3 Implementation with MongoDB

To evaluate the proposed algorithms, a version of NoSQL Undo was implemented in Java. This instance of the tool allows recovering MongoDB databases.

5.3.1 MongoDB

MongoDB supports replication to guarantee availability if servers fail, and horizontal scaling to maintain performance when the traffic load increases (Chodorow, 2013; MongoDB, 2016). A set of servers with replicated data is called a *replica set*. In each replica set there is a server that coordinates the replication process called *primary*, while the remaining servers are called *secondary*. Each secondary server synchronizes its state with the primary. It is possible to fragment data records into MongoDB instances, called *shards*, to balance load. A shard can be either a single server or a replica set. Data in MongoDB is structured in *documents*. Each document contains a set of key-value pairs. A set of related documents is a *collection*. The documents in a collection do not need to have the same set of key-value pairs nor the same type, as opposed to relational DBMSs that impose a strict structure to the records (rows) of a table.

MongoDB uses two logging mechanisms: *journaling*, which is the *local log* used to recover from data loss when a single server crashes; and *oplog*, which is the *global log* that ensures data consistency across replicas of a replica set. From time to time a database copy, called a *snapshot*, is saved in external storage and the journal logs are discarded; otherwise they would grow indefinitely.

In relation to journaling, MongoDB uses a local log called journal to recover a single server that failed unexpectedly. Every time MongoDB is about to write to disk, it first logs the write to a journal file. The journal is then used to recover a single replica that failed without intervention by other servers. The journal file contains non-human readable, binary, data so it is hard to process.

Oplog is the global log used by MongoDB. The primary server uses it to log every operation that changes the database. The oplog collection has limited capacity, so MongoDB removes older log entries. The oplog is stored in a (MongoDB) database, not in a file as logs usually are.

When a database statement is executed, it is stored in the oplog as simpler operations, e.g., an update statement that affects n documents is translated into n simple operations each one affecting a single document. Only operations that change the state of the database (writes) are logged. Each oplog entry contains the following fields:

- **ts:** a tuple that represents the timestamp of when the operation was executed. The first element of the tuple contains a long value that represents a timestamp and the second element contains an ordinal that is used to order operations that were executed simultaneously;

- **h:** a long integer with a unique id for each operation;

- **v:** the version number of the document affected by this operation;

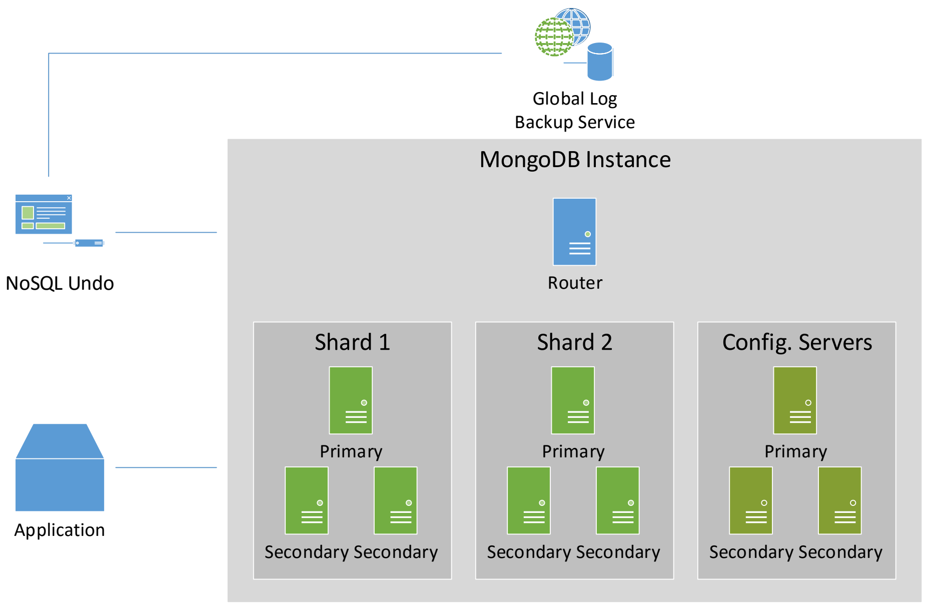

Figure 5.5: NoSQLUndo in a MongoDB instance.

- **op:** represents the executed operation. It can be: 'i' for an insert, 'u' for an update, 'd' for a delete, 'n' for no operation, and 'c' for creating a database;
- **ns:** the namespace, a string with the name of the database and the name of the collection affected by this operation separated by a dot (.);
- **o:** the document used to execute this operation. In an insert it is the document inserted, in the update is a document with the new fields used to update and in the delete is the document used as a search query;
- **o2:** a document with a search query that is used to execute an update (used only in update operations).

5.3.2 NoSQL Undo with MongoDB

The integration of NoSQL Undo with MongoDB is pretty straightforward and follows the architecture represented in Figure 5.4. We installed the Global Log Backup Service in the same network of the database. NoSQL Undo accesses both the database and the backup of the log in order to perform recovery and undo operations.

Figure 5.5 represents the architecture of the deployment used in the experimental evaluation of this work.

We used two scenarios in the experiments. *Scenario 1* corresponds to Figure 5.4. It is a fully distributed instance of MongoDB that was installed in 10 EC2 machines of *Amazon AWS*. The database is divided in two shards, each one containing 3 servers (one primary and two secondary). The configuration servers are grouped in a replica set since that is how MongoDB uses the configuration servers. Finally, there is a single router (unlike the 2 in the figure), which is a MongoDB server that is responsible for redirecting the requests to the correct servers. All the servers have the exact same configuration: t2.small instances with a 1 vCPU and 2GB of memory and running Ubuntu 14.04LTS.

We also used a second configuration in our experimental evaluation for diversity: *Scenario 2*. In that scenario NoSQL Undo was deployed in Google Compute Engine. The deployment was composed by a single replica set with 4 machines: 1 primary and 3 secondaries. Each machine had 1 vCPU and 4GB of memory. The OS used was Debian 8.

5.4 Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions using the implementation of NoSQL Undo for MongoDB: (1) what is the performance trade-off between focused recovery and the full recovery mechanism? (2) How long does it take to undo different numbers of operations? (3) How does the number of versions of a file affects the time to recover?

To inject realistic workloads we used YCSB (Cooper et al., 2010), a framework developed to evaluate the performance of different DBMSs using realistic workloads. Some examples of DBMSs supported by YCSB are MongoDB, Cassandra (Lakshman and Malik, 2010), Couchbase (Brown, 2012), DynamoDB (Sivasubramanian, 2012), and Hadoop HBase (White, 2009). We choose this framework because it is widely adopted for benchmarking NoSQL DBMSs, it provides realistic workloads, and has several configuration options: number of operations, amount of records to be inserted, distribution between reads and writes.

The YCSB workloads used in the experiments were: (A) update heavy, composed by 50% reads and 50% writes; (B) read mostly, with 95% reads and 5% writes; (C) only read, without write operations; (D) new records inserted and the most read, simulating social networks and forums where users consult the most recent records; (E) short ranges, where read operations fetch a short range of records at a time, like a conversation application, blogs and forums; and (F) records updated right after read, simulating social networks when users update their profile.

MongoDB offers two different interfaces: *synchronous*, where only one operation can be submitted at a time, and *asynchronous*, where operations can be executed in parallel.

5.4.1 Global log backup service overhead

The Global Log Backup Service runs alongside with the database. It listens for changes in the log, so when an operation is executed in the database server, the Global Log Backup Service is notified and saves the operation in external storage. To evaluate the throughput overhead of using this backup service we executed several workloads of YCSB 10 times each using Scenario 2.

Figure 5.6 presents the average throughput (operations per second) of 10 executions of several workloads of YCSB using both the asynchronous and the synchronous driver with a confidence level of 95%. The cost of having this additional service varies from 6% to 8% when using the asynchronous driver, and from 20% to 30% when using the synchronous driver. The overall throughput seems acceptable

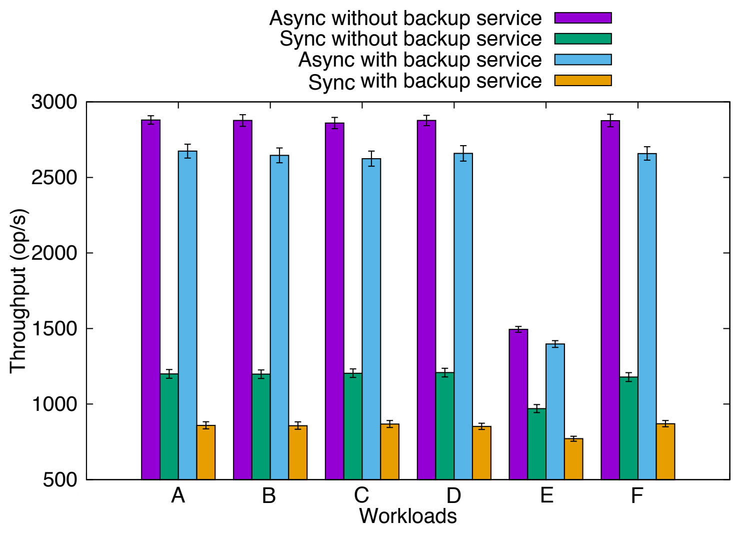

Figure 5.6: Overhead of using the Global Log Backup Service with a confidence level of 95%.

given the advantages of storing every executed operation in the database.

In terms of storage, after executing every workload of YCSB the database occupied 100MBs in disk while the global log backup occupied 120MBs. The overhead in terms of storage is considerable (120%), but this is an unavoidable cost of supporting state recovery.

5.4.2 Focused recovery versus full recovery

To evaluate how focused recovery performs in comparison with full recovery a set of operations were undone from the database using both algorithms in Scenario 1. The size of this set varied from 1 to 10,000. Each case was repeated 10 times. The goal of using different sets of incorrect operations to undo was to understand how both algorithms perform when they undo from just a few operations to almost every operation in the database.

Figure 5.7 shows the average time to recover using both methods, with a confidence level of 95%. The full recovery method performs better as the number of operations to remove increases, which makes sense as less operations have to be executed. On the other hand, the focused recovery method performs a lot better when there are just a few operations to be removed and it degrades the performance linearly as the number of operations increases. Focused recovery takes almost a second to remove 1 operation, whereas full recovery takes around 700 seconds. Both methods achieve a similar performance around 5,000 operations to be removed. This result shows that for a small number of operations to be removed, focused recovery is a better choice, but if more than 60% of the operations are incorrect then the full recovery method should be used.

5.4.3 Recovery with different versions

The focused recovery method reconstructs every document affected by incorrect operations, meaning that if a document has a thousand versions and one of them is incorrect, then the focused recovery

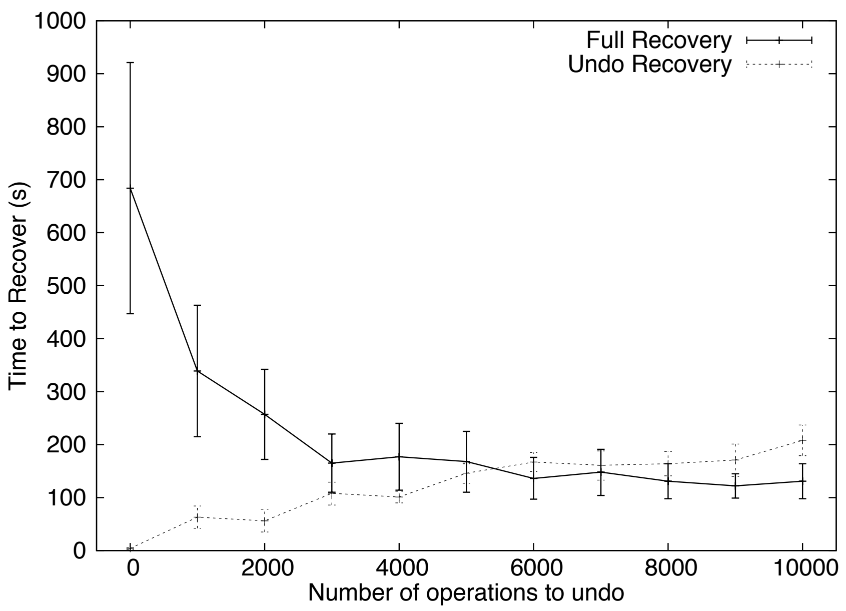

Figure 5.7: Focused recovery performance vs full recovery time with a confidence level of 95%.

method needs to re-execute the remaining 999 operations in order to reconstruct the document correctly. To evaluate how focused recovery is able to remove incorrect operations in documents with different number of versions, we executed both focused and full recovery methods in a document varying the number of versions from 1 to 100,000 in steps of 10. Each recovery execution was repeated 10 times. These experiments were conducted in Scenario 1. The average recovery time of each execution can be seen in Figure 5.8. The time to recover increases exponentially for the focused recovery, while it remains almost constant for the full recovery. This result was expected since the full recovery needs to undo one incorrect operation in every case. Focused recovery has more work to do if the number of versions of a document increase. This is because it needs to reconstruct the affected record.

5.4.4 Intrusion detection overhead

Using an IDS to tag incorrect operations facilitates the job of the database administrator. When an alarm is triggered the administrator consults the IDS log and can immediately undo incorrect operations, instead of browsing the entire database log for incorrect operations. To evaluate the cost of using an IDS to detect incorrect operations, we set up an extra machine (with the same characteristics of the others) running Snort in Scenario 1. We then added 10 rules to Snort and executed every YCSB workload.

Figure 5.9 shows the overhead of using Snort. The throughput is degraded by 10 to 30%. It is a considerable cost given the benefits of allowing the database administrator to recover a database immediately without losing time searching in the database log for incorrect operations.

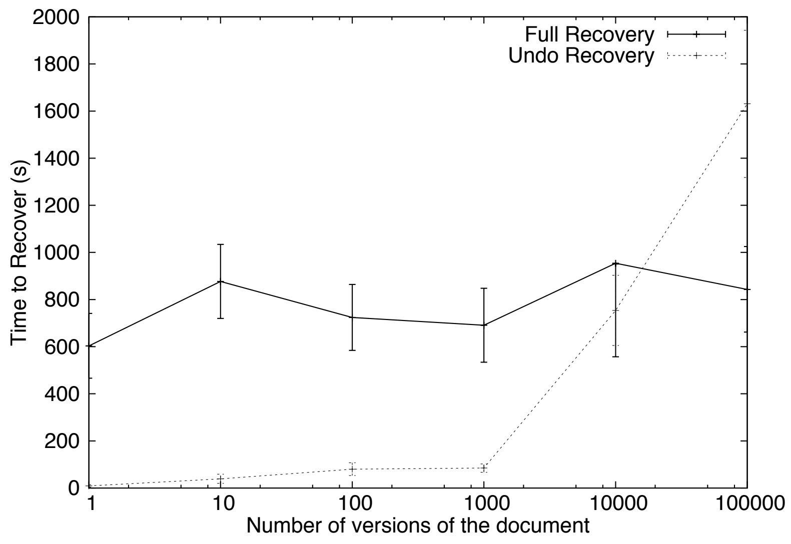

Figure 5.8: Focused and full recovery a document with different versions with 95% confidence level.

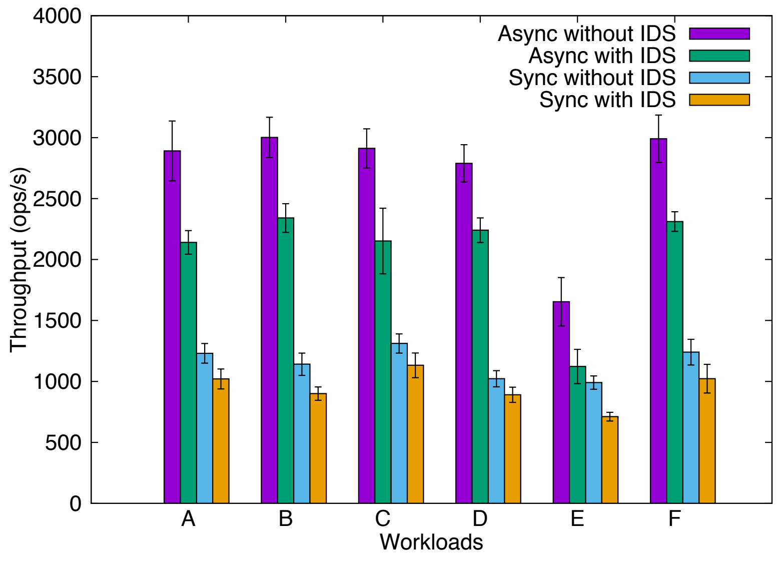

Figure 5.9: Overhead of using Snort to detect incorrect operations in a MongoDB Cluster.

5.5 Summary

This chapter presents a tool that allows the database administrator to remove incorrect operations from a NoSQL database. It runs as a client of the database management system and uses its built-in replication log and snapshots to do recovery. As presented in Table 5.1, such logs can be found in different NoSQL databases, making it possible to adopt NoSQL Undo in a variety of NoSQL databases. Like in (Goel et al., 2005b; Hsu et al., 2006), NoSQL Undo provides two different approaches to recover a database: focused and full recovery. Both methods are capable of recovering databases, but there is a trade-off between performance and number of operations to undo.

The tool allows recovering corrupted documents with just two database operations for each affected document and in much less time than other full recovery schemes. The tool recovered 10,000 documents in a database with 1,000,000 operations in the log in less than 150 seconds using full recovery and less than 200 seconds with focused recovery, which are relatively short periods of time (other systems have values in the order of tens of minutes or even hours).

6 Rectify: Black-Box Intrusion Recovery in PaaS Clouds

In this chapter 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 on web application recovery. 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 removing 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 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 undo 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) (Kruegel et al., 2005; Robertson et al., 2006; Ingham and Inoue, 2007; Nascimento and Correia, 2011) 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.

The main challenge of this work is 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 (Nascimento and Correia, 2015), 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* (Kotsiantis, 2007) 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.

Rectify was implemented in Java and configured as a container ready to be deployed in PaaS offerings. This implementation is available online.[1] We run it in the Google App Engine (Ciurana, 2009). 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.

Figure 6.1 shows where Rectify fits in the common cloud architecture. Rectify runs in two cloud levels: IaaS, by intercepting database statements, and in the PaaS, intercepting user operations.

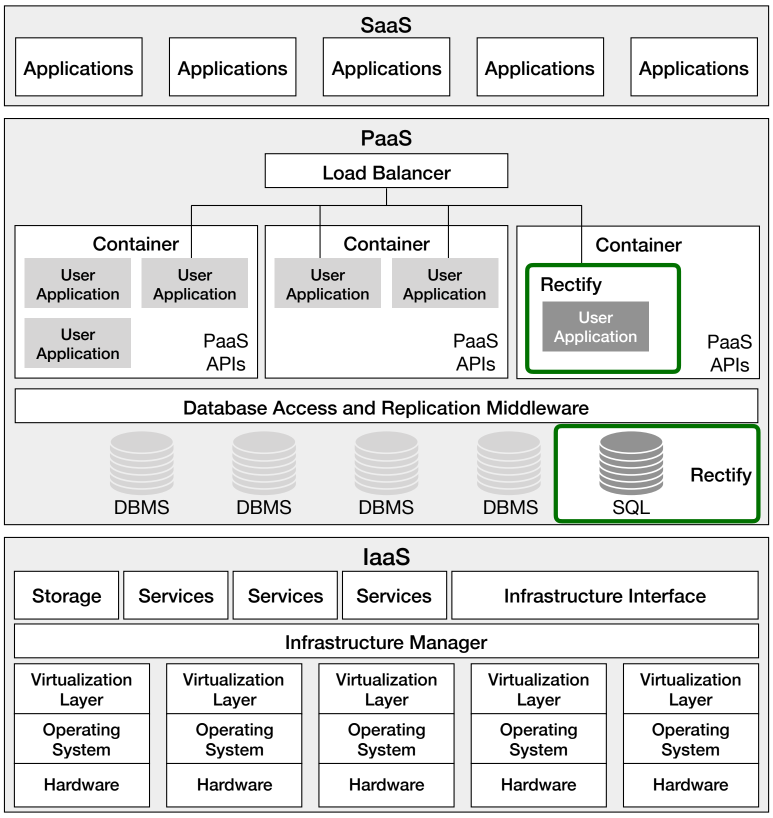

Figure 6.1: Rectify in the common cloud architecture.

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

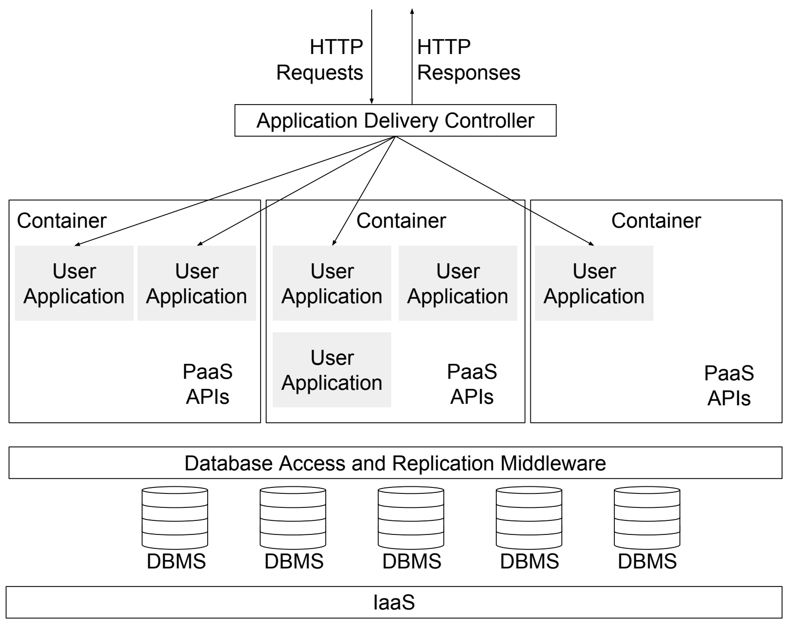

Figure 6.2: Typical PaaS Architecture.

6.1 PaaS Architecture

Platform as a Service is one of the three original cloud computing models, alongside IaaS and SaaS (Section 2.1.1).This model supports automated configuration and deployment of applications in datacenters. 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 6.2 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 (Peinl et al., 2016). 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 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 Rectify 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.

6.2 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*.

6.2.1 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 like a hacker. Or because it is done by a legitimate client that abuses the privileges of another client. 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* (Brown and Patterson, 2003) 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* (Korth et al., 1990), which removes the modifications that resulted from a malicious operation.

6.2.2 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 work as we focus on recovering the state of the application.

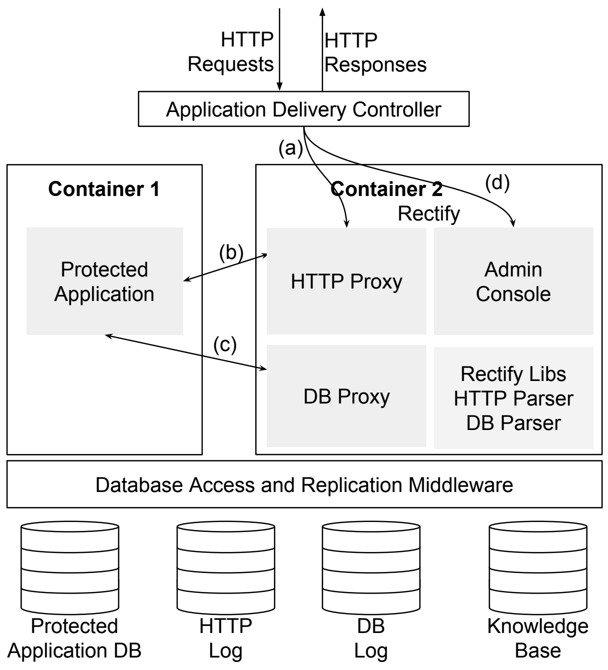

Figure 6.3: Rectify system architecture. Rectify itself is the set of components inside Container 2.

6.2.3 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 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 (Varia and Mathew, 2014). 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 6.3 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 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., username and password).

Next, we present these components of the Rectify service 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 6.3.

6.3 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 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 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 6.4.3, 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 (Heydon and Najork, 1999), 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 6.1 presents the signature record of the HTTP request `r`. The record is divided in HTTP and

	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]

Table 6.1: Example of a signature record

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.

6.4 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 6.4 presents a level-0 data flow diagram with the tasks performed in these two classification

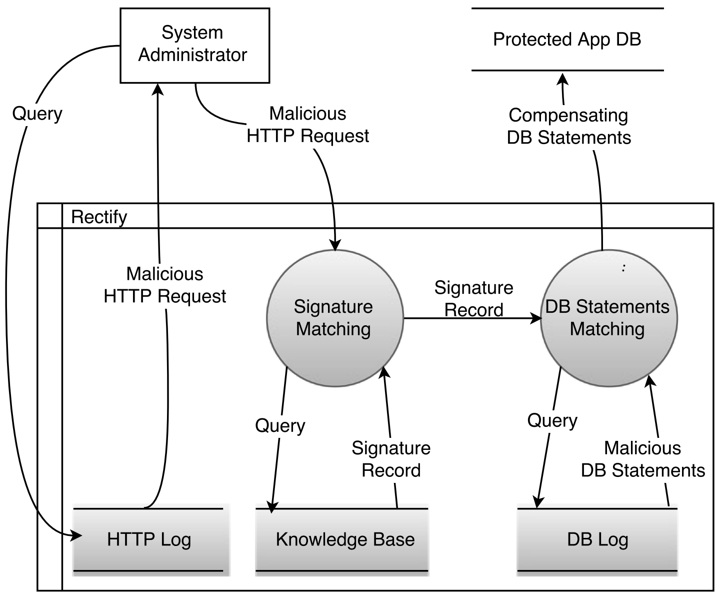

Figure 6.4: Level-0 data flow diagram of the tasks performed to solve the 2-step classification problem.

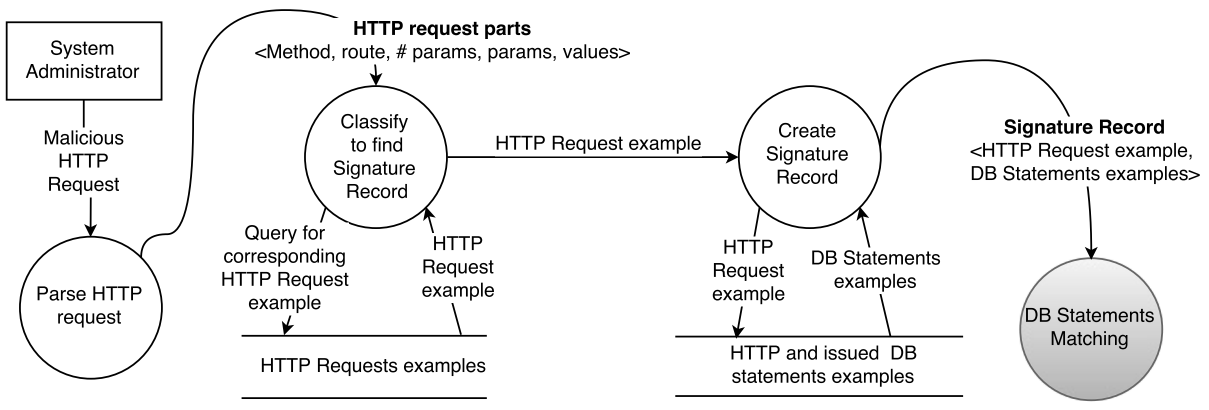

Figure 6.5: Level-1 data flow diagram with the tasks performed to identify the signature record of a malicious HTTP request.

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.

6.4.1 Step 1 - signature matching

In the first step, Rectify solves the classification problem of identifying the signature record of an HTTP request. Figure 6.5 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 to extract its relevant parts (shown in Table 6.2). 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

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

Table 6.2: Features used to classify HTTP requests

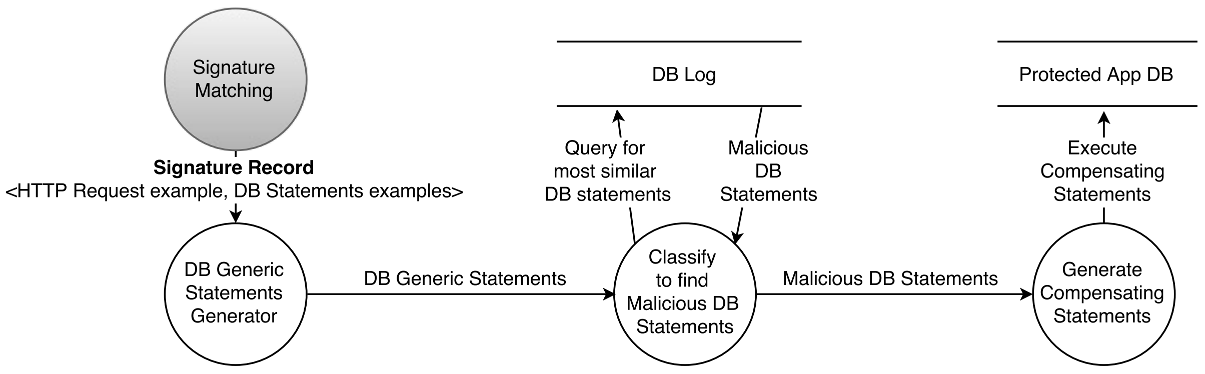
Signature Matching
Signature Record
<HTTP Request example, DB Statements examples>
DB Generic Statements Generator
DB Generic Statements
DB Log
Query for most similar DB statements
Malicious DB Statements
Classify to find Malicious DB Statements
Malicious DB Statements
Protected App DB
Execute Compensating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

Figure 6.6: 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.

request. In this case the class is a signature record. A more detailed explanation of how classifiers work can be found in (Kotsiantis, 2007). 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.

6.4.2 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 6.6 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 3.

Algorithm 3 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: $GenericStmts \leftarrow \perp$
 2: for $stmt \in sr.getDBStatements()$ do
 3:   $p\_stmt \leftarrow stmt$
 4:   for $col \in p\_stmt.getColumns()$ do
 5:     if $sr.valueComesFromHTTPRequest(stmt, col)$ then
 6:       $attr \leftarrow sr.getHTTPAttrFromDBCol(col)$
 7:       $value \leftarrow mr.getValue(attr)$
 8:       $p\_stmt.setColumnValue(col, value)$
 9:     else
10:       $p\_stmt.setColumnValue(col, NULL)$
11:     end if
12:   end for
13:   $GenericStmts \leftarrow GenericStmts \cup p\_stmt$
14: end for
15: return GenericStmts
```

Algorithm 3 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

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

Table 6.3: Features used to classify database statements

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).

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 6.3.

The classification problems presented in Sections 6.4.1-6.4.2 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 Neighbors* and many others. A full list of the studied algorithms is presented in Section 6.7.

6.4.3 Dealing with SQL injection

One of the most common attacks against web applications is SQL injection (Halfond et al., 2006a).[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 6.4.1 and 6.4.2 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

[2]Despite the term SQL in the name, all data access languages based on statements are vulnerable to such attacks including NoSQL statements 5.

all the statements in S that are not in S_{good}.

6.5 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* (Korth et al., 1990). 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 (Ammann et al., 2002; Liu et al., 2004).

We based our approach to deal with dependencies on the work presented in (Ammann et al., 2002). 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 (Nascimento and Correia, 2015).

The compensation operations that undo the effects of a malicious statement depend on the type of malicious statement. A statement that creates a record needs to be undone in a different way than a statement that updates a record.

The algorithm to calculate and execute the compensation operations is the *two pass repair* algorithm (Ammann et al., 2002) 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 6.2.3, `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 list 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. Note that Rectify, could not execute an *update* to undo a *delete* since there would be no record to update.

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 6.2.1, 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.

6.6 Implementation

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

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 6.4 lists the number of classes and lines of code of each module in our implementation.

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

Table 6.4: Number of classes and lines of code of our implementation of Rectify

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 (Ciurana, 2009) PaaS offering, which provides JEE containers.

6.7 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* (Brazell, 2011), *LimeSurvey* (LimeSurvey, 2017) and *MediaWiki* (Barrett, 2008). *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 not possible to group the actions taken by an attacker.

6.7.1 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 (Halili, 2008) 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.

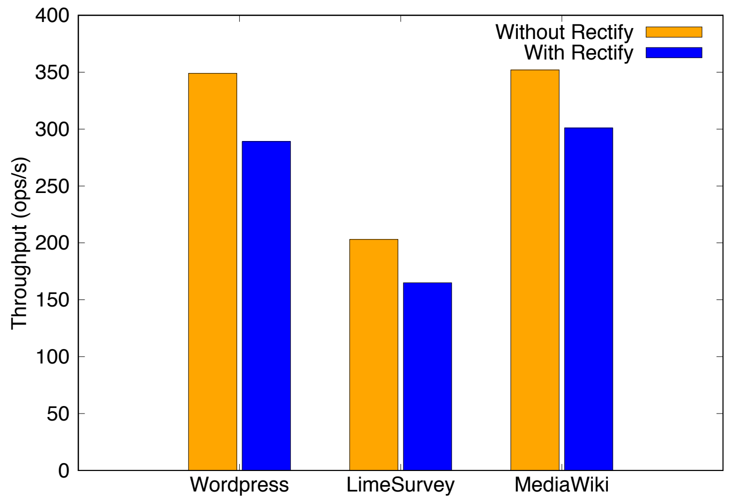

Figure 6.7: Performance overhead of using Rectify with three different applications measured in operations per second.

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

Figure 6.7 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.

6.7.2 Space overhead

The space occupied by the logs of Rectify will grow over time. We wanted to know how much space Rectify needs to store the logs of a certain number of requests. After the experiments presented in Section 6.7.1, we checked how much space the logs were taking in the database (uncompressed). Table 6.5 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 storing the logs in an external cloud storage service, such as Amazon S3 (Palankar et al., 2008). This way the scalability of Rectify would be managed automatically and the cost would be lower than storing the logs in the PaaS container.

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

Table 6.5: Space occupied by the Rectify logs in each application after 1,000,000 HTTP requests.

6.7.3 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 6.8 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.

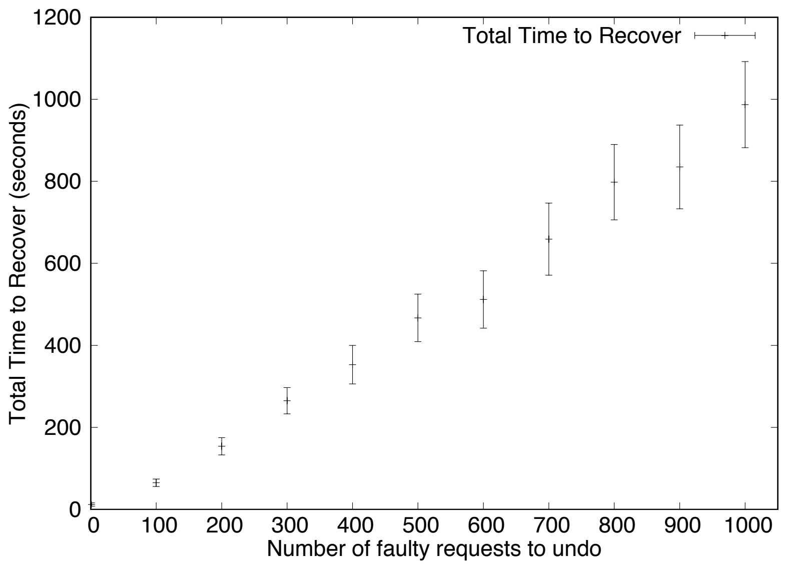

Figure 6.8: Total time to recover the protected application.

6.7.4 Accuracy of the classification algorithms

There are several algorithms that can be used to solve the classification problems presented in Section 6.4. We executed every classification algorithm available in Weka (Hall et al., 2009), 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: $\mathit{Accuracy} = (\mathit{TP} + \mathit{TN})/(P + N)$, 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.

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 6.6. 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 (Word-

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 6.6: Accuracy of the tested classification algorithms performing the first classification problem.

press, 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.

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 (Safronov and Winesett, 2014) 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
```

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.

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 6.7 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.

6.7.5 Identifying SQL injection statements

As mentioned in Section 6.4.3, 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 (Gegick et al., 2006). 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.

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%

Table 6.7: Accuracy of the tested classification algorithms performing the second classification problem

6.8 Summary

This chapter 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. The classification algorithms rely on the use of regular routes (well-structured URL patterns), a common good practice of web development. Our implementation was able to recover 1,000 malicious HTTP requests in around 16 minutes.

7 μVerum: an Intrusion Recovery Approach for Microservices Applications

This chapter presents μVerum, a novel framework designed for microservices applications, capable of recovering from intrusions without stopping the application. It focuses on recovering the affected services by executing compensating operations (Korth et al., 1990) that undo the effects of intrusions while preserving the valid data recorded in the system. μVerum takes into account the current trends in the development of microservices applications, making it possible to adapt existing applications to μVerum as a gradual process, without requiring the microservices' developers to modify the entire application code at once. μVerum assumes a system architecture with certain components, such as routers and a discovery service, which can be found in real world microservices applications (Wang and Tonse, 2018). The framework provides the following capabilities:

- **Traceability of requests:** there is a search engine that allows administrators to find every operation, among the microservices, that was caused by an unintended request issued by an attacker. After pointing a malicious request, μVerum presents the administrator a graph with the operations that were executed by the microservices and how they are related. This graph allows the administrator to trace the effects of the intrusion and select which operations he wishes to undo.

- **Non-stop intrusion recovery:** the ability to revert unintended actions by undoing, in the state of the application, everything that was caused by them. It does not require the application to be offline during the process. μVerum performs recovery by executing compensating operations (Korth et al., 1990; Lomet, 1992), i.e., special operations that when executed undo the effects of a previous, undesired, operation. These operations are concurrently executed alongside users' requests.

- **Modular design:** allows configuring μVerum to only intervene in some components of the application that need to be repaired, without degrading the performance of components that do not need to be recovered, for example, stateless services. This modular design also allows developers to adopt the μVerum approach in a gradual way.

- **Consistency:** μVerum allows developers to define invariants to ensure consistency of the application during and after recovery. These invariants are taken into account when μVerum is recovering. Section 7.3.6 formally describes the consistency invariants.

μVerum reverts malicious operations identified by the administrator of the application, using tools such as the μVerum search engine or an intrusion detection system (IDS) (Roesch, 1999; Ingham and Inoue, 2007; Kruegel et al., 2005; Nascimento and Correia, 2011; Robertson et al., 2006). We do not aim to solve the problem of intrusion detection since it has been widely studied for decades and is distinct from the problem of intrusion recovery.

We evaluated μVerum by performing experiments with the microservices application SockShop (WeaveWorks, 2018). Our experiments show that μVerum affects the performance of the application by less than 7% which can be reduced by scaling the agents of the most accessed microservices. Our prototype of an application with the μVerum approach is capable of recovering faulty requests keeping the availability of the application. To the best of our knowledge, this is the first intrusion recovery approach specifically designed for microservices applications.

Figure 7.1 indicates where μVerum is located in the cloud architecture presented throughout this thesis. In the figure there is a μVerum instance protecting each microservice that is running in its own container. The PaaS load balancer then contacts the microservice through μVerum, allowing every operation to be logged.

7.1 Monoliths vs Microservices

The development of complex web-applications has shifted from the traditional monolithic architecture to a distributed approach. By adopting this approach organizations are able to divide developers in small teams, each one being responsible for a self-contained component of the application. Such component is called a *microservice* (Newman, 2015; Thönes, 2015; Dragoni et al., 2017) and it communicates with other microservices through an interface accessible over the network using a protocol, such as SOAP (Newcomer and Lomow, 2005) or REST (Richardson and Ruby, 2008). Inter-service communication infrastructures, such as Linkerd (Linkerd, 2018) and Istio (Istio, 2018) are then responsible for dealing with the deployment, monitoring, communication and configuration of groups of microservices, called *services meshes*. This approach allows different kinds of applications (mobile apps, web sites and desktops) to only use the required microservices, instead of requiring a dedicated monolith for each one.

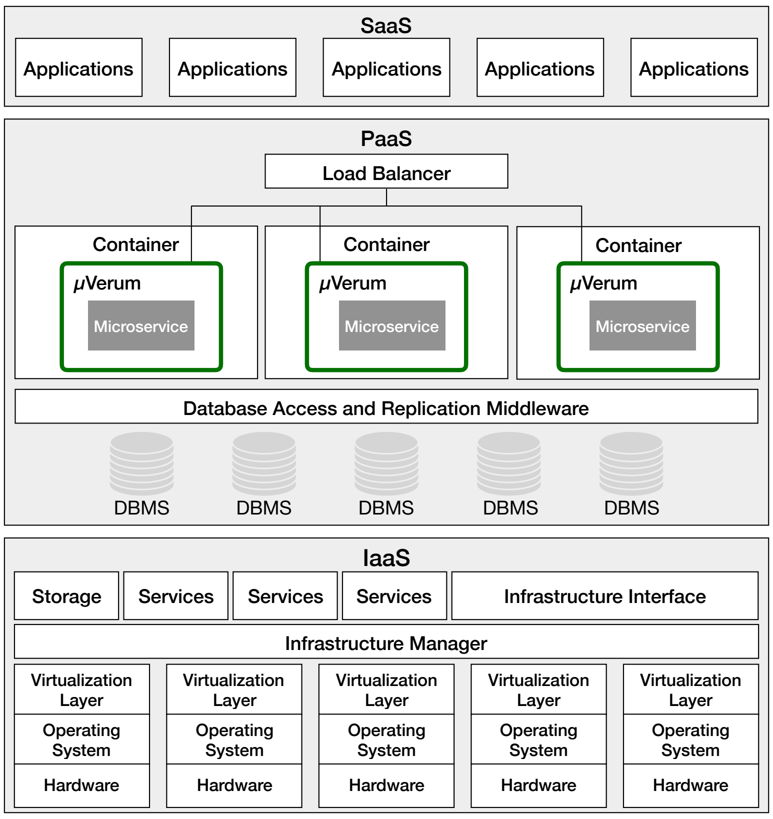

Figure 7.1: μVerum in the common cloud architecture.

Figure 7.2 suggests the differences between a monolithic (left) and a microservices application (right). In the microservices approach each application uses only the components it needs. Microservice applications typically use a database or datastore per service. This allows scalability efficiency, since only the most used components are replicated, instead of the entire system.

When an attack corrupts the state of a microservice, its effects will tend to propagate to other microservices. Since each team of developers is only in charge of a single microservice, it is difficult to understand how an attack globally affects the application and how it can be fixed.

7.2 Microservices

The microservices architecture allows software systems to be developed as a set of small, independent and self-contained services. Each service can be deployed in a specific execution environment in a different physical infrastructure and developed in its own programming language. The microservices communicate using a RESTful (Richardson and Ruby, 2008) (Representational State Transfer) or RPC-based API (Newman, 2015). This way, each service can be developed by small and independent team that only shares APIs with other teams. This allows many benefits when it comes to implement complex business applications, since developers can adopt a *divide and conquer* approach and add new features without requiring a shutdown of the entire application. However, there are some disadvantages. In order to avoid breaking the application every time a new service is added or modified,

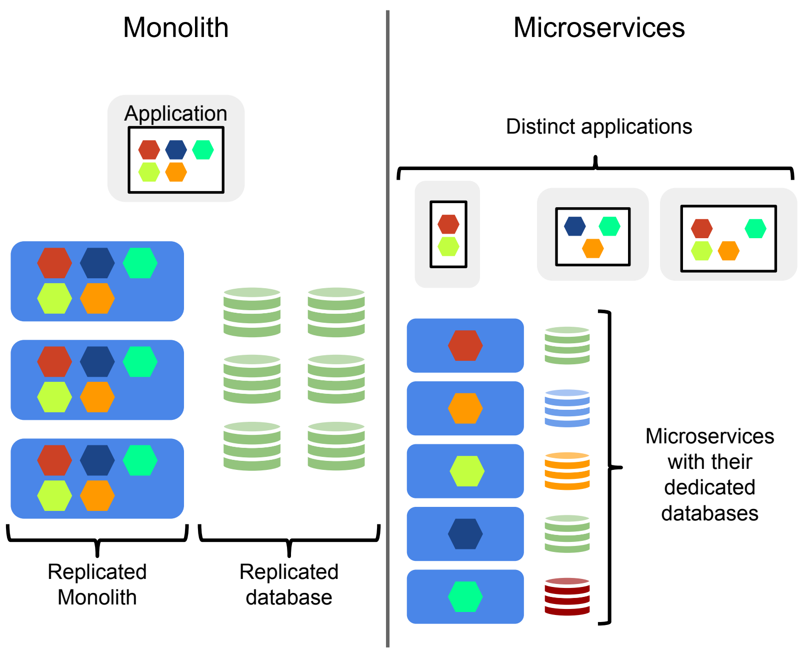

Figure 7.2: Monolith (left) versus microservices architecture (right).

developers need to follow a strategy when adding new services to the application. Some rules, such as stating that each microservice has its own data store and every service report to the same central log, are vital to minimize the bugs and facilitate the process of monitoring and detecting faults. In this approach, when things go wrong it is more difficult to trace the affected components that need to be repaired.

7.2.1 Microservices' APIs

The microservices' API is provided by a server, or service provider, to clients, or consumers. It is assumed that server and clients are distributed, and the API is invoked through the network. There are different types of interfaces, some rely on a formal definition of the available functions enforcing the consumer of the service to use the same technology as the server (tight coupling), while others are more open, allowing different technologies to be used (loose coupling). Some examples of protocols used by microservice applications are remote invocation frameworks, such as Java RMI (Downing-Troy, 1998), .NET Remoting, SOAP (Newcomer and Lomow, 2005) and REST (Richardson and Ruby, 2008).

An API of a service can be *synchronous* if the consumer of the service locks until a response is returned, or *asynchronous* if the consumer proceeds computation without reading back a response from the provider. μVerum uses a synchrony approach to recover in order to confirm that the compensating operations succeeded. The way a set of APIs is executed can be of two forms: *orchestration*, when a consumer calls a provider directly, and *choreography* when the consumer publishes API invocations to a message bus which are then consumed by the providers.

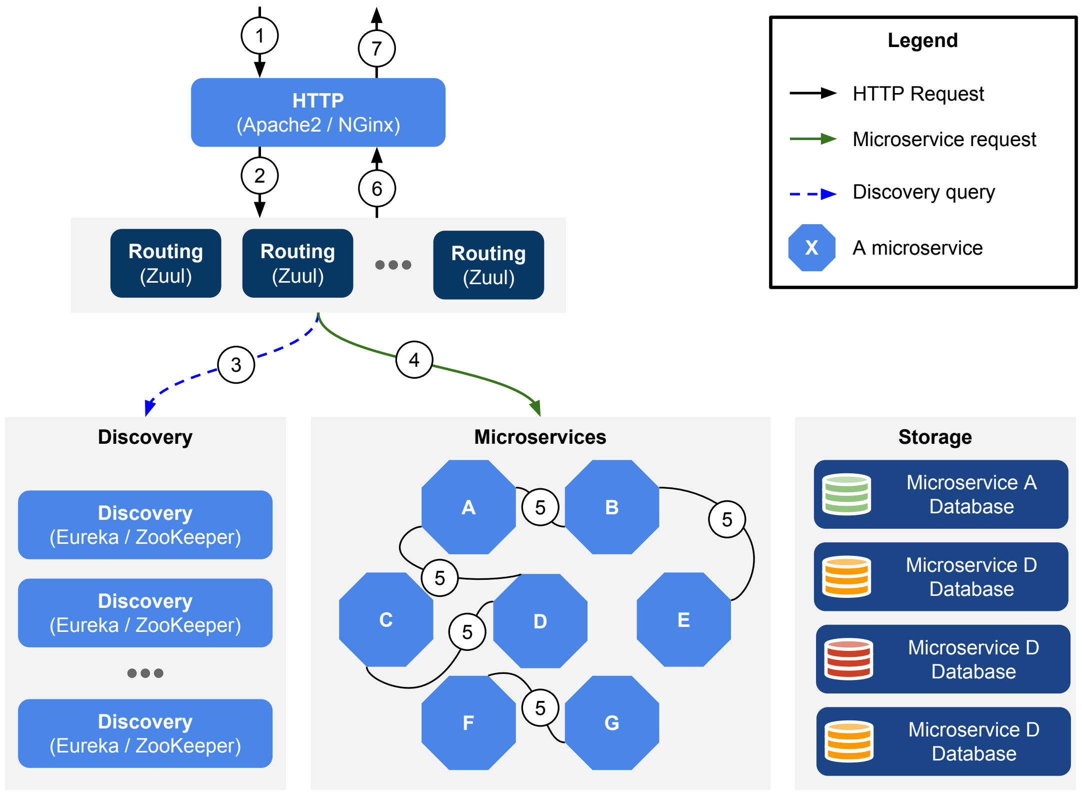

Figure 7.3: Common architecture of a microservices application.

7.2.2 Architecture of a microservices application

Unlike web applications in which there are common architectures (e.g., the MVC model), when it comes to microservices there are many different architecture decisions that influence the structure of the application. Some use routers to guide the traffic to the corresponding services (Wang and Tonse, 2018), others use a central message queue where requests are published by the clients and then consumed by the corresponding servers (O'Meara, 2018), and some may use an hybrid approach (WeaveWorks, 2018) or completely different architecture. Therefore, in order to design a recovery service, it is necessary to assume a common system architecture, otherwise we would be designing an overly general recovery service for every possible system architecture that would not be effective.

We assume a microservices architecture that is partially based in the published work of leading developers from Netflix (Wang and Tonse, 2018; Mauro, 2018) (Figure 7.3). We chose this application given the contribution it had in the development of systems to support microservice applications, such as Zuul (Netflix, 2018), Eureka (NetFlix, 2018) and Ribbon (Wang and Tonse, 2018).

HTTP server

The HTTP server is the only component of the application that needs to be publicly available for the users. Typically, it is deployed in the DMZ of the network and it serves different kinds of consumers of the application (web browser, mobile devices and other applications). This server is stateless allowing it to be replicated, while having a load balancer coordinating the distribution of requests. It is also possible to deploy cache systems that optimize the traffic alongside the HTTP servers. For simplification we will assume that there is only a single HTTP server that serves the clients and interacts with the APIs provided by the microservices.

Router

The router forwards requests to the corresponding services, translating HTTP requests to the server into service requests. The router can be replicated to cope with the traffic. Routers are useful to modify or verify requests on the fly by using filters. This allows developers to incorporate metadata in the requests or to perform integrity checks and encrypt data.

Discovery service

The discovery service is responsible for keeping record of the microservices addresses in the network. They translate a service name into an address when the router does not know it. They also may provide fault detection alerts so that the router knows that the service is not available. The discovery service can be replicated. The use of discovery services is encouraged (Taibi and Lenarduzzi, 2018; Newman, 2015) to prevent errors when a service's address changes, the network structure is updated or new servers are added for replication.

Microservices

The microservices themselves are self-contained. They provide APIs for the consumers and communicate over the network. The consumers of the microservices can either be the routers, that are forwarding requests from the HTTP server, or other microservices. It is assumed that any microservice can communicate with each other. Although this may not be the case in some real-world applications in which some services are isolated, for the remaining of the paper we will assume that services are reachable by any service.

7.2.3 Flow of a request

A user request passes through different components of the application before returning a response to the user. Figure 7.3 presents a common architecture of a microservices application with a trace of a user request. In the figure, when a request reaches the application, it is treated by an HTTP server with a public IP. This server will redirect the request to the router which will, in turn, generate a web services request. It may first consult a discovery service to translate the address of the web service to an IP address. Then, once a microservice receives a request it may generate other requests.

More specifically, the flow goes this way:

1. an HTTP request reaches the HTTP server. This request was issued by a user through a web application or mobile application;

2. if the HTTP request corresponds to a web service, then the server will forward it to a router server;

3. (optional) if the router server does not know the IP address of the corresponding microservice it will first query the discovery service, otherwise it will contact it directly;

4. the request reaches a front-end microservice responsible for dealing with application level request;

5. the front-end microservice may contact other back-end microservice or return a response to the router;

6. the router forwards this response to the HTTP server;

7. the HTTP server sends an HTTP response to the client that issued the initial request.

7.3 The μVerum Approach

The μVerum approach requires several components to be coordinated and reconciled with the application code, following an established pattern of interactions. As long as the application follows the μVerum guidelines then it will be possible to trace the effects of intrusions and later recover from their effects.

7.3.1 System model

μVerum recovers from intrusions that affect applications composed by a set of microservices. These applications are available to users with limited privileges, and maintained by systems administrators with higher privileges. Users interact with web servers, located in the public network, that redirect their requests to a subset of the microservices of the application. Users' requests are in the form of HTTP requests which in turn generate microservices requests that are exchanged between the microservices of the application. More rigorously, we make the following assumptions:

- *S1:* the API is available in a RESTful manner in the JSON format;
- *S2:* users' requests reach the application's microservices from web servers;
- *S3:* users cannot access the microservices that are not publicly available;
- *S4:* only microservices requests intercepted by μVerum agents can be recovered;
- *S5:* the PATCH with JSON method (Dusseault and Snell, 2010; Snell and Hoffman, 2014) from the HTTP protocol is free to use for the recovery process of μVerum (its goals is to fix a previous request).

7.3.2 Threat model

We define an *intrusion* as a malicious request that leads an application to a faulty state. When this happens there are two challenges in terms of recovery: first, it is necessary to detect the trail of the faulty operation, i.e., the subset of microservices that got affected by the malicious request and need to be corrected; and second, to define which compensating actions should be executed in order to lead the system back to the valid state. In the end of the recovery the state of the application should be the same as if the malicious request never happened.

Intrusions can happen when a malicious user exploits a vulnerability in the front-end of the application, e.g., a SQL injection vulnerability (Halfond et al., 2006a). Accidental operations that corrupt the state of the application are also considered as intrusion. More specifically, an intrusion consists in an HTTP request that reaches the application which, in turn, generates several microservices operations causing unwanted changes to the state of the application. We assume that an intrusion can only come from the web servers, i.e., the component that is publicly available for the users of the application, meaning that it is not possible for an attacker to trigger microservices operations from inside the application's network without them first passing through the endpoint.

The threat model is the following:

- *T1:* intrusions come from outside of the application's network;
- *T2:* intrusions cause state modifications to the application's data stores;
- *T3:* both the application and μVerum's microservices are intrusion-proof, i.e., they are not compromised or disabled.

With *T1* we assume that an attacker does not have access to the network in which the microservices are running. This is a reasonable assumption given that most attacks (Scambray et al., 2011) are originated in the front-end of the application. In *T2* although there are some exploits that fail to modify the state of the application, in this work we do not consider them as intrusions since there would not be anything to revert from. *T3* clarifies that our problem is the recovery of the state of the application, not protecting services from attacks that modify their code or configuration.

7.3.3 System architecture

In this section we describe the architecture of a microservices application configured with μVerum (Figure 7.4). In the figure some microservices (those in red/darker) are wrapped by a μVerum agent that intercepts the requests so they can be logged. μVerum requires two databases to keep operation logs (μVerum Log Database) and configuration values (μVerum Config Data).

μVerum admin

μVerum Admin is a microservice that runs alongside the other microservices of the application and controls the recovery process (top-left, grey microservice in Figure 7.4). It is the only component

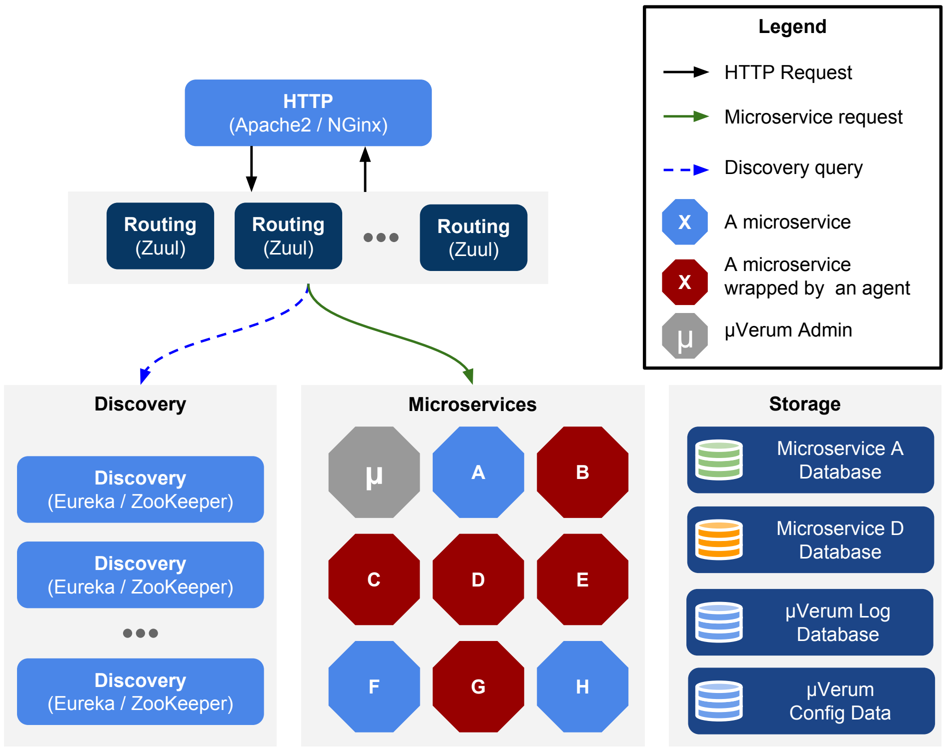

Figure 7.4: System architecture of an application adopting the μVerum approach.

of μVerum that is accessed by the administrator. When it is necessary to recover microservices, the μVerum admin fetches the logs and contacts the agents to execute the recovery operations. It is also through μVerum Admin that the administrator configures the agents.

μVerum router

The router (dark blue boxed at top of the figure) μVerum appends metadata to every request that will allow to correlate and order requests. This metadata consists in a unique serial id value. This technique of tainting requests by appending metadata in order to trace them in the future has been explored in previous works (Nguyen-Tuong et al., 2005; Papagiannis et al., 2011; Halfond et al., 2006b). With this serial id it is possible to order the HTTP requests that reach the application (from the HTTP servers) and fetch every microservice operation that was executed as a result of that HTTP request.

It is worth mentioning that the router may be distributed and/or stateless. This makes it difficult for the replicated router servers to negotiate upon a unique and serializable id. We solve this problem by delegating the generation of unique serializable ids to the μVerum log, as explained next.

μVerum log

The μVerum log is a distributed NoSQL database in which users' requests and microservices operations are logged. We chose a NoSQL database to store this information for three reasons: first, we need a database that imposes as low overhead as possible since we are writing every operation that occurs in the application. For simple write operations there are NoSQL databases that outperform SQL databases (Li and Manoharan, 2013); second, we do not need to use relations between records hence, there is no need to use a relational database; third, given that a log is an append only system, we are not interested in deleting log entries, updating them or performing concurrent reads, therefore we do not need to worry about a high level of consistency.

This approach of collecting every log entry of the several microservices to the same log database is recommended for microservice applications since it allows the administrator to view and analyze the application history as a whole instead of a collection of parts (Newman, 2015). Another advantage of using a single log for every microservice is that it allows the log to generate single and serializable ids as well as recording the execution timestamps that can be used to order and correlate events. As explained in the following sections, such information is necessary during recovery to maintain consistency.

μVerum agent

The μVerum agent is responsible for intercepting requests that reach the application's microservices. The agent is located alongside the microservice that is being monitored. For performance and management reasons, the μVerum agent can be replicated and each μVerum agent replica can deal with one or more microservices.

By default, μVerum agents log operations synchronously, i.e., a request is only forwarded to the corresponding microservice after it was logged. This way μVerum ensures that even if an agent crashes every operation that was executed by the application was logged and therefore can be undone. This approach may affect the performance of the application, that is why μVerum also allows agents to log operations asynchronously, allowing requests to be forwarded to the corresponding microservice immediately and the log entry is scheduled to be recorded in parallel. This approach has a drawback, if the μVerum agent crashes after redirecting the request and before logging it, then the operations are lost.

Besides the requests that reach the microservice, the μVerum agent also collects the status codes and the response timestamps. The status codes are used to avoid logging operations that failed. The timestamps will be used to order and correlate the execution of the requests. During recovery, μVerum executes concurrent requests in parallel to reduce the overall time to recover. Concurrent requests happen when they overlap their execution. More formally, any two operations o_1 and o_2 with timestamps $start\_ts_{o_1}$, $end\_ts_{o_1}$, $start\_ts_{o_2}$ and $end\_ts_{o_2}$ are concurrent if $(start\_ts_{o_2} < end\_ts_{o_1} \wedge end\_ts_{o_1} < end\_ts_{o_2}) \vee (start\_ts_{o_1} > start\_ts_{o_2} \wedge end\_ts_{o_1} < end\_ts_{o_2})$.

7.3.4 Design of a μVerum application

In order to use μVerum in a microservices application the developers have to follow a set of guidelines. These guidelines are aligned with good practices of software development like following the MVC pattern while developing a web application.

First, microservices have to interact with each other exclusively through HTTP using REST APIs.

Developers are free to use different protocols, however, operations that are executed using these protocols are not traced by μVerum and, as a result, cannot be undone.

Second, the developers are responsible for implementing a PATCH method for each operation they wish to be undone in the future. This is an important aspect of the μVerum approach. Given the heterogeneous architecture of microservice applications, only the developers are aware of what needs to be done locally in the microservice to undo an operation.

Third, operations that should not be recovered by μVerum must not be logged by a μVerum agent. This aspect is important given that some services may not be in control of the application's developers and in that case the recover task needs to be dealt differently. One example of such situation are the cloud storage services, social networks, banking and accountability systems. These operations may require a specific approach for recovery that is not possible to automate with μVerum.

Fourth, the application must use routing servers and a discovery service. The routing servers are used to serialize the user's requests and append meta-data. The discovery service is used to route every operation that needs to be logged to the corresponding μVerum agent.

Fifth, if some operations need to execute atomically or in a specific order then it is the responsibility of the developer to return that information when the PATCH method of an operation is executed. This way μVerum knows how to process these operations without violating the consistency requirements of the application.

7.3.5 Intrusion recovery with μVerum

μVerum is responsible for logging users' requests and microservices operations during normal execution. These are later used to perform recovery. In this section we describe how μVerum logs requests and operations and how the recovery process works.

Normal execution

During normal execution, μVerum routers and μVerum agents log users' requests and microservice operations. Requests are intercepted and logged without interfering with the application. μVerum routers are responsible for users' requests and the μVerum agents are responsible for the microservice operations which are stored in the μVerum logs. For each logged operation μVerum records the following information:

- `request_id`: a unique serial id that was originated when the original request reached the application;
- `start_ts`: a timestamp of the moment the operation was logged;
- `end_ts`: a timestamp of the moment the response of the operation was logged;
- `service`: the address of the service;
- `sender`: the address of the issuer of the request;

- `method`: the HTTP method of the request;
- `operation`: the payload of the HTTP operation.

The `start_ts` and `end_ts` timestamps allow the administrator to reason about when did the operation occur. It also allows μVerum to analyze which operations were executed concurrently. During recovery concurrent operations can be re-executed in parallel and reduce the overall time to recovery. The `service` and `sender` are used to correlate requests in order to create a graph with the services that were invoked by this request. The `method` and `operation` will be used in the recovery process by the PATCH method. For this to work it is necessary that every entry in the log is identified by two ids. The `request_id` has to be unique for each user request that reaches the application. This will ensure serialisation of the requests that is required to maintain consistency during recovery. μVerum delegates the job of generating unique ids to the distributed NoSQL database that is used to store the logs. The `request_id` is the same for every operation that was triggered as a result of the user's request that originate them. This id is propagated from microservice to microservice by the μVerum agent, which adds it to the HTTP header. Every time a μVerum agent receives a request it reads its `request_id` from the HTTP header and saves it to the log. It is worth mentioning that the developers need to forward the `request_id` to the next operation. This is done by appending an extra parameter to the next HTTP with the `request_id` that was received earlier. If a microservice operation fails to do so it is not possible to trace a graph of operations for recovery.

Recovery

Recovery is done when the administrator of the application detects faulty requests and wants to undo their effects from the affected microservices. When the administrator selects faulty requests from the log, μVerum presents him a list of faulty operations that were executed by the microservices and will be reverted. This will give an idea to the administrator of what will be recovered in the application. Once the administrator confirms the recovery, μVerum executes compensating operations (Korth et al., 1990) (one for each operation) that will revert the effects of the attack. The compensating operations are invoked by μVerum using the PATCH verb. Ideally all microservices are running at the moment that the administrator issued the recovery operation but that may not be the case. For some reason a set of the microservices that need to be recovered may be offline. In that case μVerum schedules those recovery actions to be executed as soon as possible.

Algorithm 4 describes the procedure to identify and undo microservices operations given a faulty HTTP request. The algorithm considers a single faulty request for simplicity, as recovering from several is similar. It takes as input req (line 1), the log entry with the malicious HTTP request that the administrator identified. The algorithm works the following way. First it initializes an empty list to store the undo operations that failed (line 2). This will happen if some microservices happen to be offline during the recovery process and μVerum needs to postpone the undo operation. Then it gathers every microservices operation that was caused by req (line 4). With these operations it is possible to trace a graph (line 5). This graph is created using the `service` and `sender` fields of each log entry. Every two operations that have a `service` equal to a `sender` are connected in the graph. Then this graph is presented to the administrator (line 6) so he can confirm (line 7) to undo every operation in the graph. If he confirms then the recovery process starts from the root node of the graph (lines 8 and 9).

The undo function is recursive (lines 11 to 29). It takes as input a node from the graph, in other words, a log entry. This function starts by getting the service address from the log entry (line 12) and executing the PATCH method of this address (line 13). This PATCH function was previously implemented by the developers of the application and it will undo the effects of the execution of the operation from the state of the microservice. If the operation to be undone has consistency requirements, such as, it should be executed in a certain order or it should be executed together with other operations, then the PATCH method will return an invariant. This invariant can be of two forms: *ORDER* (lines 14 to 16) or *ATOMIC* (lines 17 to 19). These special cases will be treated by the appropriate functions presented in algorithms 5 and 6. If there are no invariants, the algorithm proceeds to check if the PATCH method was successfully executed (lines 20 to 22). If it failed to execute the PATCH method then it will be stored in the pending list to be re-executed later. Then, the algorithm fetches the next nodes in the graph, *children*, and for each one of them it recursively invokes the *undo* function (lines 23 to 28). Finally, the list of services that failed to be executed is presented to the administrator (line 29) and the function terminates. This pending list will be used by a background task that runs periodically to complete the recovery operation. In fact, the first time μVerum connects to a service it will execute all the pending operations before forwarding any new requests.

Algorithm 4 Recovery algorithm without ordering or atomic requirements.

```
1:  INPUT req // malicious HTTP request
2:  $pending \leftarrow \perp$
3:  $request\_id \leftarrow req.request\_id$
4:  $log\_entries \leftarrow get\_log\_entries(request\_id)$
5:  $graph \leftarrow trace\_graph(log\_entries)$
6:  $print(graph)$
7:  if $admin\_confirms\_recovery()$ then
8:      $root \leftarrow graph.get\_root()$
9:      $undo(root)$
10: end if

11: $undo(node)$
12: $service \leftarrow node.service$
13: $result \leftarrow execute(service, operation, PATCH)$
14: if $result.invariant == ORDER$ then
15:     RETURN$undo\_ordered(0, [node], PREPARE)$
16: end if
17: if $result.invariant == ATOMIC$ then
18:     RETURN$undo\_atomic(result.atomic\_bag)$
19: end if
20: if $result.STATUS \neq SUCCESS$ then
21:     $pending \leftarrow pending \cup node$
22: end if
23: $children \leftarrow node.children$
24: if $children \neq \perp$ then
25:     for $child \in children$ do
26:         $undo(child)$
27:     end for
28: end if
29: $print(pending)$
```

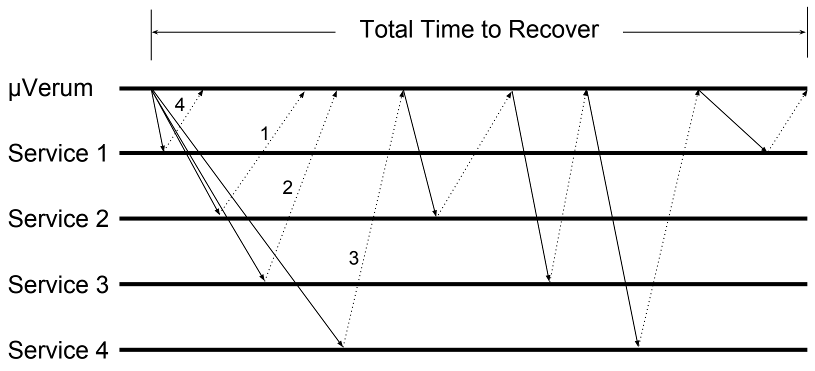

Figure 7.5: Recovering a set of microservices with ordering requirements.

7.3.6 Keeping consistency after recovery

During recovery several compensating operations are executed concurrently with the operations of the users of the application. Besides that, many of these compensating operations will revert the state of the application which, from the point-of-view of the user, may produce an inconsistent state. During recovery the users may observe some incongruent values or see some information vanish from the application. Once the application has recovered, the users should observe a consistent state. Consistency may be enforced if the developers define invariants, conditions that μVerum ensures are not broken during recovery. Formally, we define the two invariants that must be respected by μVerum after recovery as follows:

- *Ordering invariant*: any operation o that takes as predecessor another operation o' is always executed after o' was completely and successfully executed.

- *Atomicity invariant 1*: if an atomic operation o in a set of operations S_a was completely and successfully executed, then every other operation in S_a was also completely and successfully executed.

- *Atomicity invariant 2*: if an atomic operation o in a set of operations S_a fails to be executed, then every other operation in S_a is not executed or rolledback.

Figure 7.5 shows how μVerum executes recovery when there is an ordering invariant. In the figure, in a first round μVerum executes the PATCH method in the services to be recovered and gets, as a result, dependency conditions. Then, in a second round, μVerum will execute the PATCH method with an extra argument confirming that the previous service as already recovered.

Figure 7.6 shows how μVerum deals with atomic operations. In this case it will execute the required operations concurrently, since there are no ordering requirements. It is worth mentioning that in both scenarios μVerum isolates the services, i.e., it stops forwarding users' requests until the recovery is done. Every users' requests are stored in the *pending* list and will be executed once the recovery has finished.

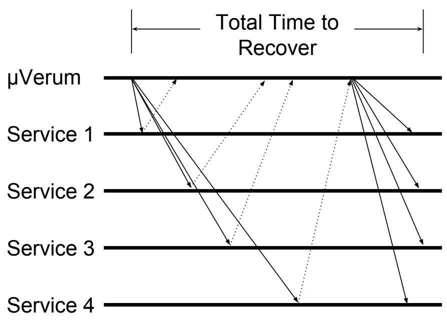

Figure 7.6: Recovering a set of microservices with atomicity requirements.

Enforce order

Some microservices may require a strict order to be executed. The same applies to the recovery operations. An example that shows the importance of ordering is a messaging application. For example, the following messages: "*I just had an accident*" and "*I am OK*" may have opposite readings if they are delivered out of order. Besides ordering the messages it is also necessary to synchronize with another service to deliver notifications for the users. In this case we have two different ordering requirements: first, there is a FIFO ordering of the messages delivery; second, there is a cause-effect order between the message delivery and the notification. A notification should arrive after the message, not before.

To enforce ordering requirements, developers define dependencies between operations. This is done by returning an ordered list of dependencies in the PATCH method, such as: *ordered_queue*$\{ms_1 \rightarrow ms_2\}$ indicating that the service ms_1 must be executed before ms_2. Then μVerum executes the operations in a serialized way following the given order. This has a drawback in terms of performance, since it does not allow μVerum to execute recovery operations concurrently.

Algorithm 5 describes the *undo_ordered* function and how it performs recovery preserving the execution ordering of the requests. This function takes as input: an index value pointing to the current node in the list, *index*, an ordered list with the services that need to be recovered by the correct order, *ordered_queue*, and a flag indicating in which phase the algorithm is, *phase*. First, the algorithm extracts from the list the current service that should be executed (line 2). Then the algorithm follows one of two branches. In the *prepare* branch (lines 3 to 16), μVerum will execute the services in the queue until it reaches the end of the graph. Each service will be executed (line 4) with the parameter *phase* set to *PREPARE*. This is just to fetch every invariant there is and fill the queue with the services. If any of the services fail to execute in the prepare phase (lines 5 to 7) then the recovery is aborted. This happens because if μVerum cannot determine the order in which the services would be executed then it cannot guarantee that the invariants are not violated. If the operation executed successfully then it will fetch the next nodes from the graph (line 8). For each of the next nodes in the graph the function *undo_ordered* will be recursively invoked (lines 9 to 13) until it reaches the end of the graph (lines 13 to 15), then it will make the recursive invocation with the *phase* variable set to *COMMIT*.

In the *commit* (lines 17 to 28), the services are executed by the given order. This is done by isolating the services to users' request (line 18). This does not mean that the requests are discarded, instead they are stored in a list to be executed after the recovery finishes. The algorithm proceeds by iterating through the queue (line 19) and each of the services is executed synchronously (lines 20 and 21), i.e., each service starts execution after the previous one has ended. If any execution fails, then a rollback is issued (line 23) and the recovery process is aborted (line 24). In the end of the function (line 27) the blocked services are resumed.

Algorithm 5 Recovery algorithm with ordering requirements.

```
 1: $undo\_ordered(index, ordered\_queue, phase)$
 2: $service \leftarrow ordered\_queue[index].service$

 3: if $phase == PREPARE$ then
 4:    $result \leftarrow execute(service, PATCH, PREPARE)$
 5:    if $result.STATUS \neq SUCCESS$ then
 6:       $abort()$
 7:    end if
 8:    $children \leftarrow node.children$
 9:    if $children \neq \perp$ then
10:       for $child \in children$ do
11:          $undo\_ordered(child.order, ordered\_queue, PREPARE)$
12:       end for
13:    else
14:       $undo\_ordered(0, ordered\_queue, COMMIT)$
15:    end if
16: end if

17: if $phase == COMMIT$ then
18:    $block\_services(ordered\_queue)$
19:    for $node \in ordered\_queue$ do
20:       $service \leftarrow node.service$
21:       $result \leftarrow execute(service, operation, PATCH)$
22:       if $result.STATUS \neq SUCCESS$ then
23:          $rollback(queue)$
24:          $abort()$
25:       end if
26:    end for
27:    $resume\_services(ordered\_queue)$
28: end if
```

Enforce atomicity

When a developer enforces atomicity of an operation it specifies a list of operations that have atomicity requirements, i.e., either all of them are executed or none of them is. It is assumed that every service in the list knows which services must be executed atomically. If any of the operations fail to execute then the recovery needs to be aborted. An example of such operation is a money transfer. This kind of operation requires a debit and a credit to be executed. If any of the operations fail, then the final balance will be invalid.

Algorithm 6 describes how μVerum performs recovery of services that hold an atomic invariant. Function *undo_atomic* (lines 1 to 6) iterates through the bag with the services. After blocking them from user requests (line 2), it will execute the PATCH method of each service asynchronously (line 5). When a service finishes executing the PATCH method (lines 7 to 14), it is removed from the bag (line 8). When the bag is empty (line 9), μVerum resumes the services that were blocked in the beginning of recovery (line 10). When a service fails to execute the PATCH method (line 12), it is added to the pending list (line 13). When a service reestablishes connection with μVerum, the first operations it executes are those in the pending list. This ensures that no other operation is executed concurrently with the atomic ones.

Algorithm 6 Recovery algorithm with atomic requirements.

```
1: $undo\_atomic(atomic\_bag)$
2: $block\_services(atomic\_bag)$
3: for $node \in atomic\_bag$ do
4:     $service \leftarrow node.service$
5:     $execute\_async(service, operation, PATCH, upon\_service\_result)$
6: end for
7: $upon\_service\_result(service, result)$
8: $atomic\_bag \leftarrow atomic\_bag \sim service$
9: if $atomic\_bag == \perp$ then
10:     $resume\_services()$
11: end if
12: if $result.STATUS \neq SUCCESS$ then
13:     $pending \leftarrow pending \cup service$
14: end if
```

7.4 μVerum Implementation

A μVerum prototype was implemented in Java, since most of the components were also written in or provide a Java API. μVerum Admin is a Spring Boot (Webb et al., 2013) microservice. The μVerum agent is a Java application that uses Little Shoot Proxy (Labs, 2018) to intercept HTTP requests. The μVerum log is a MongoDB (Chodorow, 2013) database. The μVerum router is a Zuul (Netflix, 2018) instance with a special filter that appends the metadata to the requests. The Discovery service is a non-modified Zookeeper (Hunt et al., 2010) instance. We chose Zuul and Zookeeper since they are widely used in the industry, especially for microservice applications.

In our test case μVerum is logging operations of a SockShop (WeaveWorks, 2018) application. We chose SockShop for our implementation for the following reasons: it is open-source, making it possible to use by anyone who wants to extend μVerum; it follows a system architecture similar to the one presented in Section 7.3.3; is has an hybrid architecture with some components working in orchestration and other components working in choreography, making it more similar to a real world microservice applications; it has a considerable scale, being composed by 9 microservices and 6 datastores, which makes it an interesting system to evaluate μVerum.

7.5 Experimental Evaluation

With our experiments we want to answer these questions: (A) is the μVerum approach capable of recovering from intrusions in real world applications? (B) what is the cost, in terms of performance, of using the μVerum agents to log every operation? (C) how long does it take to assess the damage and undo unintended actions?

We performed the experiments using Google Compute Engine (Krishnan and Gonzalez, 2015), which allowed us to deploy each microservice in a single and isolated virtual machine. This way we are able

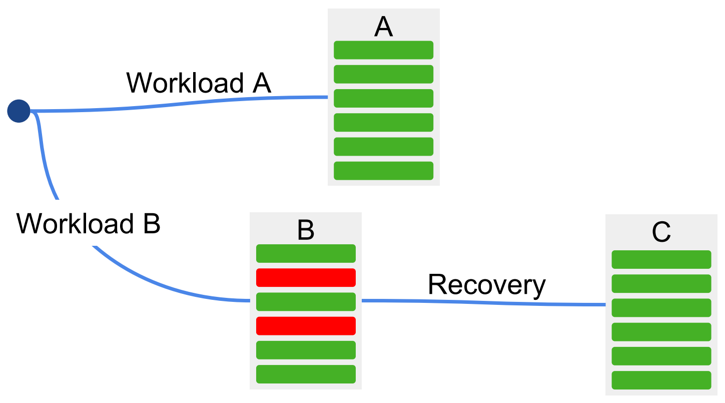

Figure 7.7: Recovery validity of μVerum. From a clean database (initial dot) we execute two different workloads: *A* with only valid operations, and *B* with the same operations as of *A* but with an extra percentage of malicious operations. *B* contains some valid records (green bars) and some corrupted records (red bars). Then recovery led the second instance to a new state *C* that is equivalent to state *A*.

to recreate an environment similar to a real-world application. We picked the *n1-standard-2* flavor for every virtual machine, which provides 2 CPUs with 7.5GB of memory.

7.5.1 Validity of μVerum recovery

μVerum successfully recovers an application if it manages to undo malicious operations from the state of the microservice (database) as if they never occurred. To evaluate this, we wrote a script that allowed us to execute a *diff*-like operation with two databases: one that was recovered by μVerum (database C) and another (database A) that received the exact same operations except the ones we considered intrusions and were undone. In other words, we want to compare a recovered database with one that was never attacked. The script compares all the deterministic values of each database, meaning that non-deterministic values generated by the database itself, such as automatic identifiers and timestamps, are not compared. We are not interested in comparing non-deterministic values since they are outside of the control of the application which cannot be logged and recovered by μVerum, that only logs application level requests.

To create both databases we executed a workload (*workload A*) with 10,000 requests using load-test from SockShop in *database A*. Then we created *workload B*, which was created using *workload A* with an extra percentage of the requests being malicious. Finally, we recovered *database B* and used our script to compare it with *database A*. Figure 7.7 shows a diagram of how these experiments were performed.

We repeated these experiments ten times to recreate different states that allowed us to verify the validity of μVerum recovery. We started by having two identical workloads of 10,000 valid operations. Then, in each experiment we added 10% of malicious requests. In every experiment μVerum was able to achieve a state *C* equal to *A* (except for the non-deterministic values that were not compared). This was not surprising given that the PATCH methods that we implemented were tested beforehand and we validated they were capable of reverting any executed operation.

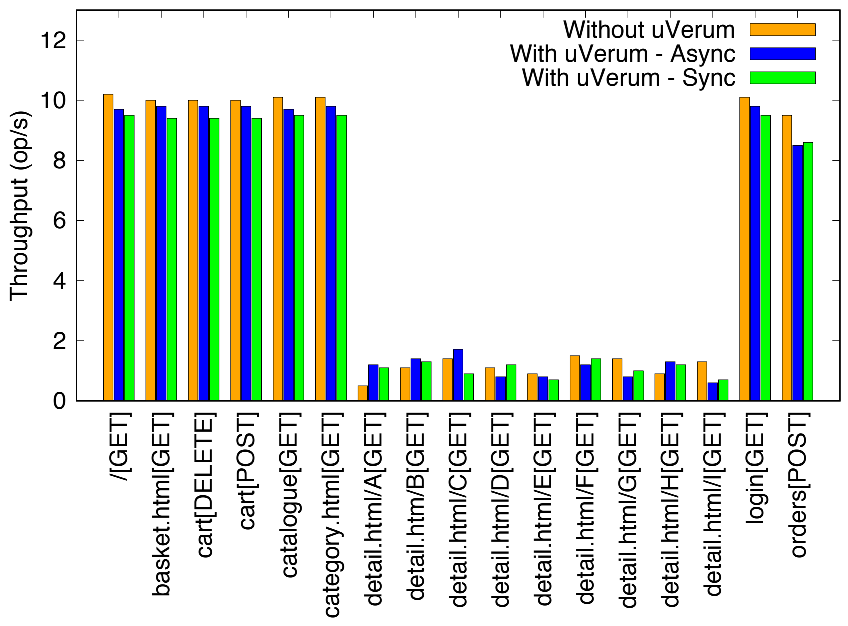

Figure 7.8: Performance overhead of μVerum with in different URLs of the application. The results, expressed in operations per second, refer to the application without μVerum (orange bar), the application with μVerum intercepting the requests asynchronously (blue bar) and with μVerum intercepting the requests synchronously (green bar).

7.5.2 Performance overhead

Having a filter log every operation that reaches a microservices induces an overhead to the performance of the application. To evaluate the performance overhead of μVerum on the application, we performed a series of experiments in which we measure the number of requests per second with and without having μVerum logging the operations. To do so we used the *loadtest* application from SockShop configured to issue 10,000 requests simulating 5 concurrent users. First, we tested with SockShop, then we repeated the same workloads with μVerum logging the requests. We performed using both logging methods of μVerum (asynchronous and synchronous).

Figure 7.8 shows the performance in requests per second of SockShop, with and without μVerum. The overhead of μVerum is around 3.7% for asynchronous logging and 6.4% for synchronous logging. Some endpoints (details.html) reveal lower overheads. This happens because theses endpoints are not being logged and the caching mechanisms of the applications allow the resource to be presented to the user without executing microservices operations.

7.5.3 Total Time to recover

The *total time to recover* (TTTR) consists in the time it takes since the moment the system administrator starts recovery until every PATCH method was successfully executed. To evaluate the TTTR we executed a recovery in order to undo a number of intrusions that varied from 10 to 100. We repeated this process 10 times. We performed these experiments with the two recovery methods of μVerum: with atomic invariants and with order invariants.

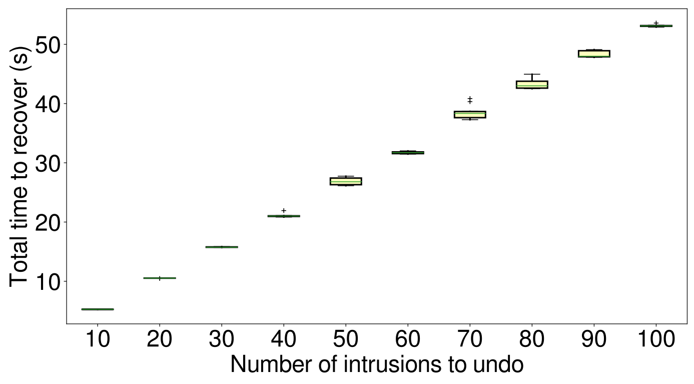

Figure 7.9: Total time to recover with *atomic invariants* changing the number of intrusions to undo.

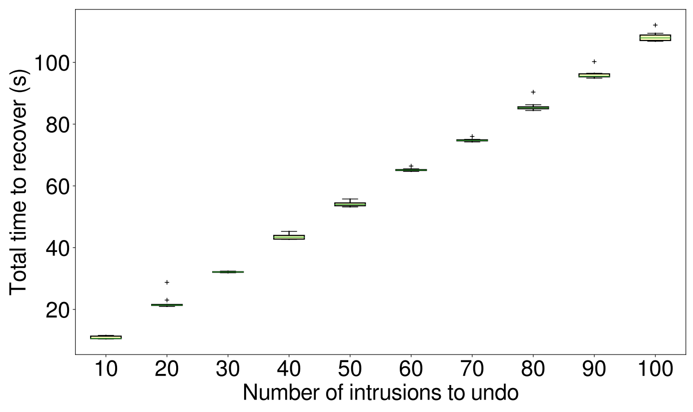

Figure 7.10: Total time to recover with *order invariants* changing the number of intrusions to undo.

Atomic recovery

Figure 7.9 presents the results for the TTTR with atomic invariants. The time to recover increases linearly, varying from 5 seconds for 10 intrusions to around 50 seconds for 100 intrusions. Undoing a single intrusion would take 0.5 seconds.

Ordering recovery

Figure 7.10 present the TTTR from 10 to 100 intrusions using order invariants. The TTTR varies from 11 (for 10 intrusions) to 107 seconds (for 100 intrusions) linearly. We implemented a PATCH method that required 5 operations to be performed in a specific order.

The PATCH method we implemented executes 6 microservices requests that affected 5 services. The TTTR varies depending on two factors: the scale of the application and the complexity of the PATCH method.

In Figure 7.11 we show the time a users' request take to process before, during, and after performing recovery. We repeated these tests varying the number of intrusions from 1 to 10. During these experiments we also executed the same load test described in Section 7.5.2 to simulate a real world application with several concurrent users. The peaks in the graph happen when the administrator issues a recovery process. Users experience a delay in the application for a while, but once recovery finishes, the application resumes normal operation. The latency varied from around 30ms during normal operation to a couple of seconds (from 1 to 5) during recovery.

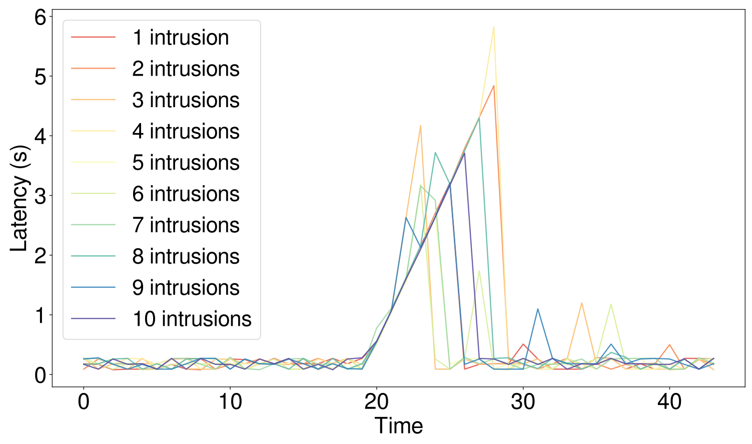

Figure 7.11: Latency of the application before, during and after recovery with *atomic invariants*, changing the number of intrusions.

7.6 Summary

This chapter presents μVerum, an intrusion recovery approach for microservices applications. The design, implementation and evaluation of our proposal shows that it is possible to recover from intrusion in microservices applications. Our results show that it is possible to keep the application available for the user while performing recovery at the expense of a momentary degradation of the performance (from 1 to 5 seconds). Using μVerum imposes an overhead to the application that varies from 3.7% to 6.4%.

8 Conclusions

This dissertation presents novel intrusion recovery mechanisms for the cloud computing model. These mechanisms allow reverting the effects of malicious operations from the system without discarding legitimate data. To recapitulate, the two questions we asked in the beginning of the thesis were:

- Is it possible to implement intrusion recovery mechanisms given all the limitations of the cloud platform?
- And if so, would such mechanisms benefit from the features provided by cloud services?

The answer to the first question is *yes*, as we presented four novel mechanisms specifically designed for the cloud computing model that are capable of recovering from intrusions. Our experiments were all performed using public clouds for files, databases, applications and services. Results show that it is possible to recover from intrusions.

The answer to the second question is also *yes*. RockFS takes advantage of cloud-backed storage services to keep data logs and metadata. NoSQL Undo uses automatic provisioning of storage to store snapshots and operation logs. Rectify uses automatic deployment of applications to instantiate the recovery mechanisms. Finally, μVerum uses virtual networking to redirect and intercept requests to the logs.

8.1 Achievements

For cloud and cloud-of-clouds storage services, we presented RockFS, an intrusion recovery system that is capable of reverting the effects of intrusions that occur on the client site. This allows users to recover files that were lost or corrupted due to malicious activity or by accident. RockFS was designed to be compatible with any network file system that uses the POSIX API. This design allows

RockFS to be used in a variety of single cloud and cloud-of-clouds storage services and does not require any modifications to the source code of the cloud platform, since it completely runs on the client device. Moreover, RockFS provides secret sharing mechanisms that allow users to distribute the access credentials and the keys used to encrypt locally cached files. Making it more difficult for attackers to access the file system and gain access to unauthorized data.

NoSQL Undo provides intrusion recovery for NoSQL databases, a kind of databases widely used in the cloud (Konstantinou et al., 2011). NoSQL Undo takes advantage of the operation logs created and maintained by the database engine to support replication. This approach has two benefits: first, it imposes less overhead given that the operation log needs to be created anyway, and second, this log already provides total ordering of the operations making it possible to maintain consistency after recovery. NoSQL Undo is generic and can be deployed alongside a variety of NoSQL databases without making any modification to the source code of the database, making it possible to deploy on existing cloud database services.

On the PaaS level of the cloud platform, we presented Rectify, an intrusion recovery system for web applications that run on the cloud. Rectify assumes a black-box system model, i.e., it does not matter the exact technology in which the web application was built, as long as it communicates with the users through HTTP and has a SQL database to store its state. Like the previous presented mechanisms, Rectify does not require any modification to the source code of the web application making it possible to be deployed in any PaaS. Unlike similar intrusion recovery mechanisms that adopt taint tracking techniques to mark and trace the damage caused by an intrusion, Rectify uses machine learning combined with a 2-step classification algorithm to correlate the application level operations, i.e. HTTP requests, with the application's state operations, i.e. database operations. Once the system administrator identifies malicious HTTP requests, Rectify can automatically revert their effects from the database.

For microservice applications we presented an intrusion recovery service called μVerum, designed taking into account a typical system architecture of microservice applications (Wang and Tonse, 2018). μVerum combines a set of good development practices with logging and recovery mechanisms to allow developers to support recovery for their services and define consistency requirements. μVerum allows developers to progressively adapt their microservices to support recover. Recovery is done online with a performance penalty, instead of requiring the application to be shut down during recovery.

Table 8.1 recapitulates the intrusion recovery mechanisms as presented in Chapter 3 and adds the novel intrusions recovery mechanisms proposed in this thesis.

8.2 Future Work

There is some planned work that will continue this line of research.

RockFS is capable of recovering intrusions with automatic provision of storage; however, it imposes an extra monetary cost. A combination of compression techniques for the log as well as automatic checkpointing of the state of the file system would reduce the required storage. Such mechanisms

Recovery System	Designed for	Selective re-execution	Compensating operations	Multiversioned	External inconsistencies	Online Recovery
Operator Undo (Brown and Patterson, 2003)	Email systems	X			Compensating operations	No
Backtracking (King and Chen, 2003)	Virtual Machine	X			Not mentioned	No
Bezoar (Oliveira et al., 2008)	Virtual Machine	X			Not mentioned	No
SHELF (Xiong et al., 2009)	Virtual Machine	X			Not mentioned	Yes
EFS (Santry et al., 1999).	File Systems			X	Not mentioned	Yes
S4 (Strunk et al., 2000)	File systems			X	Not mentioned	Yes
RFS (Zhu and Chiueh, 2003)	File Systems		X		Not mentioned	Yes
Taser (Goel et al., 2005b)	File Systems	X			Compensating operations	No
Back to the Future (Hsu et al., 2006)	File Systems	X			Not mentioned	No
Solitude (Jain et al., 2008)	File Systems	X			Compensating operations	No
Retro (Kim et al., 2010)	File systems	X			Compensating operations	No
Amman et al. (Ammann et al., 2002)	Databases		X		Not mentioned	Yes
Ekin et al. (Akkuş and Goel, 2010)	Web Applications		X		Not mentioned	No
Warp (Chandra et al., 2011)	Web Applications	X			Compensating operations	Yes
AIRE (Chandra et al., 2013)	Web Applications		X		Parallel recovery (like Git)	Yes
Shuttle (Nascimento and Correia, 2015)	Web Applications	X	X		Compensating operations	Yes
RockFS (Chapter 4)	Cloud-backed File Systems		X	X	Compensating operations	Yes
NoSQL Undo (Chapter 5)	NoSQL Databases	X	X	X	Compensating operations	Yes
Rectify (Chapter 6)	PaaS Web Applications		X		Compensating operations	Yes
μ**Verum** (Chapter 7)	Microservices		X		Compensating operations	Yes

Table 8.1: Comparison between the developed work and the related work in the literature.

should not be implemented carelessly since having copies of former system's state increases the possibilities of data breaches. It would be necessary to secure the snapshots using cryptography and hashing mechanisms.

For many NoSQL databases, NoSQL Undo is able to use the replication logs for recovery purposes. However, this approach was not thought for SQL databases. We believe that it is possible to use the same replication logs of SQL databases for recovery. However, SQL databases have additional requirements for consistency related to ACID transactions. Another feature present in most SQL databases are stored procedures that allow the automatization of certain database statements. A SQL Undo solution would need to take into account the executed transactions in the log as well as the stored procedures to ensure data consistency after recovery.

Currently Rectify supports web applications that solely interact with the user through HTTP requests. Although this model represents the majority of the web applications, in recent years we have witnessed a growth in the use of alternative technologies that allow web applications to interact with users. Such interactions cannot be intercepted and recovered by Rectify. Another difference is that many web frameworks use a JSON format to encode data exchanged between the application and the user. This encoding format is not currently supported by Rectify. Extending Rectify to be compatible with these technologies would make it support a wider range of web applications and frameworks. There is work already underway following this approach.

μVerum was designed taking into account microservices applications that interact through an orchestration approach, in which each service triggers other services' operations. An alternative interaction consists in adopting a choreography approach, in which operations are scheduled in a message queue and the corresponding microservices subscribe to those operations. It would be interesting to extend μVerum to be able to recover microservice applications that interact using a choreography approach. This would require some changes to the system architecture of μVerum and to the recovery algorithms. With such improvements, μVerum would be compatible with more complex application that use choreography or a hybrid approach that combines both techniques of interaction.

Bibliography

Abu-Libdeh, H., Princehouse, L., and Weatherspoon, H. (2010). RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 229–240.

Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. (2007). A five-year study of file-system metadata. *ACM Transactions on Storage*, 3(3):9.

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

Amazon, E. (2015). Amazon web services. *Available in: http://aws. amazon.com/es/ec2/(November 2015)*.

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

Apache (2018). Apache JClouds. `http://jclouds.apache.org`. Last checked on Nov 05, 2018.

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

Barrett, D. J. (2008). *MediaWiki. Wikipedia and Beyond*. O'Reilly Media.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency control and recovery in database systems.

Bessani, A. N., Alchieri, E. P., Correia, M., and Fraga, J. S. (2008). DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 163–176.

Bessani, A. N., Correia, M., Quaresma, B., André, F., and Sousa, P. (2011). DepSky: dependable and secure storage in a cloud-of-clouds. *EuroSys'11 Proceedings of the 6th Conference on Computer Systems*, pages 31–46.

Bessani, A. N., Mendes, R., Oliveira, T., Neves, N., Correia, M., Pasin, M., and Verissimo, P. (2014). SCFS: A shared cloud-backed file system. In *Proceedings of USENIX Annual Technical Conference*, pages 169–180.

Bessani, A. N., Santos, M., Felix, J., Neves, N. F., and Correia, M. (2013). On the efficiency of durable state machine replication. In *Proceedings of the USENIX Annual Technical Conference*, pages 169–180.

Bhardwaj, S., Jain, L., and Jain, S. (2010). Cloud computing: A study of infrastructure as a service (IaaS). *International Journal of Engineering and Information Technology*, 2(1):60–63.

Bishop, M. (2003). *Computer Security: Art and Science*. Addison-Wesley.

Blakley, G. R. (1979). Safeguarding cryptographic keys. In *Proceedings of the AFIPS National Computer Conference*, volume 48, pages 313–317.

Brazell, A. (2011). *WordPress Bible*. John Wiley and Sons.

Brewer, E. A. (2000). Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, July*, volume 7.

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

Brown, A. and Patterson, A. A. (2001). To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System Dependability (EASY'01)*.

Brown, A. and Patterson, D. (2003). Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14.

Brown, A. B. and Patterson, D. A. (2002). Rewind, repair, replay: three r's to dependability. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 70–77.

Brown, M. (2012). *Getting Started with Couchbase Server*. O'Reilly.

Burihabwa, D., Pontes, R., Felber, P., Maia, F., Mercier, H., Oliveira, R., Paulo, J., and Schiavoni, V. (2016). On the cost of safe storage for public clouds: an experimental evaluation. In *Proceedings of the 35th IEEE Symposium on Reliable Distributed Systems*, pages 157–166.

Cachin, C. and Tessaro, S. (2006). Optimal resilience for erasure-coded byzantine distributed storage. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 115–124.

Calder, B. et al. (2011). Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27.

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

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

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4.

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

Chodorow, K. (2013). *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc.

Ciurana, E. (2009). *Developing with Google App Engine*. APress.

Cloud Security Alliance (2012). Security guidance for critical areas of mobile computing.

Codd, E. F. (1969). Derivability, redundancy, and consistency of relations stored in large data banks. Technical report, IBM Research Report.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154.

Correia, M. (2013). Clouds-of-clouds for dependability and security: geo-replication meets the cloud. In *European Conference on Parallel Processing*, pages 95–104. Springer.

Costa, P., Ramos, F., and Correia, M. (2017). On the design of resilient multicloud MapReduce. *IEEE Cloud Computing*, 4(4):74–82.

Date, C. J. and Darwen, H. (1997). *A guide to SQL standard*, volume 3. Addison-Wesley Reading.

Davis, A. H., Sun, C., and Lu, J. (2002). Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, pages 58–67.

De Vos, A. (2011). *Reversible computing: fundamentals, quantum computing, and applications*. John Wiley & Sons.

Debar, H., Dacier, M., and Wespi, A. (1999). Towards a taxonomy of intrusion detection systems. *Computer Networks*, 31(8):805–822.

Denning, D. E. and Neumann, P. G. (1985). Requirements and model for IDES - a real-time intrusion detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA.

Dobre, D., Viotti, P., and Vukolić, M. (2014). Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14.

Downing-Troy, B. (1998). *Java RMI: remote method invocation*. IDG.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer.

Dusseault, L. and Snell, J. (2010). Patch method for HTTP. Technical report.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408.

Enriquez, P., Brown, A., and Patterson, D. A. (2002). Lessons from the PSTN for dependable computing. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*.

European Union Agency for Network and Information Security (2014). *Algorithms, Key Size and Parameters Report*. ENISA.

Feinberg, A. (2011). Project voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*.

Filebench (2017). Filebench. *https://github.com/filebench/filebench*.

Frank, M. P. (2005). Introduction to reversible computing: motivation, progress, and challenges. In *Proceedings of the 2nd ACM Conference on Computing Frontiers*, pages 385–390.

Gallmeister, B. (1995). *POSIX. 4 Programmers Guide: Programming for the real world*. O'Reilly.

Gasser, M. (1988). *Building a Secure Computer System*. Van Nostrand Reinhold.

Geelan, J. et al. (2009). Twenty-one experts define cloud computing. *Cloud Computing Journal*, 4:1-5.

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

Goel, A., Feng, W.-C., Maier, D., and Walpole, J. (2005a). Forensix: A robust, high-performance reconstruction system. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 155–162.

Goel, A., Po, K., Farhadi, K., Li, Z., and De Lara, E. (2005b). The Taser intrusion recovery system. In *ACM SIGOPS Operating Systems Review*, volume 39 (5), pages 163–176.

Goodson, R. R., Wylie, J., Ganger, G. R., and Reiter, M. K. (2004). Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 135–144.

Halalai, R., Felber, P., Kermarrec, A. M., and Taïani, F. (2017). Agar: A caching system for erasure-coded data. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems*, pages 23–33.

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

Halfond, W. G. J., Orso, A., and Manolios, P. (2006b). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185.

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

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

Hayes, B. (2008). Cloud computing. *Communications of the ACM*, 51(7):9–11.

Hendricks, J., Ganger, G. R., and Reiter, M. K. (2007). Low-overhead Byzantine fault-tolerant storage. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 73–86.

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

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81.

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

Hu, Y. and Panda, B. (2003). Identification of malicious transactions in database systems. In *Proceedings of the 7th International Database Engineering and Applications Symposium*, pages 329–335.

Hu, Y. and Panda, B. (2004). A data mining approach for database intrusion detection. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 711–716.

Hunt, H. W. and Szymanski, T. G. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*.

Info Security (2018). 2017: Worst Year Ever for Data Loss and Breaches. `https://www.infosecurity-magazine.com/news/2017-worst-year-ever-for-data-loss/`. Last checked on Nov 05, 2018.

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

Ioannidis, S., Keromytis, A. D., Bellovin, S. M., and Smith, J. M. (2000). Implementing a distributed firewall. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 190–199. ACM.

Istio (2018). Istio. https://istio.io.

Jain, S., Shafique, F., Djeric, V., and Goel, A. (2008). Application-level isolation and recovery with solitude. In *ACM SIGOPS Operating Systems Review*, volume 42 (4), pages 95–107.

Kamara, S. and Lauter, K. (2010). Cryptographic cloud storage. In *International Conference on Financial Cryptography and Data Security*, pages 136–149. Springer.

Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., and Kirda, E. (2015). Cutting the Gordian knot: A look under the hood of ransomware attacks. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24.

Khetrapal, A. and Ganesh, V. (2006). Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, pages 22–28.

Kim, T., Wang, X., Zeldovich, N., and Kaashoek, M. F. (2010). Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 89-104.

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

Knorr, E. and Gruman, G. (2008). What cloud computing really means. *InfoWorld*, 7.

Kononenko, O., Baysal, O., Holmes, R., and Godfrey, M. W. (2014). Mining modern repositories with elasticsearch. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 328–331. ACM.

Konstantinou, I., Angelou, E., Boumpouka, C., Tsoumakos, D., and Koziris, N. (2011). On the elasticity of NoSQL databases over cloud management platforms. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 2385-2388.

Korth, H. F., Levy, E., and Silberschatz, A. (1990). A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 95–106.

Kotsiantis, S. B. (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*, pages 3–24.

Krishnan, S. P. T. and Gonzalez, J. L. U. (2015). Google compute engine. In *Building Your Next Big Thing with Google Cloud Platform*, pages 53–81. Springer.

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

Labs, I. (2018). Little shoot proxy. https://github.com/adamfisk/LittleProxy.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

Lee, S., Low, W., and Wong, P. (2002). Learning fingerprints for a database intrusion detection system. In *Computer Security – ESORICS*, pages 264–279. Springer.

Li, Y. and Manoharan, S. (2013). A performance comparison of SQL and NoSQL databases. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 15–19.

LimeSurvey (2017). An open source survey tool. `https://www.limesurvey.org`. Last checked on Feb 02, 2017.

Linkerd (2018). Linkerd. https://linkerd.io.

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

Lomet, D. B. (1992). *MLR: A recovery method for multi-level systems*, volume 21. ACM.

Ma, D. and Tsudik, G. (2007). Forward-secure sequential aggregate authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 86–91.

Ma, D. and Tsudik, G. (2009). A new approach to secure logging. *ACM Transactions on Storage*, 5(1).

Mather, T., Kumaraswamy, S., and Latif, S. (2009). *Cloud security and privacy: an enterprise perspective on risks and compliance*. O'Reilly.

Matos, D. R. and Correia, M. (2016). NoSQL Undo: Recovering NoSQL databases by undoing operations. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*.

Matos, D. R., Pardal, M. L., and Correia, M. (2017). Rectify: Black-box intrusion recovery in PaaS clouds. In *Proceedings of the 2017 ACM/IFIP/USENIX International Middleware Conference*.

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

Mauro, T. (2018). Adopting microservices at Netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

Mcfredries, P. (2008). Technically speaking: The cloud is the computer. *IEEE Spectrum*, 45(8):20–20.

Mell, P. and Grance, T. (2011). The NIST definition of cloud computing. *National Institute of Standards and Technology*.

Mohurle, S. and Patil, M. (2017). A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5).

MongoDB, I. (2016). MongoDB Manual. `https://docs.mongodb.org/manual/`. Last checked on Feb 02, 2016.

MongoDB, I. (Nov. 2018). MongoDB. `http://www.mongodb.org`. Last checked on Nov 05, 2018.

Morita, K. (2008). Reversible computing and cellular automata—a survey. *Theoretical Computer Science*, 395(1):101–131.

Nascimento, D. and Correia, M. (2015). Shuttle: Intrusion recovery for PaaS. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, pages 653–663.

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

NetFlix (2018). Eureka. https://github.com/Netflix/eureka.

Netflix (2018). Zuul. https://github.com/Netflix/zuul.

Newcomer, E. and Lomow, G. (2005). *Understanding SOA with Web services*. Addison-Wesley.

Newman, S. (2015). *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc.

Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. (2005). Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 295–307.

Oliveira, D., Crandall, J., Wassermann, G., Wu, S. F., Su, Z., and Chong, F. (2006). ExecRecorder: VM-Based Full-System Replay for Attack Analysis and System Recovery. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability.*, page 381-390.

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

Olson, M. A., Bostic, K., and Seltzer, M. I. (1999). Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191.

O'Meara, C. (2018). Reactive Kafka microservice template. https://github.com/omearac/reactive-kafka-microservice-template.

Oppenheimer, A., Ganapathi, A., and Patterson, A. A. (2003). Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, volume 67.

OWASP (2014). Testing for NoSQL injection.
https://www.owasp.org/index.php/Testing_for_NoSQL_injection.

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

Papagiannis, I., Migliavacca, M., and Pietzuch, P. (2011). PHP Aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development*, page 13.

Peinl, R., Holzschuher, F., and Pfitzer, F. (2016). Docker cluster management for the cloud – survey results and own solution. *Journal of Grid Computing*, pages 1–18.

Pereira, S., Alves, A., Santos, N., and Chaves, R. (2016). Storekeeper: A security-enhanced cloud storage aggregation service. In *Proceedings of the 35th Symposium on Reliable Distributed Systems*.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. O'Reilly Media, Inc.

Rizun, R. (2018). S3FS - FUSE-based file system backed by Amazon S3. `http://code.google.com/p/s3fs/`. Last checked on Nov 05, 2018.

Robertson, W., Vigna, G., Kruegel, C., and Kemmerer, R. (2006). Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13th Network and Distributed System Security Symposium*.

Robinson, D. (2008). *Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services enables you to reach business goals faster*. Emereo Pty Ltd.

Roesch, M. (1999). Snort: Lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*, pages 229–238.

Safronov, M. and Winesett, J. (2014). *Web Application Development with Yii 2 and PHP*. Packt Publishing Ltd.

Sandhu, R. S. and Samarati, P. (1994). Access control: principle and practice. *IEEE Communication Magazine*, 32(9):40–48.

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

Scambray, J., Lui, V., and Sima, C. (2011). *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions*. Mc Graw Hill.

Schoenmakers, B. (1999). A simple publicly verifiable secret sharing scheme and its application to electronic voting. *Proceedings of the 19th International Cryptology Conference*, pages 148–164.

Shamir, A. (1979). How to share a secret. *Communications of ACM*, 22(11):612–613.

Shirvanian, M., Jareckiy, S., Krawczykz, H., and Saxena, N. (2017). Sphinx: A password store that perfectly hides passwords from itself. In *IEEE 37th International Conference on Distributed Computing Systems*, pages 1094–1104.

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

Slamanig, D. and Hanser, C. (2012). On cloud storage and the cloud of clouds approach. In *International Conference for Internet Technology and Secured Transactions*, pages 649–655. IEEE.

Snell, J. and Hoffman, P. (2014). JSON merge patch. Technical report.

Stanton, J. M., Stam, K. R., Mastrangelo, P., and Jolton, J. (2005). Analysis of end user security behaviors. *Computers & Security*, 24(2):124–133.

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

Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 59–68.

Sun, D. and Sun, C. (2009). Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470.

Sun, D., Xia, A., Sun, C., and Chen, D. (2004). Operational transformation for collaborative word processing. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 437–446.

Taibi, D. and Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62.

Takabi, H., Joshi, J. B. D., and Ahn, G. (2010). Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31.

Thönes, J. (2015). Microservices. *IEEE software*, 32(1):116–116.

Toffoli, T. (1980). Reversible computing. In *International Colloquium on Automata, Languages, and Programming*, pages 632–644. Springer.

Torvalds, L. and Hamano, J. (2010). Git: Fast version control system. *URL http://git-scm.com*.

Turner, M., Budgen, D., and Brereton, P. (2003). Turning software into a service. *Computer*, 36(10):38–44.

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

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

Varia, J. and Mathew, S. (2014). Overview of Amazon Web Services. *Amazon Web Services*.

Varonis (Nov. 2018). 60 Must-Know Cybersecurity Statistics for 2018. `https://www.varonis.com/blog/cybersecurity-statistics/`. Last checked on Nov 05, 2018.

Vora, M. (2011). Hadoop-HBase for large-scale data. In *Proceedings of the IEEE International Conference on Computer Science and Network Technology*, pages 601–605.

Vrable, M., Savage, S., and Voelker, G. M. (2012). BlueSky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*.

Wang, A. and Tonse, S. (2018). Announcing ribbon: Tying the netflix mid-tier services together. https://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html.

WeaveWorks (2018). Sockshop: A microservices demo application. https://www.weave.works/blog/sock-shop-microservices-demo-application.

Webb, P., Syer, D., Long, J., Nicoll, S., Winch, R., Wilkinson, A., Overdijk, M., Dupuis, C., and Deleuze, S. (2013). Spring boot reference guide. *Part IV. Spring Boot features*, 24.

White, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly.

Wool, A. (2004). A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67.

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

Yalew, S. D., Maguire Jr., G. Q., Haridi, S., and Correia, M. (2017). Hail to the thief: Protecting data from mobile ransomware with ransomSafeDroid. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*.

Zhao, R., Yue, Tak, B., and Tang, C. (2015). SafeSky: a secure cloud storage middleware for end-user applications. In *Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30.

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