# Recovery from Security Intrusions in Cloud Computing

Dário Nascimento
dario.nascimento@tecnico.ulisboa.pt

Instituto Superior Técnico
Advisor: Professor Miguel Pupo Correia

**Abstract.** We introduce a novel service for Platform as a Service (PaaS) systems that gives system administrators the power to recover their applications from intrusions. The motivation of this work is the increasing number of intrusions and critical applications in ~~Cloud, particularly in the emergence of~~ the cloud and the emergence of the PaaS model and the uma coisa sao as intrusoes outra e a emergencia. We introduce a new service where security intrusions in PaaS applications are removed and tolerated. The proposed architecture will support the removal of intrusions due to software flaws, corrupted user requests and support system corrective and preventive maintenance. This document surveys the existing intrusion recovery techniques, identifies their limitations and proposes an architecture to build a new service for intrusion recovery in PaaS. An evaluation methodology is proposed in order to validate the proposed architecture.

## 1 Introduction

Platform as a Service (PaaS) is a cloud computing model that supports automated configuration and deployment of applications. While the Infrastructure as a Service (IaaS) model is being much used to ~~obtaining~~ obtain computation resources and services on demand [**?**,**?**], PaaS ~~intents~~ aims to reduce the cost of software deployment and maintenance. This model defines an environment for execution and deployment of applications in *containers*. Containers provide software stacks, e.g., Python + MySQL, where developers can deploy their applications. Containers can be bare-metal machines, virtual machines or based on processes and resources isolation mechanisms such as Linux cgroups [**?**]. A PaaS system provides a set of programming and middleware services to support application design, implementation and maintenance. Examples of these services are load-balancing, automatic server configuration and storage APIs. These services ~~intent~~ aim to give a well tested and integrated development and deployment framework [**?**]. PaaS systems are provided either by cloud providers [**?**,**?**,**?**,**?**] or open source projects [**?**,**?**,**?**]. ~~The~~ Basides natural metrics such as cost and performance, the success of PaaS systems will also be established by the qualities of their services: correctness, security, consistency and recovery.

The number of critical and complex applications in the cloud, particularly using PaaS, is increasing rapidly. Most of the customers' and companies' critical applications and valuable informations are migrating to cloud environments. Consequently, the value of the deployed applications is superior. As a result, the risk of intrusion is higher because the exploitation of vulnerabilities is more attractive and profitable. An intrusion happens when an attacker exploits a vulnerability successfully. Intrusions can be considered as faults. Faults may cause system failure and, consequently, application downtime ~~. Downtime~~ which has significant business losses [**?**]. Recovery services are needed to remove the intrusion and restore the

correct application state.

Prevention and detection of malicious activities are the priorities of a substantial number of security processes. However, preventing vulnerabilities by design is not enough because software tends to have flaws due to complexity and budget/time constraints [**?**]. More, attackers can spend years developing new ingenious and unanticipated attack methods having access to what protects the application. On the opposite side, guardians have to predict new methods to mitigate vulnerabilities and to solve attacks in few minutes to prevent intrusions. On the other hand, a vast number of redundancy mechanisms and Byzantine fault tolerance protocols are designed for random faults but intrusions are intentional malicious faults designed by human attackers to leverage protocols and systems vulnerabilities. Most of these techniques do not prevent application level attacks or usage mistakes. If attackers use a valid user request, e.g., stealing his credentials, the replication mechanism will spread the corrupted data. Therefore, the application integrity can be compromised and the intrusion reaches its goal bringing the system down to repair.

The approach followed in this work consists in recovering the state of applications when intrusions happen, instead of trying to prevent them from happening. Intrusion recovery does not aim to substitute prevention but to be an additional security mechanism. Similarly to fault tolerance, intrusion recovery accepts that faults occur and have to be processed. Data backup solutions roll back intrusion effects but require extensive administrator effort to replay legitimate actions. Our goal is to design, implement and evaluate *Shuttle* [1], an intrusion recovery service for PaaS systems. Shuttle recovers from intrusions in the software domain due to software flaws, corrupted requests, input mistakes, corrupted data and suspicious intrusions in PaaS containers. Shuttle also supports corrective and preventive maintenance of PaaS ~~'~~ applications. Shuttle aims to recover the integrity of applications without compromising their availability. Shuttle cannot avoid information leaks, so confidentiality is out of the scope of this work. ~~Recover~~ Normally, recovering from intrusions requires extensive human intervention to remove the intrusion and restore the application state. We believe that a service that removes the intrusion effects and restores the applications integrity, without exposing a downtime during the process, is a significant asset to the administrators of PaaS applications.

The rapid and continuous decline in computation and storage costs in cloud platforms makes affordable to store user requests, to use database checkpoints and to replay previous user requests. We will use these mechanisms to recover from intrusions. Despite the time and computation demand to replay the user requests, cloud pricing models provide the same cost for 1000 machines during 1 hour than 1 machine during 1000 hours [**?**]. Shuttle leverage the resources elasticity in PaaS environments, and their capability to scale horizontally, to record and to replay ~~non-intrusive~~ non-malicious user requests. It uses clean PaaS container images to renew the application containers, removing the corrupted state in the image, and replays user requests in parallel. This mechanism recovers the application integrity recreating an intrusion-free state. More, the container image may include software updates to fix previous flaws. Shuttle only requires a valid input record and an intrusion-free container to recover the applications state integrity, i.e., a state that allows the applications to behave

[1] Shuttle stands for the traveling mean between present and previous application states. (Shuttle significa o meio de transporte usado para viajar entre o passado e o presente da aplicação - por causa de ser carregar o snapshot (ir ao passado) e voltar ao presente no redo... como explicar?! :S)

according to their specification. Shuttle will ~~provide intrusion recovery by design of a new service for PaaS systems. This service will~~ be available for developers of PaaS applications usage without installation, configuration. In addition, their applications source code will remain identical~~.~~, or remain unmodified as much as possible.

In this work, we propose~~the~~ , to the best of our knowledge, the first intrusion recovery service for PaaS applications. We also are amongst the first to consider ~~recover in distributed storage applications . We are the~~ recovery in distributed database applications where the data may be stored in multiple database servers. We suggest, to the best of our knowledge, the first intrusion recovery system ~~which~~ using request replay that take into consideration applications running in various containers. We incorporate the possibility of ~~renew~~ renewing the containers image to remove intrusions. Moreover, only few works have been presented that accomplish intrusion recovery without service downtime.

The remainder of the document is structured as follows. In Section 2 explains the goals and expected results of our work. Section 3 presents the fundamental concepts and previous intrusion recovery proposals. Section **??** describes briefly the architecture of PaaS frameworks and the proposed architecture for intrusion recovery service. Section 4 defines the methodology which will be followed in order to validate the proposed service. Finally, Section 5 presents the schedule of future work and Section 6 concludes the document.

## 2 Goals

This work addresses the problem of providing an intrusion recovery system for applications deployed in Platform as a Service clouds. Our overall goal is to *make PaaS applications operational despite intrusions*. More precisely, our service aims to help the administrators to recover from the following ~~intrusions~~problems:

- *Software flaws:* Computing or database containers have been compromised due to software vulnerabilities.
- *Malicious or ~~accidental~~ accidentally corrupted requests:* ~~Accidental~~ Accidentally or malicious user, attacker or administrator requests that perform undesired operations and corrupt the application data.
- *Unknown intrusions in PaaS containers:* The ~~concrete intrusion occurrence has not been detected in the PaaS container but the container is suspected to be compromised.~~ container may has been compromised but the intrusion occurrence was not detected. help.....

Shuttle *supports software updates* to prevent future intrusions and allows operators to try new configurations or software versions without effects in the application behavior perceived by users.

In order to achieve these goals, Shuttle shall meet the following requirements:

- *Remove intrusion effects:* Remove corrupted data in operating system, database and application level instances and update affected legitimate actions. usando o sistema de renovar os containers, remove do OS tambem...
- *Remove selected malicious requests:* Help administrator to track the intrusion producing the set of actions affected by an externally provided list of malicious actions.

– *Support software update:* After recovery, the application state has to be compliant with the new version of the software.
– *Recover without stopping the application:* Recover the application without exposing users to application downtime.
– *Determinism:* Despite parallel execution of requests, the results of re-execution are the same as the result of original execution if the application source code and requests remain equal.
– *Low runtime overhead:* The recording of operations or state for recovery purposes should have a negligible impact in the runtime performance.
– *NoSQL database snapshot:* NoSQ databases will have to be extended to support database snapshots, in order to reduce the recovery time.
– *PaaS integration:* The source code of the application shall remains unmodified as much as possible. PaaS developers do not need to install or configure Shuttle. Shuttle is built in a generic manner and it is reused in each deployed application.

## 3 Related Work

In this section, we give background information on relevant concepts and techniques that are touched in our work. We start with a taxonomy of dependability in Section 3.1. This includes a discussion of intrusions and methods to avoid or tolerate them. Section 3.2 introduces the main intrusion recovery techniques. Each of the following sections describe a number of relevant proposals for ~~recover~~ recovery in the levels where PaaS applications are attacked: operating system, database and application. Finally, Section 3.5 discusses the contributions of these works in the context of this work.

### 3.1 Dependability Concepts

The *dependability* of a computing system is the ability to deliver a ~~trusted service [?]~~ trustworthy service [?] . In particular, the concept of dependability encompasses the following attributes: *availability*: service readiness for authorized users; *confidentiality*: absence of unauthorized disclosure of information; *safety*: absence of catastrophic failures; *reliability:* continuity of correct service; *integrity* absence of improper system state. Three core concepts in dependability are: fault, error and failure.

A *fault* is the cause of an error. The source of a fault belongs to the software or hardware domain and it can be ~~introduce~~ introduced either *accidentally* or *maliciously* during the system development, production or operation phases [?,?]. Faults can deviate the system from its specified behavior leading to errors (Fig. 1). *Errors* are ~~inconsistent parts of system~~the part of the system state that may cause a subsequent failure. definicao continua errada?. A system *failure* occurs when errors become observable at the system interface. In the context of this work, we consider faults which are generated by humans in an accidentally or deliberate, malicious or non-malicious way. In particular, we target software flaws faults and interaction faults from input mistakes, attacks and intrusions [?].

An *intrusion* is a malicious fault resulting from an intentional vulnerability exploitation. As originally proposed generically for faults [?,?], intrusions can be omissive, suspending a system component, and/or assertive, changing a component to deliver a service with a not

specified format or meaning. In order to develop dependable system, delivering a resilient service, we can use a combination of intrusion forecast, prevention, detection, mitigation, tolerance and recovery ~~,~~(Fig. 1).

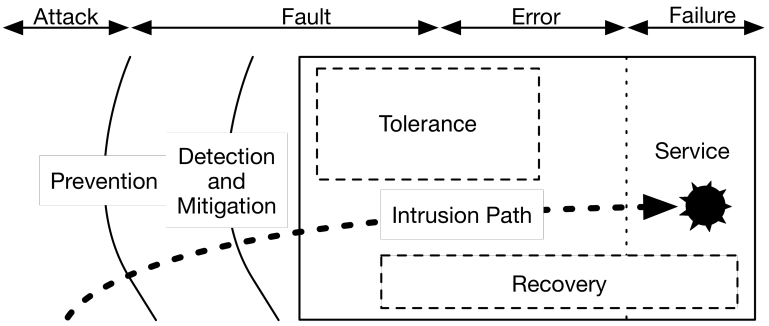


Fig. 1: Intrusion path across the system

*Intrusion forecast and prevention* are realized by design and they seek to prevent future attackers from exploiting vulnerabilities. However, preventing intrusions by design is hard. Software has flaws due to its complexity and budget/time constraints [**?**,**?**]. Also system administrators, as humans, can make security configuration mistakes or users may grant access to attackers [**?**]. Moreover attackers can spend years developing new ingenious and unanticipated intrusions having access to what protects the system while the guardians have to predict them. Due to this asymmetry, it is arguably impossible to protect all vulnerabilities by design. Therefore the vulnerabilities of prevention mechanisms can be exploited successfully leading to an intrusion. A vast number of vulnerabilities and attacks are listed for instance in the National Vulnerability Database [**?**] .

*Intrusion detection and mitigation* mechanisms ~~[?]~~ monitor the system to detect suspicious actions that may be connected with an intrusion. However, intrusion detection systems turns the system attack-aware but not attack-resilient~~.~~, that is, it cannot maintain the integrity and availability of the application in face of attacks [**?**] . Intrusion detection systems may not detect unknown vulnerabilities. More, attackers may use encrypted actions or use legitimate requests.

*Intrusion tolerance* is the last line of defense against attacks before the system failure (Fig. 1). Intrusion tolerance is the ability of a system to keep providing a, possibly degraded but adequate, service during and after an intrusion [**?**]. Instead of trying to prevent every single intrusion, these are allowed, but tolerated because the system has mechanisms that prevent the intrusion from ~~generate~~ generating a system failure [**?**]. Intrusion tolerance ~~hides~~ mechanisms hide the errors effects using redundancy mechanisms or ~~detecting, processing and recovering faults.~~ using intrusion recovery systems, which detect, process and recover from intrusions. sim, considero que seja a ultima linha de defesa antes do sistema falhar, e que pode recorrer ao IRS para mascarar ou se o sistema falhar ai tem mesmo de ser recu-

perado.

Many intrusion tolerance mechanisms are based on replicating state with Byzantine fault-tolerant protocols [**?**,**?**,**?**]. The idea is to ensure that a service remains operational and correct as long as no more than a certain number of replicas are compromised. In *disaster recovery* solutions, the state is replicated in remote sites, allowing the recovery from catastrophes that destroy a site [**?**]. Replicas can be implemented in different manners to provide diversity, to reduce the risk of common failure modes. In addition, a proactive recovery mechanism can reboot the replicas periodically to rejuvenate and restore their soft-state and security assets, e.g., keys ~~[?,?] .[?]~~ [?,?,?] .

The above-mentioned intrusion tolerance replication mechanisms can tolerate some intrusions targeted at design faults (vulnerabilities), as long as diversity is used. Most of the fault tolerance mechanisms do not prevent ~~accidental~~ accidentally or malicious faults at application level, e.g., using valid user requests. State replication mechanisms facilitate the damage to spread from one site to many sites as they copy data without ~~distinguish~~ distinguishing between legitimate and malicious sources. Furthermore, most ~~of~~ proactive recovery techniques rejuvenate only the application soft-state but intrusions can affect its persistent state. Consequently, intrusions may transverse these intrusion tolerance mechanisms and cause a system failure.

### 3.2 Intrusion Recovery

The main focus of this work are intrusion recovery mechanisms which accept intrusions but detect, process and recover from their effects. These mechanisms remove all actions related to the attack, their effects on legitimate actions and return the application to a correct state. These mechanisms can be used to tolerate intrusions or to recover from system failures. In order to recover from intrusions and restore a consistent application behavior,the system administrator detects the intrusion, manages the exploited vulnerabilities and removes the intrusion effects. This process should change the application state to an intrusion-free state.

The first phase of intrusion recovery, out of the scope of this work, concerns the intrusion detection. Automated intrusion detection systems (IDS) are used to detect intrusions or suspicious behaviors. This phase may need human intervention to prevent false positives, which trigger recovery mechanisms and can result in legitimate data losses. Thus the detection phase can have a significant delay. The detection delay should be minimized because intrusion effects spread in the meantime between intrusion achievement and detection. More, intrusion recovery services should provide tools to help administrators to review the application behavior and determine which weaknesses were exploited.

The second phase, also out of the scope of this work, is vulnerability management. Vulnerabilities are identified, classified and mitigated ~~[?]~~ after their detection by a group of persons which the NIST names as the patch and vulnerability group. Vulnerabilities are fixed by configuration adjustments or applying a security software patch, i.e., by inserting a piece of code developed to address a specific problem in an existing piece of software.

The third phase, and the one that this work is about, consists in removing the intrusion effects. Intrusions affect the application integrity, confidentiality ~~availability.~~ and/or

availability. To recover from availability or confidentiality violations is out of the scope of this document. However, we argue that the design of the applications should encompass cryptography techniques which ~~do not prevent unauthorized disclosure of restricted information but reduces its~~ may reduce data relevance and protects the data ~~*secrecy* [?] .~~ secrecy [?] .

Intrusion removal processes recover from integrity violations recreating an intrusion-free state. Due to the fact that the system availability is result of the integrity of each system component [?], these processes contribute to recover the system availability. More, the removal processes should not reduce the system availability. We argue that intrusion recovery services should avoid the system downtime and support the execution of recovery processes in background without externalization to users. They can accomplish some of the goals of intrusion tolerance mechanisms if they keep providing a, possibly degraded but adequate, service during and after an intrusion recovery.

The following section explains distinct recover processes where the application integrity is restored by determining the effects of the detected intrusion actions, reverting them and recreating a correct state.

### 3.3 Formalization of the Recovery Process

As discussed in previous section, intrusion recovery services detect the intrusion effects, revert them and restore the application to a correct state. Here we explain this process by formally modeling the application as a sequence of actions and outline the distinct approaches to perform intrusion recovery.

An application execution is modeled as a set of actions $A$ and a set of objects $D$. Actions are described by a type (read, write, others more complex), the value(s) read/written, and a timestamp (which defines the order of the actions). Each object has a state or value and a set of operations that can modify it. We specify $A_{intrusion}$ as the subsequence of actions of $A$ whereby the attacker compromises the application during the intrusion, $A_{after}$ as the subsequence of actions that begins after the intrusion ~~begins~~ begin (including the first action of the intrusion) and $A_{legal}$ as the subsequence of legitimate actions in $A$. Notice that $A_{legal} = A - A_{intrusion}$.

A recovery service aims to set the state of a service to $D_{recovered}$ at the end of the recovery process. This set shall be composed of objects as if their state was defined exclusively by a set of legitimate actions $A_{recovered}$. The objects of the subset $D_{recovered}$ represent a new ~~*intrusion-free* and *consistent state*~~**intrusion-free** and **consistent state**. A consistent state is a state valid according to the application specification. The state is intrusion-free if it is created only by legitimate actions. ~~Both do not guarantee that the application behaves with respect to a~~ as frases estavam a mais If the application respects the specification (correctness), ~~which is mainly a responsibility of application-level code. A consistent guarantees that it is predictable according to the the application specification. If the application respects the specification,~~ then changing from $D$ to $D_{recovered}$ performs service restoration ~~[?]~~ [?], i.e., restores the application service to a correct behavior.

A basic recovery service, like a backup mechanism, tries to obtain, after the recovery, the subset of object values $D_{recovered}$ written before the intrusion, which do not include the attacker actions, i.e., $D_{recovered} = D - D_{after} : D_{recovered} \cap D_{intrusion} = \emptyset$.

We define the set of **tainted** actions, $A_{tainted}$, and the set of tainted objects, $D_{tainted}$, at a certain instant in the following way: if an action belongs to $A_{intrusion}$, then it belows to $A_{tainted}$; if an object belongs to $D_{intrusion}$ then it belongs to $D_{tainted}$; if an action

in $A_{legal}$ reads an object in $D_{tainted}$, then that action belongs to $A_{tainted}$; if an action in $A_{tainted}$ writes in an object in $D_{legal}$, then that object belongs to $D_{tainted}$. Therefore, $A_{tainted}$ includes $A_{intrusion}$ but typically also actions from $A_{legal}$ that were corrupted by corrupted state. Also, $D_{tainted}$ includes $D_{intrusion}$ but typically also objects from $D_{legal}$ that were corrupted by corrupted state. Then, the set of object values written only by ~~non-intrusive~~ non-malicious actions is not the same as the set of objects obtained after removing the values written by ~~intrusive~~ malicious or tainted actions, i.e., $D_{legal}$ written by $A \cap A_{intrusion} = \emptyset$ is not equal to $D_{legal}$ written by $D - D_{intrusion}$ or $D_{legal}$ written by ~~$D - D_{tainted})$~~ $D - D_{tainted}$. In other words, to remove the objects written by intrusions and tainted actions is necessary but not enough to obtain the set of objects that are not produced by the subset $A_{tainted}$.

More advanced systems [**?**,**?**,**?**] attempt to obtain $D_{recovered}$ where the values of the objects of $D_{tainted}$ are removed from the current state $D$. To do so, the value of each object in $D_{tainted}$ is replaced by a previous value. These systems keep the objects written by legitimate actions, $D_{legal}$, unmodified.

Consider an hypothetical application execution at a certain point in time, after the intrusion, where $A$ is replaced by the set $A_{recovered} = A - A_{intrusion} = A_{legal}$, i.e., where the intrusion actions $A_{intrusion}$ are not executed. We would have in this application: $A \cap A_{intrusion} = \emptyset \implies D_{intrusion} = \emptyset, A_{tainted} = \emptyset \implies D_{tainted} = \emptyset$. In other words, if the ~~intrusive~~ malicious action are removed, the state, $D$, ~~do~~ does not have the objects written by $A_{intrusion}$. For this reason, the sequence of tainted actions $A_{tainted}$ is empty~~and the originally tainted actions~~. The set of tainted actions in the real application execution, which includes $A_{intrusion}$, would read different values and have a different execution if $A_{intrusion}$ would be empty. Therefore, ~~$A_{tainted}$ should be **replay**~~ if $A_{intrusion}$ and $D_{intrusion}$ are removed, then $A_{tainted}$ should be **replayed** because the actions of $A_{tainted}$ are not contaminated by malicious data during their re-execution. The replay process restores the application to a correct state $D_{recovered}$, which is intrusion-free.

~~The works explained in the following sections define two distinct approaches to update the set of object $D$ to $D_{recovered}$ because of changes in the execution of $A_{tainted}$: **rewind** and **selective replay**. The selective replay approach loads only the previous versions of the tainted objects, $D_{tainted}$, and replays only the legitimate operations, which were tainted, $A_{tainted} \notin D_{intrusion}$, to update the objects in $D$. The $D_{legal} \notin D_{tainted}$ remain untouched. The alternative approach, rewind [**?**] , designates a process that loads a system wide snapshot previous to the intrusion moment and replays every action in $A - A_{snapshot} - A_{intrusion}$. However, this process can take a long time.~~ The sequence of actions $A_{before}$ performed before the intrusion, i.e., $A_{before} = A - A_{after}$, can be extensive. Each action takes a variable but not null time to perform. Therefore, to replay $A_{recovered} : A_{before} \subseteq A_{recovered}$ may takes an excessive amount of time. We define the subsets $D_{snapshot}(t)$ and $A_{snapshot}(t) : A_{snapshot} \subseteq A$ as the subsets of object values and actions executed before the begin of a snapshot operation at instant $t$. The snapshot operation copies the value of the object immediately or on the next write operation. If the attack is subsequent to $t$, then $A_{after} \cap A_{snapshot} = \emptyset \implies (A_{intrusion} \cup A_{tainted}) \cap A_{snapshot}(t) = \emptyset$, i.e., the snapshot is not affected by intrusion. For that reason, the service can replay only $A - A_{snapshot} - A_{intrusion}$ using the object set $D_{snapshot}$ as base.

The works explained in the following sections define two distinct approaches to update the set of object $D$ to $D_{recovered}$ because of changes in the execution of $A_{tainted}$: **rewind**

and **selective replay**. The selective replay approach loads only the previous versions of the tainted objects, $D_{tainted}$, and replays only the legitimate operations, which were tainted, $A_{tainted} \notin D_{intrusion}$, to update the objects in $D$. The $D_{legal} \notin D_{tainted}$ remain untouched. The alternative approach, rewind [**?**] , designates a process that loads a system wide snapshot previous to the intrusion moment and replays every action in $A - A_{snapshot} - A_{intrusion}$. However, this process can take a long time.

A **version** is a snapshot of a single object value before the instant $t$. They can be recorded with the sequence of actions that read or write them before the instant $t$. We define a **compensating** action as an action that reverts the effects of a original action, for instance writing a previous value. A compensation process can obtain a previous snapshot or version. For this propose, we define the sequence $A_{compensation}(t)$ as the compensation of $A_{posteriori}(t)$, the sequence of actions after instant $t$. The compensation process applies the sequence of compensating actions $A_{compensation}(t)$ on the current version of the objects, in reverse order, to obtain a previous snapshot or version.

Recovery services have two distinct phases: **record phase** and **recovery phase**. The record phase is the service usual state where the application is running and the service records the application actions. In order to perform replay, the application actions do not need to be idempotent but their re-execution must be deterministic. The record phase should record the actions input and the value of every non-deterministic behavior to turn their re-execution into a deterministic process. The **recovery phase** can have three phases: determine the affected actions and/or objects, remove these effects and replay the necessary actions to recover a consistent state, as already explained in Section 3.2. Recovery services, which support **runtime recovery** when record and recovery phases occur simultaneously, do not require application downtime.

Since the actions read and write objects from a shared set of object values $D$, we can establish dependencies between actions. Dependencies can be visualized as a graph. The nodes of an **action dependency graph** represent actions and the edges indicate dependencies though shared objects. The **object dependency graph** establishes dependencies between objects through actions. Dependency graphs are used to order the re-execution of actions [**?**], get the sequence of actions affected by an object value change [**?**], get the sequence of actions tainted by an intrusion [**?**] or resolve the set of objects and actions that caused the intrusion using a set of known tainted objects [**?**].

A **taint algorithm** aims to define the tainted objects $D_{tainted}$ from a source sequence of ~~intrusive~~ malicious actions $A_{intrusion}$ or objects $D_{intrusion}$ using the dependency graph. The **taint propagation via replay** [**?**] algorithm begins with the set of $D_{tainted}$ determined by the base **taint algorithm** and expands the set $D_{tainted}$. It restores the values of $D_{intrusion} \cup D_{tainted}$ and replays only the legal actions that output $D_{intrusion} \cup D_{tainted}$ during the original execution. Then it replays the actions dependent from $D_{intrusion} \cup D_{tainted}$, updating their output objects. While the forward actions have different input, they are also replayed and their outputs are updated.

Dependencies are established during the record phase or at recovery time using object and action records. The level of abstraction influences the record technique and the dependency extraction method. The abstraction level outlines the recoverable attacks. In the next paragraphs, we explain the relevant works at abstraction levels where services deployed in

PaaS are attacked: operating system, database and application.

### 3.4 Recovery at ~~operating system~~ Operating System Level

In this section, we present the main intrusion recovery proposals for operating systems. First, we present the proposal that introduced the main dependencies for operating systems. Then, we present two intrusion recovery systems that use dependency rules and tainting propagation via replay, respectively. Finally, we present proposals to recover from intrusions in computing clusters, virtual machines and network file systems.

**BackTracker [?]:** Backtracker proposes a tainting algorithm to track intrusions. It does not perform any proactive task to remove or recover from intrusions but provides a set of rules for intrusion recovery in operating systems.

Backtracker proposes a tainting algorithm that does tainting analysis offline, after attack detection, as follows. First a graph is initialized with an initial set of compromised processes or files, $D_{tainted}$, identified by the administrator. Then, Backtracker reads the log of system calls from the most recent entry until the intrusion moment. For each process, if it depends on a file or process currently present in the graph then the remaining objects dependent from the process are also added to the graph. The result is a dependency graph (Fig. 2) with the objects, including $D_{intrusion}$, which the compromised objects depend from. The following dependency rules establish the graph edges:

- *Dependencies process-process:*
  - *Process depends on its parent process*: Processes forked from tainted parents are tainted.
  - *Thread depends on other threads*: Clone system calls to create new threads establish bi-directional dependences since threads share the same address space. Algorithms to taint memory addresses [?] have a significant overhead. Signaling communication between processes also establish dependencies.
- *Dependencies process-file:*
  - *File depends on Process:* If the process writes the file.
  - *Process depends on File:* If the process reads the file.
- *Dependencies process-filename:*
  - *Process depends on filename:* If the process issues any system call that includes the filename, e.g., open, create, link, mkdir, rename, stat, chmod. The process is also dependent of all parent directories of file.
  - *Filename depends on process:* If any system call modifies the filename, e.g., create, link, unlink, rename.
  - *Process depends on directory:* If the process reads one directory then it depends on every all filenames on directory.

Objects shared between many processes, e.g., */tmp/* or */var/run/utmp* are likely to produce false dependencies leading to false positives. Therefore, Backtracker proposes a *white-list* filter that ignores common shared files. However, this technique relies in the administrator knowledge. More, it generates false negatives because it allows the attackers to hide their actions in objects that belong to the white-list.

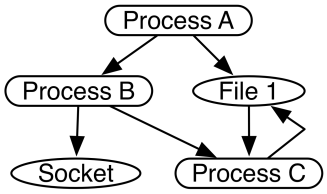


Fig. 2: Dependency graph generated by BackTracker

**Taser [?]:** Taser removes the intrusion effects from the file system used by the operating system. To do that, it loads a previous version of ~~the tainted files from a file system snapshot. Then~~ each tainted file, $D_{tainted}$, from a file-system snapshot, $D_{snapshot}(t)$. Then, to recover the tainted objects, it replays the legitimate ~~actions after the intrusion to recover the latest value~~ modification actions of each tainted object ~~before the intrusion. .~~ since the snapshot instant $t$.

Taser relies on Forensix [?] to audit the system actions during the record phase. Forensix logs the names and arguments of every system call related to process management, file system operations and networking. In order to determine the intrusion effects, Taser builds a *object dependency graph* using a set of rules similar to the rules of Backtracker [?]. The object definition encompass files, sockets and processes. Since these rules result in a large number of false dependencies, which mark legitimate objects as *tainted*, Taser provides not only a white list mechanism but also establishes four optimistic policies that ignore some dependencies. However, attackers can leverage these optimistic policies to penetrate the operating system.

The recovery phase is started with a set of tainted objects provided by an administrator or an IDS. The provided set of objects can either be the source or the result of an attack. In the latter case, Taser, like Backtracker [?], transverses the dependency graph in reverse causality order to identify the set of attack source objects, $D_{intrusion}$, which compromised the provided objects. After, at *propagation phase*, Taser transverses the dependency graph from the source objects of the attack, $D_{intrusion}$, to the current moment, adding all tainted objects to the set $D_{tainted}$.

Taser removes the intrusion effects loading a previous version of the tainted objects from a file-system snapshot, $D_{snapshot}(t)$. Then, to recover a coherent state, Taser performs *selective replay*~~replaying, ~~, i.e., it replays, sequentially, the legitimate write operations of the tainted ~~objects~~ files since the snapshot. Non-tainted files remain unchanged. Since TASER does not checkpoint the state of processes nether the input of each system call, the system must be restarted to remove the current non-persistent states and processes must be replayed from the beginning to load their non-persistent state and perform their system calls with the correct state. This issue has a significant overhead specially for long-run processes as web servers.

Taser does not update the objects originally dependent from tainted objects. In other words, the replay process only recovers a consistent state for the originally tainted objects. Therefore, Taser ignores the set of actions that read the modified version of the tainted files and have a different execution and output. This problem is addressed in [?]. More, Taser uses rules to determine the affected files and remove their effects, so it can mistakenly mark

legitimate operations as tainted and induce to legitimate data losses.

**RETRO [?]:** RETRO provides the capability of removing files affected by a set of identified attacking actions. It restores the corrupted files to a previously version using a file system snapshot and then performs selective replay using *taint propagation via replay.*

During the record phase, the kernel module of RETRO creates periodic snapshots of the file system~~storing not only the input but also~~ . RETRO logs the input and the output objects of ~~system calls~~ each system call and their associated process. The object definition encompasses not only files and directories but also TCP sessions and the operating system console ~~. The dependency graph is finer-grained than the graph of Backtracker [?] , since~~ (tty). The dependencies are established per system call instead of per process ~~, and it contains more information~~ graph. Therefore, the graph is finer-grained than the graph of ~~Taser~~Backtracker [?] and reduces the number of false positives.

During the recovery phase, RETRO requires the administrator to identify $A_{intrusion}$, processes, system calls or $D_{intrusion}$ objects which caused the intrusion. First, it removes the ~~intrusive~~ malicious system calls from the graph. Then it performs *taint propagation via replay.* To do so, it loads a previous version, from a snapshot, of the objects in $D_{intrusion}$. Then, the system calls, which are dependent from the restored objects, are replayed and their output objects are updated. The forward system calls, which depend from the updated objects, are also replayed while their inputs are different from the original execution. The propagation is done thought the output of system calls with different execution. The recovery process terminates when propagation stops. ~~Since RETRO~~ Since RETRO records the system call input, it can replay processes with system call granularity instead of process granularity, so the replay process may stop earlier. However, ~~since RETRO does not checkpoint the state of processes, the system must be restarted to remove the current non-persistent states and processes must be replayed from the beginning to load their non-persistent state and perform their system calls with the correct state. This issue has a significant overhead specially for long-run processes as web servers.~~ this mechanisms forces that the process re-executions has the same sequence of system calls as its original execution. Therefore, the process source code nether its sequence of actions can not change.

Since RETRO replays the processes, the external state may change. External changes are manifested through terminal and network objects. RETRO emails the administrator with the textual difference between the original and recovery outputs on each user terminal. RETRO also maintains one object for each TCP connection and one object for each IP and port pair. It compares the outgoing data with the original execution. Different outgoing traffic is presented to the user. Later work of the same authors, *Dare* [?], extends RETRO to recover from intrusions in distributed systems. It adds the dependencies through sockets. The machines involved in a network session add socket objects to the dependency graph. Network protocol, source and destination IP and ports and one ID, which is exchanged in every package during the connection, globally identify each socket object. The recovery phase is similar RETRO except on network system calls handlers. Compromised network sessions must be replayed since their input depends on destination server. Therefore, prior to invoke the system call for network session establishment, Dare invokes a remote method at receiver Dare daemon to rollback the network session. The receiver rollback the dependent objects to the version before session establishment and replays their dependencies. The remote method response includes the re-execution output. The local system updates the system call output. The re-execution is propagated if this output is different.

The efficiency of RETRO comes from avoiding to replay the actions if their input ~~is~~ remains equal. RETRO requires human intervention to solve external inconsistencies. Dare solves it but it is limited to clusters where every operating system runs a Dare and RETRO daemon. RETRO can not recover from an intrusion whose log files have been garbage collected or deleted. Dare supports distributed re-execution but, as RETRO, the affected machines must be offline because its propagation algorithm shutdown the service during the repair phase.

**Bezoar [?]:** Bezoar proposes a rewind based approach to recover from ~~network sourced attacks~~ attacks coming from the network in virtual machines (VM). The snapshot is performed by VM forking using copy-on-write. This snapshot technique encompasses the entire system: processes and kernel spaces, resources, file system, virtual memory, CPU registers, virtual hard disk and memory of all virtual external devices. Bezoar tracks how the data from network connections propagates in memory. During the recovery phase, the administrator identifies the network connections used by the attackers. Bezoar removes the intrusion effects using rewind. It loads a previous VM snapshot and it replays the system execution ignoring all network packets from the identified malicious sources.

The recovery process using rewind is longer than RETRO or Taser because all external requests are replayed. Bezoar requires system outage during the replay phase and does not provide any external consistency warranties.

**Repairable File System (RFS) [?]:** RFS is designed to recover compromised network file systems. The novelty in RFS comparing with the ~~previous~~ previously introduced systems is its client-server architecture. RFS includes a network file system (NFS) client module and a server NFS module.

The client module tracks the system calls using ExecRecorder [?] and establishes the dependence between processes and NFS requests. Every request to the NFS server is marked with a request ID and the client ID. At server side, the request interceptor logs all requests sent by clients to update files. Requests are ordered in per-filename queues. They are processed locally and write operations are mirrored to external server asynchronously after reply. The external server keeps all file versions.

During the recovery phase, the server defines the contaminated processes using the client logs. A process is contaminated if it matches a set of rules similar to Backtracker [?]. RFS adds the concept of contaminated file and contaminated file block. If a file is contaminated, all its blocks are contaminated. The reverse is not true, i.e., processes remains non-contaminated if they read a correct block from a contaminated file. RFS uses the version server to rollback only the affected files. ~~RFS is resilient to log attacks but legitimate clients need to store their log. More, RFS, as Taser, does not update the files dependent from tainted files and can remove legitimate files due to false positive dependencies.~~

**Summary:** Dependencies at the operating system level are established by rules based on BackTracker [?]. These rules are vulnerable to false positives and false negatives. While Taser [?] and RFS [?] recover from intrusions removing the effects only in corrupted files, RETRO [?] ~~propagates the effects of read dependencies. However, to propagate the effects based on input changesrequires to store the input of every action and to replay the processes since the beginning.~~ removes the values written by $A_{intrusion}$ and replays the forward system calls while their input changes. If $A_{intrusion}$ is identified properly, then RETRO does not have false positives.

The operating system level services are vulnerable to attacks inside kernel because they only audit the system calls. Attackers can compromise the recovery system because the log daemons are installed in the machine where attacks are performed. The recovery guarantees are limited by the system administrator capability to detect the attack and pinpoint the intrusion source. Administrators must avoid false positives to prevent legitimate data losses. However, remove false dependencies can take a while because the low abstraction level creates bigger dependency graphs and logs.

### 3.5 Evaluation of Technologies

The previous subsections describe various services that use different approaches to recover from intrusions. The level of abstraction outlines the recorded elements. Intrusion recovery services at the operating system level records the system calls and the file system. At the database level they track inter-transaction dependencies. At the application level they track transactions, requests and execution code (Table 1). The log at operating system level is more detailed but may obfuscate the attack in false positive prevention techniques. At database and application level, the log is semantically rich but does not track low level intrusions.

Most of services require the administrator to identify the initial set of corrupted actions or objects (Table 1). The administrator may be supported by an intrusion detection system (IDS). The alternative, proposed in Warp [**?**], tracks the requests which invoked modified code files. The identification of the actions or objects can incur on false positives and negatives due to administrator mistakes. Tracking the invoked code files requires to change the interpreter (JVM, Python, PHP) but avoids false positives and negatives.

The taint algorithm, which determines $D_{intrusion}$ and $D_{tainted}$, is performed statically using the original execution dependencies recorded in a dependency graph or dynamically replaying the legitimate actions which have a different input in replay phase than in original execution. The later reflects the changes during the replay phase, therefore it is clearly better but requires to store the input of every action. An alternative proposed by Undo for Operators [**?**] sorts the requests, using the knowledge of the application protocol, and then load a previous snapshot and replays the legitimate requests. This approach is possible because *every* legitimate request is replay with a known order. The tradeoff between perform taint propagation via replay or replay every request is equivalent to a trade-off between storage and computation. In the first, the system must store every version of each data item while in the later system only stores the versions of each data item periodically. In the worst case, if all data entries are tainted by the intrusion, the first approach will replay the same number of requests as the second.

Warp [**?**] supports application source code changes tracking the original code invocations and comparing output when the application functions are re-invoked. Undo for Operators [**?**] support application changes using application dependent compensating actions to resolve conflicts. Goel *et Al* [**?**] do not support code changes because the used taint via replay mechanism stops if the action input is similar. The remaining proposals do not support code change.

| System | Data logged | Intrusion identification data | Taint mechanism | Supports ~~code~~ action changes |
|---|---|---|---|---|
| [?]Taser | System call inputs | Tainted or intrusion source objects | Graph | ~~-~~✗ |
| [?] [?] RETRO,Dare | System call inputs and outputs | Intrusion ~~Objects~~ objects or Actions | Taint via replay | ~~-~~✗ |
| [?] ITDB | Read/write sets using SQL parsing | Intrusion objects | Set expansion (graph) | ~~-~~✗ |
| [?] Phoenix | Read/write sets using DBMS modification | Intrusion ~~Actions~~ actions | Graph | ~~-~~✗ |
| [?] Goel *et al.* | User, session, request, code execution and database rows | Intrusion requests | Graph and taint via replay | ✗ |
| [?] [?] Warp,Aire | Client side browser interactions, requests, user sessions, invoked PHP files | Requests which invoked the modified source code files | Taint via replay | ✓ |
| [?] Undo for Operators | Requests | Intrusion requests | Application protocol dependencies | ✓ |
| Shuttle (this work) | User requests, session and read/write set using DBMS changes | Intrusion requests | Graph and taint via replay | ✓ |

Table 1: Summary of storing and intrusion tracking options

At recovery phase, the intrusion recovery services remove the intrusion effects and recover a consistent state (Table 2). Three of the possible options to remove the intrusion are: snapshot, compensating and versioning. Versioning, which is a per-entry and per-write snapshot, is finer-grained and allows the reading of previous versions without replay the actions after the snapshot. However, the storage requirements to keep the versions of every entry in a large application can be an economic barrier. The usage of actions compensation requires the knowledge of the actions that revert the original actions.

To recover a consistent state, intrusion recovery services should replay the legitimate actions dependent from the intrusion actions (Section 3.3). While Undo for Operators [?] propose to replay all legitimate user requests sorted by an application-dependent algorithm, the remaining services use tainting via replay to selectively replay only the dependent actions. The late approach may hide indirect dependencies [?] but recovers faster. Undo for Operators [?], replays every request generating conflicts that must be solved in order to achieve a consistent state. Warp [?] queues the conflicts for later solving by users. The proposals [?,?,?,?] support runtime recovery (Section 3.3). This characteristic is required to support intrusion tolerance but allows intrusion to spread during the recovery period. To prevent that, ITDB [?] denies the access to tainted data. However, it compromises the availability during the recovery period.

| System | Previous state recovery | Effect removal | Replay phase | Runtime recovery | Externally consistent |
|---|---|---|---|---|---|
| [?]Taser | Snapshot | ~~Remove intrusion requests;~~ load legitimate entries value from a snapshot | No | | |
| [?] [?] RETRO,Dare | Snapshot | ~~Remove intrusion requests;~~ load legitimate entries value from a snapshot | Tainting via replay | | |
| [?] ITDB | Transaction compensating | Compensate tainted transactions | No | ✓ | |
| [?] Phoenix | Row versioning | Abort tainted transactions | No | | |
| [?] Goel *et al.* | Transaction compensating | Compensate tainted transactions | Tainting via replay | | |
| [?] [?] Warp,Aire | Row versioning | Load previous entry version | Tainting via replay | ✓ | ✓ |
| [?] Undo for Operators | Snapshot | ~~Load~~ load snapshot | Replay all requests sorted by application semantics | | ✓ |
| Shuttle (this work) | Snapshot | Load snapshot | Replay tainted requests sorting by original execution read/write sets | ✓ | ✓ |

Table 2: Summary of state recovery options

## 4 Evaluation

The evaluation of the proposed architecture will be done experimentally, building a prototype. This evaluation aims to validate and evaluate Shuttle and to prove its scalability in large implementations. ~~that it~~ The validation aims to show that Shuttle allows to recover from attacks against at least five of the OWASP Top 10 Application Security Risks: injection flaws, broken authentication, security misconfiguration, missing function level access control and using components with vulnerabilities [?]. Shuttle should achieve the proposed goals and its results should match the evaluation metrics bounds which allows the Shuttle usage in real environments. We evaluate our proposal by how effective and how efficient it is, i.e., the false positive and false negative rates, the record and recovery overhead and the availability during the ~~recover~~ recovery process. In detail, we intend to measure the following:

- **Recording:** We want to quantify the *logging overhead* imposed measuring the *delay* per request and system *throughput*, *resources usage* and *maximum load* variance due to recording mechanisms comparing with common execution, the *log size* needed to recover the system from a previous snapshot. We also will measure the time required to perform a system snapshot.
- **Recovery:** We aim to measure the effectiveness in terms of *precision* and *recall*. Precision is the fraction of the updated items that are correctly updated. Recall represents the proportion of tainted items updated out of the total number of tainted items. We can measure the number of false positives from precision, and the number of false positives from the recall. We will check the recovery scalability and measure its duration for different numbers of requests, checkpoints and dependencies.
- **Integrity and availability:** We will measure and trace the percentage of corrupted data items and the percentage of available data items during the recovery process. These values help to define the application availability during the recovery process.

– **Consistent replay:** We want to ~~claim~~ guarantee that our service provides the same results as original execution if the requests and application code remain identical, even when supporting parallel and concurrent requests.
– **Parallel replay:** We will measure the performance changes when the replay is performed in parallel.
– **Cost:** We will measure the monetary cost of the intrusion recovery process using a public cloud providers.

We will do the measurements taking into account varying sizes of the state and time between snapshots, the attack type, the damage created, the detection delay, the request arrival rate. More, we will validate if the application integrity is enough to support an adequate service during the recovery process in the hope of providing an intrusion tolerant application.

We will develop a demo web application which will be deployed on a PaaS system. Since independent user sessions would be trivial to replay, the application will have highly dependent interactions between requests from different users. The application will be a Java Spring [**?**] implementation of a Question and Answering (QA) application, like Stack Overflow [**?**] and Yahoo! Answers [**?**]. Spring is one of the most used Java enterprise web frameworks and it is compatible with most of the current PaaS systems. To highlight the Shuttle scalability, the application will store its state in a NoSQL database.

We will develop a *testing tool* to evaluate the service using HTTP requests from multiple nodes coordinated by a master node. The tool aims to simulate a real web site load from multiple geo-distributed clients. To evaluate the efficiency, the tool will measure the response time and throughput of each user. To evaluate the effectiveness, the tool will analyze the consistency of the application responses during the recovery phase. These results will be extracted from a set of intrusion scenarios~~We will compare the overhead during normal operation. New~~ . These scenarios will include accidentally or malicious requests, external actions, e.g. ssh sessions, that compromise the PaaS containers and software updates to fix previous software flaws. These intrusion scenarios will be created according to typical application vulnerabilities. We will compare the overhead during normal and recovery operations.

The prototype evaluation is branched in ~~Private Cloud and Public Cloud~~private cloud and public cloud. Due to public cloud costs, we will implement a test prototype on a private cloud and evaluate the final prototype on a public cloud. First, our prototype will be deployed in a PaaS system provided by AppScale [**?**] or OpenShift [**?**]. The open source PaaS systems will run over OpenStack [**?**]. Later, we will perform the final evaluation running the application over Amazon Web Service IaaS [**?**] or Google Cloud Platform [**?**] to overcome our limited resources and scale to medium enterprise size scenarios.

## 5 Schedule of ~~future work~~Future Work

Future work is scheduled as follows:

– Feb 1 - Jul 30: Service implementation
– Jul 1 - Jul 15: Service deployment on local cloud and testing
– Jul 16 - Aug 1: Service deployment on public cloud (AWS), experimental evaluation of the results.

– April 15 - Aug 15: Write a paper describing the project
– Jul 15 - Set 30: Finnish the writing of the dissertation
– Oct 1, 2014: Deliver the MsC dissertation

The service implementation will be implemented and evaluated in phases. At the first phase, we will design the testing tool, the proxy and the demo application. At the second phase, we will handle the dependency auditors and graph. At the latest phase, we will develop the capability to perform parallel replay and integrate with a PaaS system. After, we will perform the measurements and evaluation. More details about the schedule are available in the Appendix A.

## 6 Conclusion

Intrusion recovery services restore the system integrity when intrusions happen, instead of trying to prevent them from happening. We have presented a detailed overview ~~on~~ of various intrusion recovery services for operating systems, databases and web applications. However, none of these projects provide a scalable service for applications deployed in multiple servers and backed by ~~a NoSQL database.~~ one or more databases.

Having the above in mind, we proposed Shuttle, an intrusion recovery service for PaaS, that aims to make PaaS applications operational despite intrusions. Shuttle recovers from software flaws, corrupted requests and unknown intrusions. It also supports application corrective and preventive maintenance. The major challenges are the implementation of a concurrent and consistent database snapshot, establish accurate requests dependencies, contain the damage spreading and repair the system state in time to avoid application downtime.

Shuttle removes the intrusion effects in PaaS applications and restore the application to a correct state recreating an intrusion-free state. We propose, to the best of our knowledge, the first intrusion recovery service for PaaS with support to NoSQL databases.

**Acknowledgments** We are grateful to Professor Miguel Pupo Correia for the discussion and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via INESC-ID annual funding through the RC-Clouds - Resilient Computing in Clouds - Program fund grant (PTDC/EIA-EIA/115211/2009).

# A Detailed Schedule

| Shuttle | start | end | 72% |
|---|---|---|---|
| **Preparation** | **01/01/14** | **31/05/14** | **100%** |
| Related Work Reading | 01/01 | 30/04 | 100% |
| Architecture Planning | 01/03 | 10/05 | 100% |
| Documment Writting | 04/02 | 10/05 | 100% |
| PaaS and Database Options | 06/01 | 26/03 | 100% |
| Deploy local PaaS | 13/01 | 24/01 | 100% |
| Simple PaaS Application | 13/01 | 20/01 | 100% |
| Project Presentation | 05/05 | 31/05 | 100% |
| **Implementation** | **01/02/14** | **30/06/14** | **88%** |
| Testing Tool | 22/02 | 28/02 | 100% |
| Q&A Application | 21/02 | 27/03 | 100% |
| Proxy: Routing | 01/02 | 21/03 | 100% |
| Proxy: Storage | 19/03 | 28/03 | 100% |
| Redo Node | 05/04 | 13/05 | 100% |
| DB Node: Dependency Extraction | 19/03 | 01/04 | 100% |
| Manager: Dependency Graph | 05/04 | 23/05 | 90% |
| DB Node: Request Ordering | 29/03 | 23/05 | 90% |
| Parallel Redo | 24/05 | 17/06 | 50% |
| Integration with PaaS | 19/06 | 30/06 | 0% |
| **Testing** | **01/07/14** | **01/08/14** | **0%** |
| Local PaaS: Validation | 01/07 | 15/07 | 0% |
| Public PaaS: Measurement | 16/07 | 01/08 | 0% |
| **Document Writting** | **15/04/14** | **30/09/14** | **4%** |
| Papper | 15/04 | 15/08 | 5% |
| Thesis | 15/07 | 30/09 | 2% |

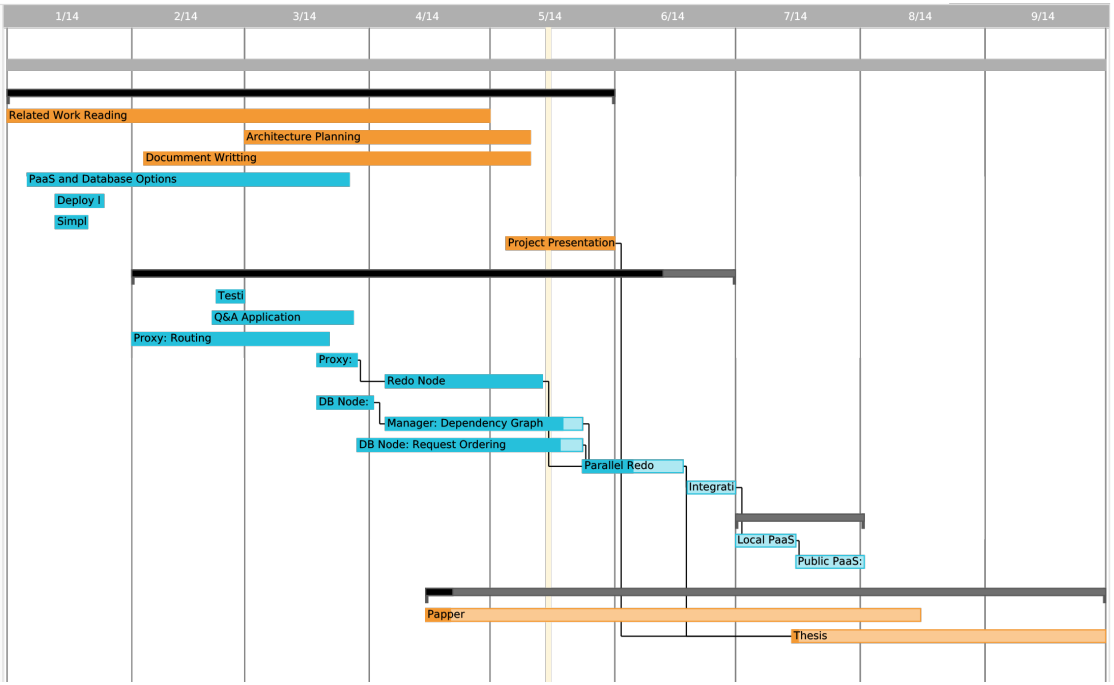


Fig. 3: Project Gantt Schedule