



# 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 for this work is the increasing number of intrusions and critical applications in the cloud, particularly in the emergence of "deveria ser "in the emergent"? the PaaS model. 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 "ou obtain?" computation resources and services on demand [1,2], PaaS in "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 [3]. 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 "to give a well tested and integrated development and deployment framework [4]. PaaS systems are provided either by cloud providers [5,6,7,8] or open source projects [9,10,11]. The success of PaaS systems will also "Com o also queria dar liberdade para outros aspectos: custo, performance... como posso deixar em aberto?" 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 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 has significant business losses [12]. Recovery services are needed to remove the intrusion and restore the correct



application state. as frases ficam curtas demais?

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 [13]. 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 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* <sup>devia escrever SHUTTLE para destacar mais?</sup><sup>1</sup>, 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. To Recover 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 [14]. Shuttle leverage the resources scalability - is correcto para designar o facto de serem virtualmente infinitos? E que se for private clouds are mais limitados... in PaaS environments, and their capability to scale horizontally, to record and to replay non-intrusive user requests. It uses clean PaaS container images to renew the application containers, removing the corrupted state in the image para referir ao goal, and replays user requests in parallel. This mechanism restores 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 retirei a persistencia i.e., a state that allows the applications to behave according to their specification. Shuttle

<sup>1</sup> Shuttle stands for the traveling mean between present and previous application states

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. **remain unmodified as much as possible?**

In this work, we propose the first intrusion recovery service for PaaS applications. We also are amongst the first to consider recovery in distributed storage applications. We are the first intrusion recovery system which take into consideration applications running in various containers. We incorporate the possibility of renew the containers image to remove intrusions. Moreover, only few works have been presented that accomplish intrusion recovery without service downtime. **fica demasiado desconectado entre frases**

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 4 describes briefly the architecture of PaaS frameworks and the proposed architecture for intrusion recovery service. Section 5 defines the methodology which will be followed in order to validate the proposed service. Finally, Section 6 presents the schedule of future work and Section 7 concludes the document.

## 2 Goals

This work addresses the problem of providing an intrusion recovery system for applications deployed in Platform as a Service. 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 intrusion

- *Software flaws*: Computing or database containers have been compromised due to software vulnerabilities.
- *Malicious or accidental corrupted requests*: Accidental 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.

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.
- *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*: NoSQL 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 remain 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 recovery in the levels where PaaS applications are attacked: operating system, database and application. Finally, Section 3.7 discusses the contributions of these works in the context of this work.

#### 3.1 Dependability Concepts

The *dependability* of a computing system is **its** the ability to deliver a trusted service [15] **nao ha problema por nao definir service? ou como esta a ref, sugere consulta?**. 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 introduced either *accidentally* or *maliciously* during the system development, production or operation phases [16,15]. Faults can deviate the system from its specified behavior leading to errors (Fig. 1). *Errors* are inconsistent parts of system. *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 accidental or deliberate, malicious or non-malicious, way - fica bom?**. In particular, we target software flaws, faults and interaction faults from input mistakes, attacks and intrusions [15].

An *intrusion* is a malicious fault resulting from an intentional vulnerability exploitation. As originally proposed generically for faults [15,17], 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 a dependable system, delivering a resilient service, we can use a combination of intrusion forecast, prevention, detection, mitigation, tolerance and recovery, (Fig. 1).

*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 [13,16]. Also system administrators, as humans, can make security configuration mistakes or users may

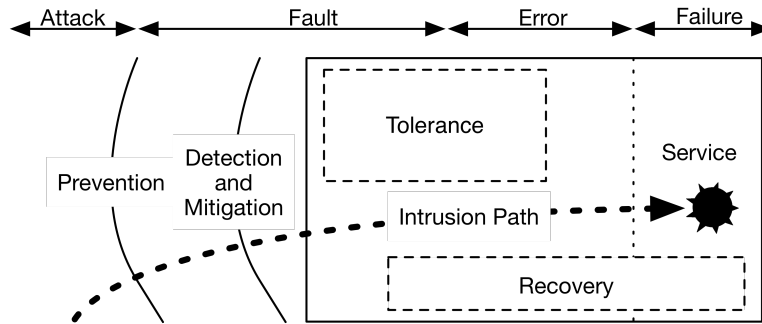


Fig. 1: Intrusion path across the system

grant access to attackers [18]. Moreover attackers can spend years developing new ingenious and unanticipated intrusions having access to what protects the system while the guardian has to predict new methods. 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. - como posso justificar que os mecanismos de prevencao sao penetraveis?

*Intrusion detection and mitigation* mechanisms [19] vale a pena citar o snort? 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. o arguente vai mandar pedras aqui, certo? Intrusion detection systems may enable mitigation techniques, e.g., to block the suspicious TCP session, but attackers may perform the intrusion before intrusion detection systems detects it. - poderia tambem dizer que o atacante explora uma nova vulnerabilidade desconhecida, o IDS (se for por assinatura, e nao por padrao, nao detecta)... ou deixo injustificado que pode ser ultrapassado?

*Intrusion tolerance* is the last line of defense against attacks (Fig. 1). Intrusion tolerance is the ability of a system to keep providing a, possibly degraded but adequate, service during and after an intrusion [20]. Instead of trying to prevent every single intrusion, these are allowed, but tolerated because the system has mechanisms that prevent the intrusion from generate a system failure [21]. Intrusion tolerance hides errors effects using redundancy mechanisms or detecting, processing and recovering faults.

Many intrusion tolerance mechanisms are based on replicating state with Byzantine fault-tolerant protocols [22,23,24]. 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 [25]. 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 [26,23]. na verdade a ideia veio de [27]

The above-mentioned intrusion tolerance replication mechanisms can tolerate some intrusions targeted at design faults (vulnerabilities), as long as diversity is used. Most of fault

tolerance mechanisms do not prevent accidental 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 between legitimate and malicious sources. Furthermore, most 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. In order to recover from intrusions and restore a consistent application behavior, the system administrator - nao ha problema em poder ser automatico, ajudado pelo IDS? 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 [28] after their detection by a IIST (pg16) diz: a group of individuals, called the patch and vulnerability group (PVG), who are specially tasked to implement the patch and vulnerability management program. - o que ponho?!. 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. These techniques may also be applied during the application usage to prevent future intrusions. - desnecessaria?

The third phase, and the one that this work is about, consists in removing the intrusion effects. Intrusions affect the application integrity, confidentiality and availability. To recover - Uma frase pode comecar assim 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 data relevance and protects the data secret [29]. Nao vou levar por ter escrito which do not prevent unauthorized?

Intrusion removal processes recover from integrity violations by recreating an intrusion-free state. Due to the fact that the system availability is result of the integrity of each system component [30], 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 - coloquei some por defesa, devo designar?! of intrusion tolerance mechanisms if they keep providing a, possibly degraded but adequate, service during and after an intrusion recovery. boa ou desnecessaria?

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. inverti a ordem que sugeri, fica mal? e que o recovery e detalhado na proxima sec...

### 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 - o professor refere-se a uma API?. 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 (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*. 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 specification (correctness), which is mainly a responsibility of application-level code. 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 belongs 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. o arguente nao vai embirrar com a repeticao? Then, the set of object values written only by non-intrusive actions is not the same as the set of objects obtained after removing the values written by intrusive 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  $D - D_{tainted}$ . 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 [31,32,33] 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 **o would e excepcionalmente autorizado?** 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 action are removed, the state,  $D$ , do 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 read different values and have a different execution. Therefore,  $A_{tainted}$  should be **replay** if  $A_{intrusion}$  and  $D_{intrusion}$  are removed because the actions of  $A_{tainted}$  are not contaminated during their re-execution. The replay process restores the application to a correct state  $D_{recovered}$ , which is intrusion-free.


alterei a ordem dos paragrafos, como acima falo de fazer replay, inverte a ordem das frases e assim faz a ponte. o problema e que digo "loads a snapshot", quando vou definir o snapshot no paragrafo seguinte, como posso resolver? mais, uso o Asnapshot que defino depois...

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 [34], 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.

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 through shared objects. The **object dependency graph** establishes dependencies between objects through actions. Dependency graphs are used to order the re-execution of actions [35], get the sequence of actions affected by an object value change [36], get the sequence of actions tainted by an intrusion [37] or resolve the set of objects and actions that caused the intrusion using a set of known tainted objects [38].

A **taint algorithm** aims to define the tainted objects  $D_{tainted}$  from a source sequence of intrusive actions  $A_{intrusion}$  or objects  $D_{intrusion}$  using the dependency graph. The **taint propagation via replay** [39] 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 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** [38]: 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 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 [40] 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. **removi o file mapping, nao achei assim tao relevante...**
- *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 **pres** 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.

**Taser [31]:** 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 it replays the legitimate actions after the intrusion to recover the latest value of each tainted object before the intrusion. **posso usar sets que defini acima para conectar as ideias?**

Taser relies on Forensix [41] 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 [38]. 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. **posso tirar o exemplo? - For example, dependencies can be established only by: process forks, file or socket writes by a tainted process, execution of a tainted file and reads from a tainted socket.** 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 [38], 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. Then, to recover a coherent state, Taser performs *selective replay* replaying, sequentially, the legitimate write operations of the tainted objects since the snapshot. Non-tainted files remain unchanged.

Taser does not update the objects 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 [42] e uma extensao que faz taint via replay. li, dai por a referencia, mas nao traz nada de muito novo. 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 [39]:** 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 the output objects of system calls and their associated process. The object definition encompasses not only files and directories but also TCP sessions and the operating system console sim, literalmente o tty, e posso designar?. The dependency graph is finer-grained than the graph of Backtracker [38], since dependencies are established per system call instead of per process, and it contains more information than the graph of Taser.

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 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 replay processes with system call granularity instead of process granularity, 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.

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* [43], 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. **esta discussao da dependencia externa, devo reduzir, podemos falar sobre isso?**

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. **sao argumentos validos?** 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 [40]:** Bezoar proposes a rewind based approach to recover from network sourced attacks **ataques oriundos da rede?** 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 **justifica a descricao de todos os componentes?**. Bezoar tracks how the data from network connections propagates in memory. During the recovery phase, the administrator identifies the ~~the~~ **invasive network connections**. 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. ~~Any external change consequent from repairing actions is external~~

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. **o bezoar esta aqui por usar rewind e por ter sido a base do snapshot copy-on-write... mas posso remover se achar desnecessario**

**Repairable File System (RFS) [44]:** RFS is designed to recover compromised network file systems. The novelty in RFS comparing with the previous **ou previously?** introduced systems is its client-server architecture. RFS includes a network file system (NFS) client module and a server NFS module. **o omentario 'nao faz sentido', era relativo a que frase?**

The client module tracks the system calls using ExecRecorder [45] 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 [38]. RFS adopts 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. **relevante? nao passa de granularity...** 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. **Devo remover este trabalho? Foi daqui que veio a ideia de marcar os pedidos com ID e mais tarde timestamp e guardar os logs nos nos da base de dados mas na pratica tem pouca novidade... TODO: fix the comment: Esta frase nao se entende nada. O que sao log attacks? Porque e que e resistente a eles? O que sao store logs?**

**Summary:** Dependencies at the operating system level are established by rules based on BackTracker [38]. These rules are vulnerable to false positives and false negatives. While Taser [31] and RFS [44] recover from intrusions removing the effects only in corrupted files, RETRO [39] propagates the effects of read dependencies. However, to propagate the effects based on input changes requires to store the input of every action and to replay the processes since the beginning.



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 Recovery at Database Level

A vast number of database management systems (DBMS) support recovery only by loading a snapshot of the database. However, this approach removes not only the malicious effects but also legitimate transactions after the snapshot. The recovery approach we are interested in the document is applied to databases similarly to what was seen in Section 3.4 for operating systems. While the operating system dependencies are established by the system calls, the database dependencies are established by transactions. **neste header, a semelhanca da seccao anterior, defino quais as regras de dependencia genericas. As regras sao partilhadas por todos os trabalhos, incluindo o shuttle e os da seccao 3.6... Neste caso cito um trabalho que tinha as frases perfeitas para descrever as dependencias... Devo mudar o header da seccao para resumir o que vou falar e explicar que introduzo as dependencias ao inicio? faz falta aqui uma linha introdutoria para nao cair em seco?** Compromised transactions are determined from an initial set of bad transactions using read and write rules. “Transaction  $T_j$  is dependent upon transaction  $T_i$  if there is a data item  $x$  such that  $T_j$  reads  $x$ ” [46] and  $T_i$  performs the latest update  $x$ . Transaction dependency is transitive. “A good transaction  $G_i$  is suspected if some bad transaction  $B_i$  affects  $G_i$ . A data item  $x$  is compromised if  $x$  is written by any bad or suspect transaction” [46]. This dependency chain is broken if a transaction performs a blind write, i.e., the transaction writes an item without read it first.


However, legitimate transactions can have different outputs even when their original execution were not tainted. For example, a intrusive transaction can remove a data entry which the following transactions would read and then write other data entries. Since user mistakes are often deletes due to wrong query arguments, this is a relevant issue. Xie *et al.* [47] propose to track the transaction that deleted the data entries keeping a copy of deleted data entries in a separated database table. To add the dependencies from the deleted data items, the SQL statement is performed in the original and delete tracking tables.




The dependency rules require to extract the read and the write set of each transaction, i.e., the set of data entries that each transaction read or modifies. The following proposals use different methods to extract these sets and restore the tainted data entries.

**ITDB [32,46,30]:** Intrusion Tolerant Database (ITDB) performs intrusion recovery in databases using compensating and supports *runtime recovery*, i.e., the database service remains available during the recovery process. ITDB uses the set of dependency rules described above - refiro-me ao 1o paragrafo, como  so designar? and extracts the read and write set parsing the SQL statements. 

faltam as ref em Liu et al... Ele tem varios papers sobre um sistema que ele fez na tese de doutoramento, tirei 3 dos 5 papers Devo colocar ITDB? ou devo colocar Liu et al e citar o paper do ITDB? 

During the record phase, ITDB audits the read and write sets of each transaction. Most relational DBMS only keep write logs. Therefore, Liu *et al.* [32] propose a pre-defined per-transaction type template to extract the read set of parsed SQL statements. This approach is application dependent since updates in application queries require updates in their templates.

At recovery phase, ITDB initiates a set  $D_{tainted}$  with the intrusion source data items. O paper designa sempre como data item, nao diz row ou field.. Then, it reads the log of read and write sets of each transaction from the intrusion moment to the present. For each transactions, ITDB keeps the write set until the transaction commits or aborts; if the transaction commits after reading some data item that belongs to  $D_{tainted}$ , ITDB adds the data items in the transaction write set to  $D_{tainted}$  and the transaction must be compensated. The compensating of a transaction reverts the effects of the original transaction. It performs the inverse modification of the original transaction to restore the previous values. Repaired entries in  $D_{tainted}$  are tracked to prevent compensating of later transactions from restore a repaired data entry to its version after the attack. After this process, the latest legitimate value of  $D_{tainted}$  entries is recovered. 

If the intrusion propagation is faster than the recovery process then the recovery phase is endless because the damage will spread through new transactions. To prevent damage spreading, ITDB blocks the read accesses to the data entries in  $D_{tainted}$  since identification of  $D_{tainted}$  requires log analysis, Liu *et al.* propose a *multi-phase damage container* technique to avoid damage spread through new requests during the recovery phase. This damage containing approach denies the read access to the data items were written after the intrusion. Then, it releases  unconta  the data entries that are mistakenly contained during the containing phase. This approach speeds-up the recovery phase and confines the damaged data items. More, it supports *runtime recovery*. However it decreases the system availability during the recovery period and degrades the performance. 

An alternative approach concerns the throughput asymmetry between the recovery and the user flows. The asymmetry can be neutralized if the repair request priority is increased.



Then, the availability is not compromised anymore but users may read tainted data and propagate the damage slower.

The ITDB architecture includes an intrusion detection system. The IDS is application aware and acts at transaction level. Liu *et al.* propose isolation in terms of users: when a suspicious transaction is reported by the IDS, every write operation from the suspicious user is done in isolated tables. e um extra mas achei interessante...

The ITDB does not perform transaction replay during the recovery phase. Therefore, it ignores that legitimate executions can be influenced by the updated values. Liu *et al.* propose a theoretical model [48] based on possible workflows and versioning. However, predict every possible workflow requires extensive computational and storage resources. mais uma vez, nao me alongo aqui em detalhes. i o paper, tem ideias interessantes para compreender o recovery mas e muito teorico

**Phoenix [33]:** Phoenix removes the intrusion effects using a versioned database. While Liu *et al.* [46] rely on templates of SQL statements and read the log in recovery time, Phoenix changes the DBMS code to extract read dependencies and proposes a runtime algorithm to check dependencies between transactions.

Phoenix performs every write operation appending a new row to the table. This new row includes an unique transaction id to support the restoration of previous row versions. Phoenix modifies the PostgreSQL DBMS code to intercept read queries during their execution and to extract the transaction id of each accessed row. The logged data is used to update the dependency graph.

At the recovery process, Phoenix identifies the set of affected transactions,  $A_{tainted}$ , from a root set of intrusive transactions,  $A_{intrusion}$ , using the dependency graph. Then it changes these transactions status **ou state?** to *abort* in the PostgreSQL transaction log. Since PostgreSQL, in serializable snapshot isolation mode, exposes only the row version of the latest non-aborted transaction, the row is restored and the effect of tainted transactions are removed.

**Summary:** Recovery systems for transactional databases differ on their methods to track the read and write sets and to restore previous values. ITDB is application dependent because it requires pre-defined templates to parse the SQL statements. On the other hand, Phoenix is application independent but DBMS dependent because it modifies the DBMS source code and relies on the usage of serializable snapshot isolation mode.

ITDB is records the previous versions in contrast to Phoenix which just records the transactions. The first requires more storage capacity to save the versions, the second requires the knowledge of the compensating of each transaction and more computation resources during recovery to revert the tainted transactions.

The data item granularity affects the runtime performance overhead and the accuracy of dependency tracking. Coarser granularity, e.g., row, results in lower performance overhead but a higher probability of false dependence if two transactions read and write different portions of the same data item. Moreover, an attack can compromise just an independent part of the transaction and legitimate data is removed. isto aqui e novo, nao esta escrito nos paragrafos anteriores. coloquei aqui por ser algo generico mas aplicavel a todos... as 2 ultimas frases, fazem introducao a proxima seccao...



### 3.6 Recovery at Application Level

Web applications are the main type of application which is deployed in PaaS. These applications are usually composed of a three tier architecture: presentation tier, application-logic tier and data tier. The following works assume the data tier to be a database. Previous section works establish the dependencies between transactions using their read and write sets. However, they ignore the dependencies between transactions at application level. For example, an application can read a record A through a transaction 1, compute a new value B, based on A, and write the value B through a transaction 2. The application-logic tier is often state-less, the requests are independent and the database is the only mean of communication between requests (Section 4.1). **nao devia escrever algo antes desta frase? parece-me solta...**

**Data Recovery for Web Applications [37]:** Goel *et al.* propose a recovery service that selectively removes intrusion effects from web applications that store their persistent data in a SQL database. Since each user request may involve multiple transactions, it tracks the user, the session, the request and the accessed rows. The proposal uses a tainting algorithm and compensating transactions.

During the record phase, a monitor logs each user request and the database rows and tables read and written by the transactions associated with it. The transactions are stamped with an id that establishes the replay order, since the database uses serializable snapshot isolation.

The recovery phase proceeds as follows. First, the administrator identifies the malicious requests  $A_{intrusion}$ . The logged data also allows the administrator to identify every request from a suspicious user. Then, using a dependency graph, it determines the tainted requests. The dependency between transactions is established in a similar manner to the Section 3.5 but **with the table as unit**. Such coarse-grained approach can generate many false dependencies. Therefore Goel *et al.* use *taint propagation via replay*. More, it proposes to modify the PHP-interpreter to reduce the false dependencies between transactions using a variable-level tainting. In other words, variables that read tainted rows or fields are also tainted; rows or fields written by tainted variables are tainted. This process increases the precision of the set of tainted requests. Finally, the compensation transactions of the tainted requests are applied in reverse serialization order on the current state of the database to selectively revert the effects of the database operation issued by the tainted requests.

**POIROT [49]:** POIROT is a service that, given a patch for a newly discovered security vulnerability in a web application code, helps administrators to detect past intrusion that exploited the vulnerability. POIROT does not recover from intrusions but proposes a tainting algorithm for application code. During the normal execution, every user request and response is stored. The log of each request includes the invoked code blocks. After the attack discover phase, the software is updated to fix its flaws. POIROT identifies the changed code blocks and requests dependent from them during the normal execution. The affected requests are replayed. During the re-execution phase, each function invocation is forked into two threads: the updated e non-updated version [50]. Functions invocations are executed in parallel and their output are compared. If outputs are similar, only one execution proceeds otherwise the request execution stops since the request was affected by code patch. **o projecto esta aqui porque este conceito e usado no warp... devo dizer isso no header da section?**

**Warp [36]:** Warp is a patch based intrusion recovery service for single server web applications backed by a transactional database. Unlike previous approaches, Warp allows administrators to retroactively apply security updates without tracking down the source of intrusion and supports attacks at user browser level. Warp is based on RETRO [39] *taint propagation via replay* approach and removes the intrusion effects using a versioned database. The Warp prototype uses PHP and PostgreSQL.

During the normal execution, Warp uses a client browser extension to record all JavaScript events that occur during the visit of each page. For each event, Warp records the event parameters, including the target DOM element. HTTP requests are stamped with a client ID and a visit ID to track dependencies between requests at browser level. On server side, Warp records every requests received and forwards the request to the PHP application. Since Warp uses PHP, an interpreted language, it records which files were used during the original request execution and records the non-deterministic functions. Warp stores the database queries input/output and tracks the accessed table partitions using a SQL statement parser. To conclude, the HTTP response is logged and packed with all execution records.

Warp proposes a time-travel versioned SQL database **propoe no paper, implementam e usam... como devo explicar?**. Each row is identified using a row ID and includes a *start time* and *end time* timestamp columns that establish the row validity period. Warp reverts the intrusion effects, in a specific row, loading one of its previous versions. Warp supports concurrent repair using more two integer columns to define the begin and end of each *repair generation*. At repair phase, the current repair generation ID is incremented to fork the database. User requests are performed in the current generation while recovery requests are perform in the next generation. After replaying the requests retrieved until the begin of the reparation **or repairing?** process, the server stops and applies the remain requests.

During the repair phase, the administrator updates the application software to fix its flaws. Then, Warp determines the requests which used the modified source code files [49] [50]. These requests are the root cause of changes during the re-execution. To update the database and reflect the patch, each user request is replay using a server-side browser. The modified PHP interpreter intercepts non-deterministic function calls during the replay and returns the original logged value. Also, database read queries are replayed only if the set of affected rows is different or its content was modified. Write queries are replayed loading a previous version of the rows and replaying the SQL statement. Each row, which has a different result after re-execution, is now tainted and all requests that read the row are also replay at browser level. Finally, the HTTP response is compared with original and any difference is logged. If responses are different, the following dependent user interactions are replayed in the server side browser. **ajuda a reescrever...**

The server-side browser may fail to replay the original user request. The request may depend from a reverted action of the attacker. These conflict cases are queued and handled by users later.

Warp rewrites the SQL statements, which are used by the application to perform write operations in the database, adding four extra columns per table: start/end timestamp and start/end generation. However, these columns may be a considerable storage overhead. More, Warp is not transparent in the database schema because it does not support foreign keys. It also depends from a client browser extension to log and modify every request. Warp is designed for single machine applications: it does not support application deployed in multiple servers or distributed databases since versions are timestamp based.

**Aire [51]:** Aire is an intrusion recovery service for loosely coupled web services. It extends the concept of local recovery in Warp [36] tracking attacks across services. While Dare [43] aims to recover a server cluster synchronously using RETRO [39] in each node operating system, Aire performs recovery in asynchronous third-party services using Warp [36].

Aire supports remote servers downtime without locking the recovery. To achieve this goal, the pendent repair requests are queued until the remote server is recovered. Since clients may see a partial repaired state, Aire proposes a model based on eventual consistency [52] [53]. The model allows clients to observe the result of update operations in different orders or delayed during a period called *inconsistency window*, i.e., until the remote server recovers. Aire considers the repair process as a concurrent client. To repair key-value database entries, Aire creates a new branch [54] and re-applies legitimate changes. At end of local repair, Aire moves the current branch pointer to the repaired branch.

Like Warp [36], during the normal execution, Aire records the service requests, responses and database accesses. Requests and responses exchanged between web-services, which support Aire, are identified using an unique ID to establish the dependencies.

The recovery phase process as follows. First, the administrator identifies the corrupted requests. The administrator can create, delete or replace a previous request or change a response to remove the intrusion actions. Second, Aire creates a new branch [54] which will contain the set of changes. Third, Aire does a local repair of the application in a similar manner to Warp [36]. In contrast to Warp [36], starts the recovery process in remote servers if at least one of the requests or responses sent previously is modified. To do so, Aire sends a request to the source or destination of the message. If the remote server is offline, the repair request is queued. As repair messages propagate between the servers to start successive repairing actions, the global state of the system is repaired. However, clients may see an inconsistent state during this process. Therefore, Aire applications must support eventual consistency.

The Aire approach using eventual consistency targets a specific kind of application that solve conflicts. Moreover, Aire recovers third-party web-services which must have an Aire daemon. Aire requires the administrator to pinpoint the corrupted requests. Finally, Aire, as Dare [43], requires application downtime during the recovery process.

**Undo for Operators [35]:** Undo for Operators allows the system administrators to recover from their mistakes, software problems or data corruption in email services with file system storage. The design is base on “Three R’s: Rewind, Repair and Replay” [34] where operator loads a system-wide snapshot previous to the intrusion, repairs the software flaws and replays the user-level requests to roll-forward. In contrast to the selective replay approach, where only the tainted entries are reverted, the rewind reverts all entries. On one hand, it requires to redo more requests. On the other hand, it does not require to determine which data entries are tainted because every entry is reverted.

Undo for Operators proposes a proxy interposed between the application and its users. The proxy intercepts application requests. This architecture allows to fix the software flaws or replace the operating system or the application of the application-logic tier elements. However, the proposed architecture is protocol dependent: the proxy implementation only supports IMAP and SMTP. Undo for Operators define the concept of *verb*. Each protocol operation has its own *verb* class. A verb object encapsulates a single interaction (request/response) of the user and exposes an interface to establish the order between requests and their dependencies.

During the normal execution, user requests are encapsulated into verbs and sent to a remote machine: the *undo manager*. The undo manager uses the interface of verbs to define its dependency, i.e., if it can be executed in parallel or must have a causal order with other request. The dependency is established per verb type **depending from the operation and its arguments**. Thus, the dependency mechanism is application dependent. For example, **the operations of email sending** (SMTP protocol) are commutable and independent because there are not ordering guaranties in the email delivery. On the other hand, the order of delete (expunge) and list (fetch) operations in the same folder is relevant and they are not independent. If two verbs are dependent, the second is delayed upon the first is processed. This method establishes a serialization ordering but it can create a significant performance overhead on concurrently arriving interactions and requires protocol knowledge.

The recovery phase strategy is as follows. First the operator determines the corrupted verbs and fixes their order adding, deleting or changing verbs. Second, the application is *rewind*, i.e., a system-wide snapshot is loaded to remove any corrupted data. Third, the operator patch the software flaws of the application. Finally, all legitimate requests started after the intrusion,  $A_{legitimate}$ , are **re-send** by the proxy to rebuild the application state.

External inconsistencies may come out when the requests are replayed. These inconsistencies are detected comparing the re-execution and the original responses. Different responses trigger a compensating action defined per verb to keep an external consistent state. Again, these actions are application dependent.

During the recovery process, Undo for Operators replaces the current system state by a system wide snapshot. Consequently, it does not support runtime recovery. Moreover, it relies on protocol knowledge to establish dependencies, compensating actions and to sort the requests during the recovery phase. Therefore, any protocol change requires to modify the supported verbs. Intrusions can use corrupted requests which are not **encompassed** in the known verb classes and cause a system fail.

**Summary:** Goel *et al.* and Warp [36] establish dependencies using **the request read/write set of the database** and use taint via replay. Undo for Operators [35] establishes dependencies using the knowledge of the operations protocol. Unlike Goel *et al.*, Warp [36] and Undo for Operators [35] support application repair. Warp tracks the requests affected by the modified file. Undo for Operators replays every request using its proxy. While Goel *et al.* ignores external consistence, Warp [36] detects inconsistencies in responses and replays the user interaction in a browser and Undo for Operators uses compensating actions based on protocol-specific knowledge.

The approaches to remove the intrusion effects are also distinct. Goel *et al.* uses compensating transactions to create a system wide snapshot, Undo for Operators loads a previous snapshot and Warp keeps the versions of each data entry. If the tainted request are few, then Goel e Warp have a significant advantage because they replay only the tainted requests. On the other hand, Undo for Operators requires less storage than the remaining options. More, the knowledge of inverse transaction is required to create the compensation of transactions.

**usar goel et al e chato... e preferivel usar: Data Recovery for Web Applications?**

### 3.7 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 [36], 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 `como me posso referir ao interpretador de código` VM, o python, o 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 [35] 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 [36] supports application source code changes tracking the original code invocations and comparing output when the application functions are re-invoked. Undo for Operators [35] support application changes using application dependent compensating actions to resolve conflicts.

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 [35] 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 [47] but recovers faster. Undo for

| System                  | Data logged  | Intrusion identification<br><i>o que que e que o operador tem de fornecer para identificar os quests</i> | Taint mechanism                   | Supports code changes  |
|-------------------------|--|--|-----------------------------------|--|
| [31]Taser               | System call inputs   | Tainted or intrusion source objects  | Graph                             | -  |
| [39] [43] RETRO,Dare    | System call inputs and outputs   | Intrusion Objects or Actions   | Taint via replay                  | -  |
| [32] ITDB               | Read/write sets using SQL parsing  | Intrusion objects  | Set (graph) expansion             | - <i>nao e suportado mas tambem nunca seria aplicavel, ponho um x?</i> |
| [33] Phoenix            | Read/write sets using DBMS modification                                      | Intrusion Actions  | Graph                             | -  |
| [37] Goel <i>et al.</i> | User, session, request, code execution and database rows                     | Intrusion requests   | Graph and taint via replay        | <i>x</i>   |
| [36] [51] Warp,Aire     | Client side browser interactions, requests, user sessions, invoked PHP files | Requests which invoked the modified source code files <i>help</i>  | Taint via replay                  | ✓  |
| [35] 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

Operators [35], replays every request generating conflicts that must be solved in order to achieve a consistent state. Warp [36] queues the conflicts for later solving by users. The proposals [32,33,36,51] *como posso listar uma serie de trabalhos?* support runtime recovery (Section 3.3). This characteristic is required to support intrusion tolerance but *it* allows intrusion to spread during the recovery period. To prevent that, ITDB [32] 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 |
|-------------------------|--------------------------|--|---|------------------|-----------------------|
| [31]Taser               | Snapshot                 | Remove intrusion requests; load legitimate entries value from a snapshot | No  |                  |                       |
| [39] [43] RETRO,Dare    | Snapshot                 | Remove intrusion requests; load legitimate entries value from a snapshot | Tainting via replay <b>help</b>                                       |                  |                       |
| [32] ITDB               | Transaction compensating | Compensate tainted transactions  | No  | ✓                |                       |
| [33] Phoenix            | Row versioning           | Abort tainted transactions   | No  |                  |                       |
| [37] Goel <i>et al.</i> | Transaction compensating | Compensate tainted transactions  | Tainting via replay   |                  |                       |
| [36] [51] Warp,Aire     | Row versioning           | Load previous entry version  | Tainting via replay   | ✓                | ✓                     |
| [35] Undo for Operators | Snapshot                 | 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 Architecture

In the following sections, we first present a generic Platform as a Service architecture. Afterwards, we sketch the architecture of Shuttle and discuss its main design choices.

### 4.1 Platform as a Service

esta seccao vai ser para levar tarefa do arguente porque o PaaS nao esta definido em papers de modo tao claro como o IaaS, e algo novo. Platform as a Service (PaaS) is a cloud computing model for automated configuration and deployment of applications [55,4,14]. The objective is to provide an abstracted, generic and well-tested set of programming and middleware services where developers can design, implement, deploy and maintain their applications written in high-level languages supported by the provider. PaaS provides a service-oriented access to data storage, middleware solutions and other services hiding lower level details and providing access to a pay-per-usage scalable software infrastructure.



~~PaaS application are deployed in one or more containers. [1] container e o termo que escolhi para descrever a unidade de deployment, nao ha muita literatura sobre PaaS como ha do restante, o termo vem do linux containers. Containers are performance and/or functionality isolated environments. Containers are bare metal servers, virtual operating systems instances running in hypervisors, e.g., Xen [56], KVM [57], or lightweight in kernel resources accounting and isolation systems like the Linux cgroups [3]. These containers are managed directly or through an IaaS framework (e.g., OpenStack [58], AWS EC2 [59], Eucalyptus [60]). The applications deployed in PaaS are designed to scale horizontally tenho de definir o que e? Each container is isolated and distributed states are avoided. The application state~~





~~applications written in high-level languages supported by the provider. PaaS provides a service-oriented access to data storage, middleware solutions and other services hiding lower level details and providing access to a pay per usage scalable software infrastructure.~~

PaaS application are deployed in one or more *containers*. [1] **container e o termo que escolhi para descrever a unidade de deployment, nao ha muita literatura sobre PaaS como ha do restante, o termo vem do linux containers.** Containers are performance and/or functionality isolated environments. Containers are bare-metal servers, virtual operating systems instances running in hypervisors, e.g., Xen [56], KVM [57], or lightweight in-kernel resources accounting and isolation systems like the Linux cgroups [3]. These containers are managed directly or through an IaaS framework (e.g., OpenStack [58], AWS EC2 [59], Eucalyptus [60]). The applications deployed in PaaS are designed to scale horizontally **tenho de definir o que e?** Each container is isolated and distributed states are avoided. The application state is usually maintained by the database or the session cookies. A base PaaS framework is composed of the following components (Fig. 2):

- **Load balancer:** Route user requests based on tenant application location and container load.
- **Node controller:** Local node metering, node configuration, tear-up, tear-down of new containers or tenants.
- **Cloud controller:** Manages the creation of new nodes.
- **Metering, auto-scaling and billing:** Retrieve the metering data from each node. The load balancer uses this information to perform request routing while the cloud controller automatically decides when to scale.
- **Containers:** The isolated environment where tenant applications run.
- **Database node:** A single DBMS shared, or not, between multiple tenants. The database middleware is built-on multiple nodes to provide scalability and replication.
- **Authentication manager:** Provides user and system authentication.

The PaaS frameworks are often integrated with code repositories and development tools.

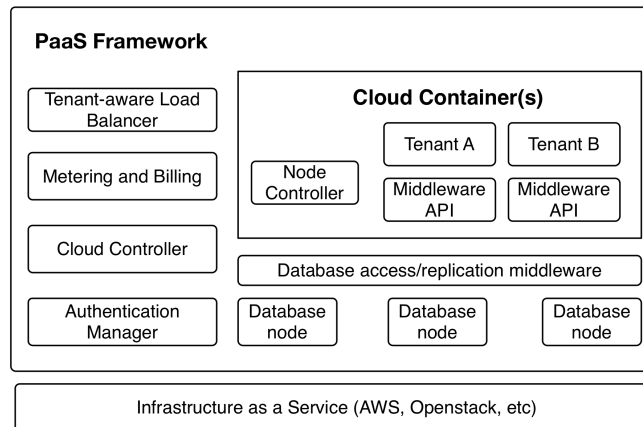


Fig. 2: Generic Architecture of PaaS

## 4.2 Shuttle

Shuttle is an intrusion recovery service for PaaS. It recovers from intrusions on software domain due to software flaws, corrupted requests, input mistakes, corrupted data and suspicious intrusions in PaaS containers. While previous works aimed to perform intrusion recovery in applications with a single database, Shuttle targets PaaS applications which are deployed in containers and backed by **SQL or NoSQL databases - ou coloco so NoSQL?**. Since typical PaaS applications are designed to support high usage loads, our main contribution is a scalable intrusion recovery service that is transparent for application developers. Shuttle logs and replays user HTTP requests to recover from intrusions.

Our service is designed using a proxy architecture that logs the user HTTP requests without modification of PaaS components. Shuttle is composed of the following modules (Figure 3):

- **Proxy:** The HTTP proxy logs all user HTTP requests, adds a timestamp mark to their header and forwards them to the load balancer.
- **Load balancer:** The PaaS system dependent load balancer which spreads the requests through each web server according to their usage.
- **Web servers:** The application-logic tier made of HTTP web servers which serves user requests. The applications access the database through a provided PaaS library. This library contains the **database client interceptor**. This tier auto-scales horizontally.
- **Database servers:** The database servers runs a single transactional database or a NoSQL key-value storage software that stores the application persistent state. Each server contains a **database server auditor** that logs the requests dependency at database level.
- **Request storage:** A scalable database that stores the user requests and replaying metadata.
- **Replay nodes:** The replay nodes are HTTP clients that read previous requests from the requests storage and send them to the web servers for being re-execute during the recovery phase. These worker nodes are coordinated by the manager.
- **Manager:** The master which retrieves the dependencies and coordinates the replay nodes.

PaaS architectures fit the transparency and scalability requirements of Shuttle **devo referir architectures ou architecture?**. They support to auto-scale horizontally the data and application-logic tiers and to deploy an array of replay nodes. More, they include a load balancer and nearly all deployed application use HTTP protocol. Finally, most developers use a provided storage API, which can hide the Shuttle service. **devo retirar este paragrafo? a ideia aqui era explicar como e que, baseado na seccao anterior e nos dados que como e que o Paas faz mesmo sentido...**

Shuttle recovers from software flaws faults and interaction faults from input mistakes, attacks and intrusions. To do so, Shuttle has two phases: record and replay.

During the record phase, the *proxy* logs the user HTTP requests, adds a timestamp to its header as request id and forwards them to the load balancer. These requests are stored in an external distributed storage system asynchronously **o envio e feito assincronamente, como posso dizer?**. The access to database servers is intercepted by the *proxy* that serializes the accesses to each key allowing multiple-reads or a single write. The access order in each database server is recorded including the request id and sent to the *manager*. The response is also stored in the external storage.

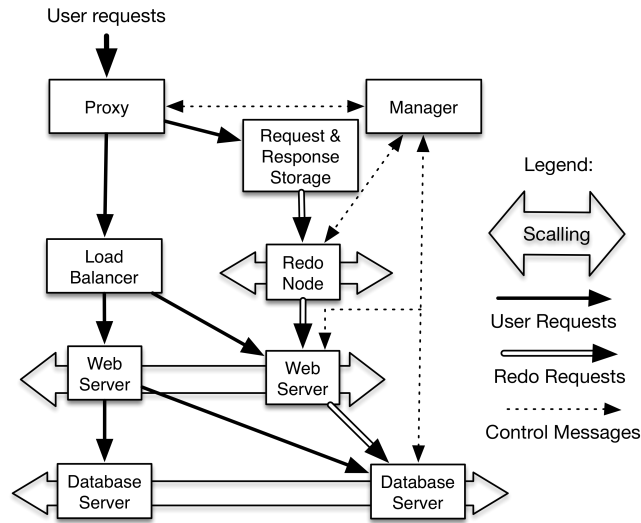


Fig. 3: Overview of the proposed service




The replay phase is initialized when intrusions are detected or the application software requires an update. The detection of the fault is out of the scope of this work. In order to fix the exploited vulnerabilities, Shuttle supports software updates in the application-logic tier or database-tier. More, it supports the modification of user requests to remove mistakes or malicious behaviors. Then the *manager* loads the snapshot on each database node and new PaaS instances, which may include code fixes, are initiated. The new application is intrusion-less because the snapshot and instances are clean and the code is fixed. The requests are replayed in parallel using a set of HTTP clients named *replay nodes*. These worker-nodes are coordinated by the *manager*. The requests are replayed in a different database version. Since users still use the corrupted computation instances and database, the application remains online, perhaps with a degraded behavior, without exposing downtime to users. After recovery, the users requests are switched to the application with fixed state.

The first version of Shuttle will replay every user request [35]. We assume that critical software flaws and intrusions can be detected in a short period of time, from seconds to one week. If a fault exists during a long period then the application may tolerate a longer recovery phase because the recovery process does not require application downtime. Still, one of the main challenges of Shuttle is to reduce the recovery time. To do so, we use two techniques: **snapshot - explicio na Seccao 3.1 como e que o snapshot reduz o tempo de recovery e assegurar volto a dizer... mas posso meter algo tipo: operations checkp** and parallel replay.


Performing **ou 'to perform' snapshot in a distributed NoSQL storage** is not trivial since it has to be consistent with the computation tier user requests. User requests may include multiple database accesses. The snapshot has to include only completed user requests, otherwise the replay phase will be inconsistent because Shuttle would replay a request in a snapshot that contained part of its old persistent state. Therefore, the *manager* leverages the request timestamping to schedule a snapshot. The administrator defines a future in-

stant in time,  $t$ , where the snapshot will occur and the manager forwards this information to every database node. The requests, which have a timestamp bigger than  $t$ , will write in a new version and read the latest version. The remain requests will write in the previous version and read only the versions written by requests with timestamp lower than  $t$ . This mechanism is based on copy-on-write mechanisms.

To support runtime-recovery, the snapshot is loaded in a new branch, similar to the model used by [54]. The branch isolation allows multiple, perhaps simultaneous, replay attempts without compromising the exposed application behavior.

Previous proposals [37] leverage the request serialization provided by snapshot isolation in transactional DBMS to sort the replay operations. However, PaaS applications perform multiple *transactions* accessing independent keys in NoSQL database nodes. Therefore, we need a new approach to serialize the requests and the key accesses during the replay phase. More, the keys accessed during the replay phase may change due to code updates, request modification or multi-threaded execution. While *Undo for Operators* [35] uses the application protocol knowledge to establish the dependency between requests, our work aims to support any application deployed on PaaS. Therefore, we use the captured database key access list to generate a dependency graph. The graph reflects the dependencies between original execution. False dependencies in this graph  not cause data  or inconsistent state since they are used only to order the requests replay as in [46].  porque estava sublinhado?

Shuttle can not execute requests in serial order. It not only would have performance degradation but also lock the replay phase because the requests are originally executed in parallel and may depend from each other. On the other hand, PaaS applications have loosely coupled requests that share data only by database requests. Therefore, Shuttle will perform as many requests in parallel as possible to reduce the duration of the replay phase. The PaaS controller will auto-scale the database and application-logic tiers adding more nodes. More, Shuttle manager can request the PaaS controller to initiate multiple replay nodes to increase the replay performance. Yet, the user requests are processed in parallel using multi-threaded servers and the system messages, including the database requests, do not have a delivering order. With this in mind, Shuttle uses an access list per key to record and to order the accesses to each key. This method is dead-lock free because if the code and requests remain equal. However, if the code or the requests changes then some request may not access the same keys or read/write the same content during the replay phase creating request starvation and locking the replay process. Therefore, the database client interceptor will fetch the list of keys accessed by the original request execution and simulate the access to the remaining keys to unlock the requests, which are waiting in the key.

There are four main sources of non-determinism in each application server: random number generation, time accesses, concurrent threads and requests order. The concurrent threads are ordered using the database access mechanisms,  which were introduced above. To solve the remaining issues, the *proxy* adds pseudo-random number generator seed and its timestamp (also used as request ID) to the request header. Application developers can access these non-deterministic variables in a deterministic way using the PaaS middleware provided by their SDK. This mechanism is language independent. Despite the fact it just returns one timestamp per request, we argue that it is acceptable for most applications.

The elected database to backend the prototype is the key-value storage Voldemort [61]. Voldemort is an open source implementation of Amazon Dynamo DB [52] in use by LinkedIn [62]. Key-value storage databases are common in PaaS environments because they scale properly and provide a simple put,get,delete API. We leverage the API simplicity to avoid delays and mistakes of parsing SQL queries. More, *in* [52] the application is aware of the data schema and decides on the conflict resolution the best suited version. We may leverage this characteristic to solve replay conflicts and application integrity issues.

The external users requests and responses will be stored in the scalable NoSQL database Cassandra [63]. The database can be replicated to a remote site to prevent catastrophic disasters. Shuttle uses the externally stored user requests to recover from any kind of intrusion in the computation and database nodes. Since PaaS defines containers to run each module, Shuttle will collaborate with the coordinator of the PaaS framework to lunch new containers using intrusion-free container images, which may include software updates. The software updates can prevent future intrusions or fix a discovered flaw [?,36]. The old containers, which may contain unknown intrusions, are shutdown. Shuttle updates the application persistent state to be coherent with the new software version.

By replaying the user requests in parallel using intrusion-free containers and a clean database snapshot, we have the best hope to recover from security intrusions in PaaS applications. **fica aceitavel? - ou devo mandar ao lixo?**



## 5 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. **Our aim to validate is showing - esta bem?** that it 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 [64]. 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 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 *queremos garantir que, como devo escrever?* 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 [65] implementation of a Question and Answering (QA) application, like Stack Overflow [66] and Yahoo! Answers [67]. 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, *similar to the ones proposed in the related works. aim to compare the Shuttle effectiveness against previous intrusion recovery proposals. - Tenho o problema de que os restantes projectos fazem testes de performance pura apenas (brown) ou criam cenarios de utilizacao da sua aplicacao (Wordpress, Drupal, MediaWiki, AskBot, etc e veem como e que um pedido malicioso se propaga. Como uso NoSQL e converter todas uma aplicacao tipo Wordpress, seria impossivel... Como devo fazer?)* We will compare the overhead during normal operation. New intrusion scenarios will be created according to typical application vulnerabilities.

The prototype evaluation is branched in 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 [9] or OpenShift [8]. The open source PaaS systems will run over OpenStack [58]. Later, we will perform the final evaluation running the application over Amazon Web Service IaaS [59] or Google Cloud Platform [68] to overcome our limited resources and scale to medium enterprise size scenarios.

## 6 Schedule of 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: Finish 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. **devo acrescentar muito mais? as datas estao folgadas...**



## 7 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. **devo manter NoSQL, Distributed, Key-Value ou SQL?**



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

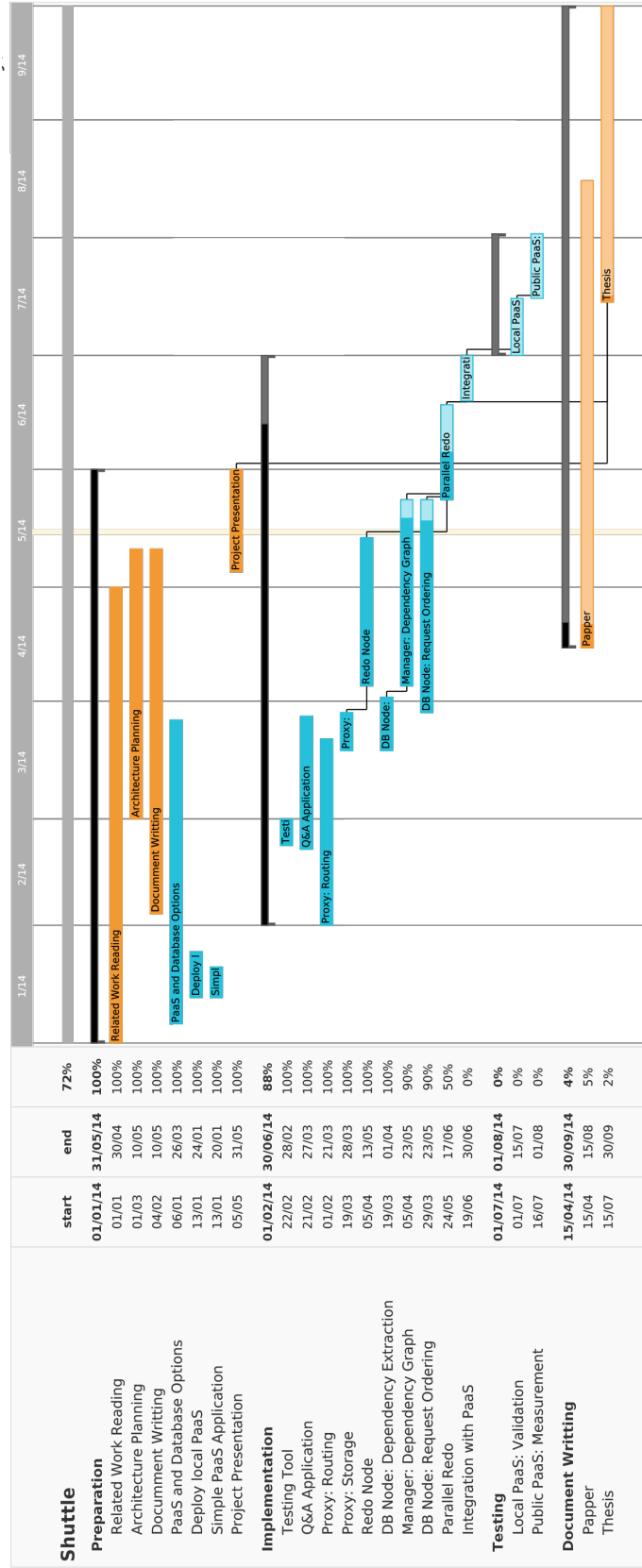


Fig. 4: Project Gantt Schedule



## References

1. A. Lenk, M. Klems, and J. Nimis, "What's inside the Cloud? An architectural map of the Cloud landscape," ... *Challenges of Cloud* ..., pp. 23–31, 2009. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1564625>
2. M. Armbrust, A. D. Joseph, R. H. Katz, and D. A. Patterson, "Above the Clouds : A Berkeley View of Cloud Computing," *Science*, vol. 53, no. UCB/EECS-2009-28, pp. 07–013, 2009. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.7163&rep=rep1&type=pdf>
3. P. Menage, "Adding generic process containers to the linux kernel," *Linux Symposium*, 2007. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.798&rep=rep1&type=pdf#page=45>
4. L. Vaquero, L. Roderio-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer* ..., vol. 41, no. 1, pp. 45–52, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1925869>
5. Amazon elastic beanstalk. [Online]. Available: <http://aws.amazon.com/pt/elasticbeanstalk/>
6. Google app engine. [Online]. Available: <https://developers.google.com/appengine/>
7. Heroku. [Online]. Available: <https://www.heroku.com/>
8. Openshift. [Online]. Available: <http://openshift.redhat.com>
9. N. Chohan, C. Bunch, S. Pang, and C. Krintz, "Appscale: Scalable and open appengine application development and deployment," *Cloud Computing*, 2010. [Online]. Available: [http://link.springer.com/chapter/10.1007/978-3-642-12636-9\\_4](http://link.springer.com/chapter/10.1007/978-3-642-12636-9_4)
10. Cloudfoundry. [Online]. Available: <http://www.cloudfoundry.com/>
11. [Online]. Available: <http://stratos.incubator.apache.org/>
12. D. Patterson, "A Simple Way to Estimate the Cost of Downtime." *LISA*, pp. 1–4, 2002. [Online]. Available: [http://static.usenix.org/event/lisa02/tech/full\\_papers/patterson/patterson.html/](http://static.usenix.org/event/lisa02/tech/full_papers/patterson/patterson.html/)
13. R. Charette, "Why software fails," *IEEE spectrum*, no. September 2005, pp. 42–49, 2005. [Online]. Available: <http://www.rose-hulman.edu/Users/faculty/young/OldFiles/CS-Courses/csse372/201310/Readings/WhySWFails-Charette.pdf>
14. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "A view of Cloud Computing," *Communications of the* ..., 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1721672>
15. C. Avizienis, A. and Laprie, J.-C. and Randell, B. and Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 010028, pp. 11–33, 2004. [Online]. Available: <http://www.malekinezhad.com/FOCD.pdf>
16. C. Landwehr and A. Bull, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys* ( ... ), pp. 1–36, 1994. [Online]. Available: <http://dl.acm.org/citation.cfm?id=185412>
17. D. Powell, "Failure mode assumptions and assumption coverage," *Fault-Tolerant Computing, 1992. FTCS-22. Digest of* ..., pp. 1–18, 1992. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=243562](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=243562)
18. A. Brown and D. Patterson, "To err is human," *Proceedings of the First Workshop on* ..., 2001. [Online]. Available: <http://roc.cs.berkeley.edu/talks/pdf/easy01.pdf>
19. M. Roesch, "Snort – Lightweight Intrusion Detection for Networks," *LISA*, 1999. [Online]. Available: [http://static.usenix.org/publications/library/proceedings/lisa99/full\\_papers/roesch/roesch.pdf](http://static.usenix.org/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf)
20. V. Stavridou, "Intrusion tolerant software architectures," ... *& Exposition II, 2001* ..., 2001. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=932175](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932175)
21. P. Verissimo, N. Neves, and M. Correia, "Intrusion-tolerant architectures: Concepts and design," *Architecting Dependable Systems*, vol. 11583, 2003. [Online]. Available: [http://link.springer.com/chapter/10.1007/3-540-45177-3\\_1](http://link.springer.com/chapter/10.1007/3-540-45177-3_1)

22. F. Schneider, "Implementing Fault-Tolerant Approach: A Tutorial Services Using the State Machine," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, 1990. [Online]. Available: <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Implementing+Fault-Tolerant+Approach+:+A+Tutorial+Services+Using+the+State+Machine#5>
23. M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002. [Online]. Available: <http://dl.acm.org/citation.cfm?id=571640>
24. G. Veronese, M. Correia, and A. Bessani, "Efficient Byzantine fault tolerance," pp. 1–15, 2013. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=6081855](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6081855)
25. T. Wood and E. Cecchet, "Disaster Recovery as a Cloud Service : Economic Benefits & Deployment Challenges," ...on *Hot Topics in Cloud* ..., 2010. [Online]. Available: [http://www.usenix.org/event/hotcloud10/tech/full\\_papers/Wood.pdf](http://www.usenix.org/event/hotcloud10/tech/full_papers/Wood.pdf)
26. G. Candea and A. Fox, "Recursive restartability: turning the reboot sledgehammer into a scalpel," *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, no. May, pp. 125–130, 2001. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=990072>
27. P. Sousa and A. Bessani, "Highly available intrusion-tolerant services with proactive-reactive recovery," *Parallel and ...*, vol. 21, no. 4, pp. 452–465, 2010. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=5010435](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5010435)
28. P. Mell, T. Bergeron, and D. Henning, "Creating a Patch and Vulnerability Management Program: Recommendations of the National Institute of Standards and Technology (NIST)," 2005. [Online]. Available: <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Creating+a+Patch+and+Vulnerability+Management>
29. U. Maheshwari, R. Vingralek, and W. Shapiro, "How to build a trusted database system on untrusted storage," ...on *Operating System Design & ...*, 2000. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251239>
30. H. Wang, P. Liu, and L. Li, "Evaluating the survivability of Intrusion Tolerant Database systems and the impact of intrusion detection deficiencies," *International Journal of Information and Computer ...*, vol. 1, no. 3, p. 315, 2007. [Online]. Available: <http://www.inderscience.com/link.php?id=13958>  
<http://inderscience.metapress.com/index/J126N3468876VJQ4.pdf>
31. A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," *Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05*, p. 163, 2005. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=1095810.1095826>
32. P. Liu, J. Jing, P. Luenam, and Y. Wang, "The design and implementation of a self-healing database system," ...*Information Systems*, vol. 23, no. 3, pp. 247–269, Nov. 2004. [Online]. Available: <http://link.springer.com/10.1023/B:JIIS.0000047394.02444.8d>  
<http://link.springer.com/article/10.1023/B:JIIS.0000047394.02444.8d>
33. D. Pilania, "Design, Implementation, and Evaluation of A Repairable Database Management System," *20th Annual Computer Security Applications Conference*, pp. 179–188. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1377228>
34. A. Brown and D. Patterson, "Rewind , Repair , Replay : Three R ' s to Dependability," *Proceedings of the 10th workshop on ACM ...*, pp. 1–4, 2002. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1133387>
35. A. B. Brown and D. A. Patterson, "Undo for Operators: Building an Undoable E-mail Store," 2003.
36. R. Chandra, T. Kim, and M. Shah, "Intrusion recovery for database-backed web applications," *Proceedings of the ...*, p. 101, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2043556.2043567>  
<http://dl.acm.org/citation.cfm?id=2043567>
37. I. E. Akkus and A. Goel, "Data recovery for web applications," *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pp. 81–90, Jun. 2010. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5544951>

38. S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 223, Dec. 2003. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1165389.945467>
39. T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion Recovery Using Selective Re-execution."
40. D. a. S. D. Oliveira, J. R. Crandall, G. Wassermann, S. Ye, S. F. Wu, Z. Su, and F. T. Chong, "Bezoar: Automated virtual machine-based full-system recovery from control-flow hijacking attacks," *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, pp. 121–128, 2008. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4575125>
41. a. Goel, D. Maier, and J. Walpole, "Forensix: A Robust, High-Performance Reconstruction System," *25th IEEE International Conference on Distributed Computing Systems Workshops*, pp. 155–162. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1437170>
42. F. Shafique, K. Po, and A. Goel, "Correlating multi-session attacks via replay," *Proc. of the Second Workshop on Hot ...*, 2006. [Online]. Available: [http://static.usenix.org/events/hotdep06/tech/prelim\\_papers/shafique/shafique.html/](http://static.usenix.org/events/hotdep06/tech/prelim_papers/shafique/shafique.html/)
43. T. Kim, R. Chandra, and N. Zeldovich, "Recovering from intrusions in distributed systems with DARE," *Proceedings of the Asia-Pacific Workshop on Systems - APSYS '12*, pp. 1–7, 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2349896.2349906>
44. N. Zhu and T.-c. Chiueh, "Design, Implementation, and Evaluation of Repairable File Service." *DSN*, 2003. [Online]. Available: <http://www.computer.org/comp/proceedings/dsn/2003/1952/00/19520217.pdf>
45. D. de Oliveira and J. Crandall, "ExecRecorder: VM-based full-system replay for attack analysis and system recovery," *...and system support for ...*, 2006. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1181320>
46. P. Ammann, S. Jajodia, and P. Liu, "Recovery from malicious transactions," *...Engineering, IEEE Transactions ...*, vol. 14, no. 5, pp. 1167–1185, 2002. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=1033782](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1033782)
47. M. Xie, H. Zhu, Y. Feng, and G. Hu, "Tracking and repairing damaged databases using before image table," *Frontier of Computer Science and ...*, 2008. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=4736507](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4736507)
48. M. Yu, P. Liu, and W. Zang, "Multi-version attack recovery for work-flow systems," *Computer Security Applications ...*, 2003. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=1254319](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1254319)
49. T. Kim, R. Chandra, and N. Zeldovich, "Efficient patch-based auditing for web application vulnerabilities," *...of the 10th USENIX conference on ...*, 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2387899>
50. X. Wang, N. Zeldovich, and M. F. Kaashoek, "Retroactive auditing," *Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11*, p. 1, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2103799.2103810>
51. R. Chandra, T. Kim, and N. Zeldovich, "Asynchronous intrusion recovery for interconnected web services," *Proceedings of the Twenty-Fourth ACM ...*, 2013. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2522725>
52. G. DeCandia, D. Hastorun, and M. Jampani, "Dynamo: Amazon's highly available key-value store," *SOSP*, pp. 205–220, 2007. [Online]. Available: <http://www.read.seas.harvard.edu/kohler/class/cs239-w08/decandia07dynamo.pdf>
53. W. Vogels, "Eventually consistent," *Communications of the ACM*, 2009. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1435432>
54. S. Chacon, *Pro Git*. Apress, 2009.
55. L. Vaquero and L. Roderio-Merino, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM ...*, vol. 39, no. 1, pp. 50–55, 2008. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1496100>

56. Xen. [Online]. Available: <http://www.xenserver.org/>
57. Kvm. [Online]. Available: <http://www.linux-kvm.org/>
58. openstack. [Online]. Available: <https://www.openstack.org/>
59. Amazon web services. [Online]. Available: <http://aws.amazon.com/>
60. Eucalyptus. [Online]. Available: <https://www.eucalyptus.com/>
61. R. Sumbaly, J. Kreps, and L. Gao, "Serving large-scale batch computed data with project voldemort," *Proceedings of the 10th ...*, 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2208479>
62. Linkedin. [Online]. Available: <http://linkedin.com>
63. A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1773922>
64. J. Williams and D. Wichers, "OWASP top 10-2013," *OWASP Foundation, April*, 2013. [Online]. Available: [https://www.owasp.org/index.php/Top\\_10\\_2013-Top\\_10](https://www.owasp.org/index.php/Top_10_2013-Top_10)
65. Spring. [Online]. Available: <http://spring.io/>
66. Stack overflow. [Online]. Available: <http://stackoverflow.com/>
67. Yahoo answers. [Online]. Available: <https://answers.yahoo.com/>
68. Google cloud platform. [Online]. Available: <https://cloud.google.com>