Shuttle: Intrusion Recovery for PaaS using Redo

Dário Nascimento
dario.nascimento@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Miguel Pupo Correia)

Abstract. Motivated by the increase number of intrusions and critical applications on Cloud, particularly on the emerging of its Platform as a Service (PaaS) model, and by the perspective of recovery from intrusions, we introduce a novel module for PaaS that gives system administrators the power of undo their cloud applications. While security research focus on prevention and distributed systems research on fault tolerance, we introduce a new module where security faults on PaaS' services are tolerated. The proposed architecture will remove any intrusion due to software flaws, corrupted user requests and support system corrective and preventive maintenance. This document surveys the existing redo techniques, identifies their limitations and propose the architecture to build a new module for redo on PaaS. A evaluation methodology is proposed in order to validate the proposed architecture.

1 Introduction

Platform as a Service (PaaS) is a cloud computing model for automated configuration and deployment of services. While the Infrastructure as a Service (IaaS) model became relevant to provide computation resources on demand [1] [2], PaaS intent to reduce the cost of scale software deployment and maintenance. It defines a container-like environment for execution and deployment of services in IaaS. Containers are machine images that provide solution stacks, e.g. Python+MySQL, where developers deploy their services for users. PaaS provides a set of programming and middleware services to support service design, implementation and maintenance on top of IaaS. For example, load-balancing according to the load of server, automatic server configuration and storage APIs. It ensure developers to have a well tested and an integrated deployment framework [3]. PaaS environments are provided either by cloud providers [4] [5][6] [7] or open-source projects [8] [9] [10]. PaaS market is developing fast and its shares will not be established by pure peak performance but by quality of service: correctness, security, consistency and recovery.

The number of critical and complex services in cloud environments, particularly using PaaS, is increasing rapidly. However, one of side effects of complexity and fast development is the number of security vulnerabilities. Since nowadays most of customers' and companies' critical and valuable information is located

in cloud, the service value is superior and vulnerability exploitation becomes more attractive and profitable, rising the risk of intrusion. These services have higher potential to have intrusion. When a vulnerability exploitation succeeds, intruders provoke system faults. Faults may cause system failures and consequent downtime which has significant business losses [11]. Recovery processes are needed to remove faults and restore the proper service. Intrusion recovery mechanisms can provide intrusion tolerance if they prevent catastrophic system failures [12] and avoid downtime of system.

Prevention and detection of malicious activity are the priorities of most security processes. However, preventing vulnerabilities by design is not enough because software has flaws due to complexity and budget/time constraints [13]. More, attackers can spend years developing new ingenious and unanticipated intrusions methods having access to what protects the system while the guardian has to predict new methods and solve attacks in few minutes to prevent intrusions. Redundancy and byzantine fault tolerance protocols are design for arbitrary faults but intrusions are intentional malicious faults designed by human attackers to leverage protocols and systems vulnerabilities. Attackers may compromise a replica quorum. If the attacker uses a valid user request, e.g. stolen credentials, the replication mechanism spread the corrupted data. The intrusion goal is to compromise the integrity of system and bring it down to repair.

Instead of trying to prevent every single intrusion, increasing the mean-time-to-failure (MTTF), we focus on recover when they happen, trying to tolerate them and reducing the mean-time-to-repair (MTTR). Data backup solutions roll back intrusions effects but require extensive administrator to re-execute legitimate actions. Our goal is to design, implement and evaluate *Shuttle* [1], a intrusion recovery system for PaaS using redo. Shuttle prevents information losses and recovers from intrusions on software domain due to software flaws and input mistakes [14]. Shuttle also supports system corrective and preventive maintenance. Since intrusions recovery requires extensive human intervention to restore the system state, we believe a system that provides automatic intrusion recovery removing the corrupted data and restoring systems integrity is a significant asset. The rapid and continuous decline in computation and storage costs in cloud platforms makes affordable to store user requests, create database checkpoints and redo user requests after software patches. Despite the time and computation demand for request re-execution, cloud pricing models provide the same cost for 1000 machines during 1 hour than 1 machine during 1000 hours [15] fica mal usar numeros?. Therefore system recovery is faster using more instances during redo process. Shuttle will provide intrusion tolerance by design of new modules for PaaS frameworks.

The remainder of the document is organized as follows: Section 2 brief the goals and expected results of our work; Section 3 presents the fundamental con-

[1] Shuttle stands for the traveling mean between present and previous contexts

cepts and survey on previous solutions; Section 4 describes briefly the architecture of PaaS frameworks and the proposed architecture for intrusion recovery system; Section 5 defines the scientific methodology which will be followed in order to validate the solution. Finally, Section 6 presents the schedule of future work and Section 7 concludes the document.

2 Goals

This work addresses the problem of provide an intrusion recovery system for services deploy in Platform as a Service. Our overall goal is to *make applications deployed in PaaS secure and operational despite intrusions*. More precisely, our system aims to help the administrators to recover from the follow faults:

- *Software Flaws:* Computing or database instances are compromised due to software vulnerabilities and require software patch to fix.
- *Corrupted requests (Input Mistakes):* Accidental or malicious user or administrator requests that perform undesired operations.
- *Unknown Intrusions:* The intrusion occurrent have not been detected.

Shuttle also *supports system corrective and preventive maintenance:* it supports software updates to prevent or remove intrusions and allow operators to try new configurations or software versions without affects on current service.

In order to archive these goals we plan to: posso manter o sujeito subentendido como se completasse a frase?

- *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 proposing a set of corrupted request from a externally provided list of malicious requests.
- *Support software update:* reflect automatically software updated on service's persistent state.
- *Warm-start and swiftly:* Recover the service without exposing downtime to user. However, corrupted states may be expose during the recovery phase.
- *Determinism, dependency and parallel execution:* Despite parallel execution of requests, same service application source code and requests mean equal result on re-execution.
- *Transparency:* Service behavior and performance characteristics during record phase are, as close as possible, similar to original service.
- *NO-SQL Database Snapshot:* Database checkpoint, without downtime, to reduce log size and speed up the recovery phase.
- *PaaS Integration:* The source code of the application remains identical. PaaS developers do not need to install or configure Shuttle. Shuttle is built in a generic manner and reused in each deployed service.

3 Related Work

In this section we survey relevant concepts and techniques for our work. Section 3.1 presents concepts of fault, intrusion and methods to treat them. Section 3.2 introduces the main intrusion recovery techniques. Finally, we describe a number of relevant proposals to recover in levels where PaaS services can be attacked: operating system, database and application.

3.1 Fault Characterisation

Dependability of a computing system is the ability to deliver service that can justifiably be trusted. "The service delivered by a system is its behavior as it is perceived by its users". Dependability encompasses the following attributes: *availability*: service readiness for authorised users; *confidentiality*: absence of unauthorised disclosure of information; *safety*: absence of catastrophic failures; *reliability:* continuity of correct service; *integrity* absence of improper system state [14], i.e. the state is correct or has beed repaired after intrusion. The dependability of the system can be affect by faults, errors and failures.

Faults are the cause of errors that may lead to system failures. System failures occur when errors reach the service and deviates the service to deliver an incorrect function. Faults are flaws on software or hardware domain that may have been introduce either accidentally or maliciously [16] during the system development or operation. Faults encompass input mistakes, software flaws and intrusions [14].

Intrusions are malicious faults resulting from vulnerability exploitation by human attackers which change system to erroneous state. In order to develop dependable system, delivering a resilient service, we need a combination of intrusion: forecast, prevention, detection, mitigation, tolerance, removal.

Intrusion prevention is realised by design and it seek to prevent future attackers from exploiting vulnerabilities. However preventing intrusions on 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 grant access to attackers[17]. 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 impossible to protect all vulnerabilities by design. Therefore vulnerabilities can be exploited successfully leading to an intrusion. Intrusion detection systems (IDS) attempt to detect intrusions by monitoring the system events [18]. Signature-based IDS match these input against a database of known attacks. Anomaly-based IDS look for statistic patterns. IDS turns the system attack-aware but not attack resilient.

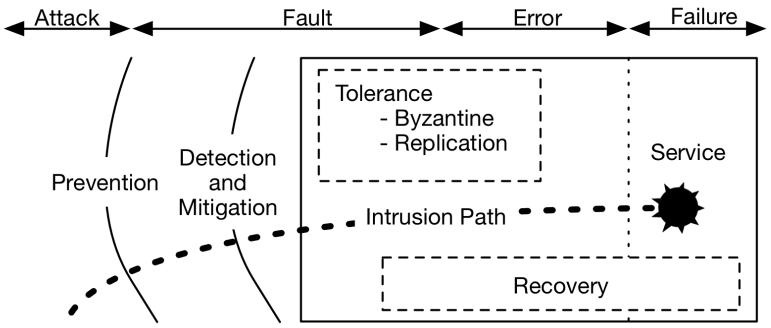

Fig. 1: Intrusion path inside systems

Intrusion tolerance, or *survivability*, is the last line of defence Fig. 1. Intrusion tolerance is the ability of a system to keep providing a, possibly degraded but adequate, service after penetration [19]. Instead of trying to prevent every single intrusion, these are allowed, but tolerated: the system trigger mechanisms that prevent the intrusion from generate a system failure [12]. Intrusion tolerance requires intrusion detection, containment, mitigation and recovery.
Some base techniques for intrusion tolerance involve fault tolerance concepts as redundancy, disaster recovery, recursive restartability and diversity. Redundancy and byzantine fault tolerant protocols [20] prevent a single compromised component from leading to an immediate system failure. Disaster recovery solutions [21] are a failover mechanism, a subtype of replication. It prevents from manmade or natural catastrophes using remote replicas. Diversity explores the orthogonality between system vulnerabilities to reduce the risk of common-mode failures. Recursive restartability [22][23] [24] [25] reboots replicated system components periodically to rejuvenate the replicas, restoring the soft-state and security assests, e.g. keys. However persistent data is still vulnerable and attackers can compromise all replicas before restart.
Fault tolerance methods are designed for arbitrary faults and they are too complex for cloud deployment [26]. Moreover, replication techniques facilitates the damage spread from one site to many sites as they do not distinguish between legitimate and malicious data. Intrusion are systematic and intentional malicious faults designed by human attackers to leverage design, deploy and operation vulnerabilities. Attackers can architect intrusions to compromise a quorum of replicas simultaneously. Unlike hardware failures, intrusions may not be detected immediately as attackers hide their attacks. We need another approach to avoid system failures. Intrusion tolerance can be archived if the intrusion is recovered before the error affects the service i.e. without exposing a downtime due to system failure.

3.2 Intrusion Recovery

Intrusion recovery removes all actions related to the attack, their effects on legitimate actions and return the service to correct function. Recovery methods concern to reduce the mean time to repair (MTTR) instead of fault avoidance to increase the mean time to fail (MTTF). Since inherent availability is MTTF/MTTR, reduce MTTR with recovery methods can improve the system availability considerably [27]. Also the downtime cost on web services [11] sustain the demand for efficient and effective intrusion recovery solutions.

Intrusion recovery techniques encompass the goals of intrusion removal during use and, if timely without exposing downtime, intrusion tolerance Fig.1. In order to recover from intrusion, we need to detect the intrusion, manage the vulnerability, remove the penetration effects and recover to a coherent state.

Some intrusion recovery methods require intrusion detection or corrupted dataset identification to remove penetration effects. Others revert the full-system state to a moment ahead of penetration or apply software patches. Since penetration effects spread during the period between penetration and intrusion detection, the delay of intrusion detection must be minimized. The anomaly threshold establish the trade-off between false positives and false negatives. For higher precision, the threshold must be high thus many intrusion cannot be identified. Yet, low threshold detects more suspicious behaviors but incur on false alarms which may result in legitimate data losses. Human intervention is needed to prevent false alarms so the detection can have a significant delay. IDS are out of document scope but we argue that recovery systems should support concurrent recovery to avoid system outages during false alarms and provide tools to help administrators to review the system behavior and determine which weaknesses were exploited.

After penetration or suspicious behavior detection, the system vulnerabilities must be managed. Vulnerabilities are identified, classified and mitigated [28]. Vulnerabilities are fixed by configuration adjustments or software patching, i.e. a piece of code developed to address a specific problem in an existing piece of software. These techniques are applied during system usage to prevent future intrusions.

Successful intrusions, named penetrations, affect the integrity of the system and/or its confidentiality. Recovery from confidentiality penetrations, i.e. unauthorized disclosure to restricted information, is out of the scope of this document. However, we argue that system design should encompass tolerance techniques to tolerate unauthorized data access reducing the data relevance with cryptography techniques protecting the data *secrecy* [29]. The penetration effects on the integrity of the system define the service availability, reliability and safety during the recovery period. The integrity and availability of the system is the sum of integrity and availability of each data entry and system component weighted by their importance [30]. Intrusion tolerant systems have minimal bounds for availability and integrity to provide a fair service. The following section introduces approaches to recover system integrity reverting the penetration effects, changing the system state to a coherent state and restoring the correct service.

3.3 Recovery Models

As discussed in previous section, intrusion recovery systems remove affected data items and restores the correct service. Here we specify this process formally using the notion of history and outline the distinct approaches for intrusion recovery.

O que acha? A history H models a system execution composed by a set of actions that operate on a set of objects. A subset $S_{checkpoint}(t)$ of a history H is a subsequence of actions executed after a checkpoint operation on instant t. The set of actions $S_{intrusion}$ from an attacker compromises the system during the intrusion period. We specify the subset S_{after}, where $S_{intrusion} \subseteq S_{after}$, as the subsequence of actions that occur after the intrusion begin. $S_{contaminated}$ define the subset of S_{after} affected, or *tainted*, by $S_{intrusion}$. If an attack is posteriori to t then $S_{after} \subseteq S_{checkpoint}(t)$ and the checkpoint is not affected by intrusion $S_{contaminated} \notin S_{checkpoint}(t)$. There are two distinct approaches to perform $H - S_{contaminated}$, i.e. remove the affects of intrusion from the system state: **Rewind** and **Forward** recovery.

Rewind recovery: [31] In rewind recovery, also known as roll-back, *all* system state is reverted to a error-free state recorded before the fault occurred. After, solely legitimate actions after checkpoint, $S_{legitimate} = S_{checkpoint} - S_{contaminated}$, are re-executed to restore the correct state. Any corrupted data entry in persistent state is restored. The checkpoint avoids to recovery from an empty state, i.e. replay every action in $H - S_{contaminated}$. Despite checkpointing, the re-execution of every legitimate action can be a long process if the number of user requests is high. The main difference between rewind and backup is that legitimate data items are recovered without extensive administrator interference.

Forward recovery: [**?**]Forward recovery, also known as selective undo [], backward[] or roll-forward[], takes the system state H at recovery time and removes the effects of contaminated operations on affected data entries until the entire system is clean and able to move forward. There are two main approaches to selectively undo the effects: *versioning* and *compensation*. *Compensation* [33] cover techniques which execute an inverted operations to revert the contaminated operations effects. *Versioning* encompasses technique that load previous versions of affected data set. Unlike rewind, the repair time is proportional to the number of contaminated operations but software patches are not supported as legitimate actions are not re-executed. The forward recovery technique figures out the corrupted data from $S_{contaminated}$ so unknown corrupted data is maintained.

Removing the intrusion affected actions is sufficient to archive intrusion recovery. However, legitimate actions dependent on compromised data can have a different execution since their arguments are restored. Therefore, some proposals [**?**] re-execute legitimate actions. We name this process as **redo** which occurs during the **replay phase**. To redo actions, recovery systems record the system actions and their dependencies during the normal system execution, or **record**

phase. Actions do not need to be idempotent but their re-execution must be deterministic.

The main performance metrics to evaluate intrusion recovery solutions are the follow: time, the intrusion should be recovered on acceptable period; space, the amount of space need to redo the actions should be minimized [34]. The qualitative metrics are: precision, from recovered entries, how many were corrupted; recall, from recovered entries, how many were corrupted. The values of qualitative metrics are determined by the system capacity to identify the affected items.

Propagation, known also as backtracking[35] or tainting [33], identifies data and actions, $S_{contaminated}$, that could have been affected by the initial attacking actions $S_{intrusion}$ using action records and dependence techniques. The dependencies can be established during service execution or on recovery time. These rules may be use to create a *dependency graph*. The nodes in the dependency graph represent objects such as files, data sets. The edges between indicates a dependency relation between objects. The level of abstraction influences the techniques used to extract dependencies. Low abstraction levels, as operating system, use system call dependency while higher do it between actions or user requests. The abstraction level outlines the recoverable attacks.
We present relevant works on abstractions levels where services deployed in PaaS can be attacked: operating system, database and application. Hardware attacks are out of scope. We will briefly describe each of these systems in the next paragraphs.

3.4 Recovery on Operating System Level

Intrusion recovery in Operating Systems (OS) is done according to its file system (FS). File systems are the persistent state of Operating Systems. By removing intrusions effects on FS, the recovery system has the best hope of revert the OS to a flawless state. Propagations mechanisms on operating system layer involve the dependency between system calls, files, sockets and processes. First, we introduce the concept of dependence in operating systems level. Then we present intrusion recovery systems based on rule-base and tainting propagation via replay. Finally proposals for recovery in clusters, virtual machines and network file systems.

BackTracker [35]: Backtracker proposes a tainting algorithm to track intrusions in Operating Systems. The algorithm requires an administrator to provide a set of compromised processes or files. Using dependencies between files and processes, Backtracker defines every file and process that could have compromise the provided files. The following rules establish the dependencies:

- *Dependencies between Process and Process* :
  - *New Process*: Processes forked from tainted parents are tainted.

    - *New thread*: Clone system calls to create new threads establish bi-directional dependences since threads share the same address space. Memory addresses tainting [36] has a significant overhead.
    - *Signaling*: Communication between processes establish dependency.
- *Dependencies Process/File*:
    - *File depends on Process:* If the process writes the file.
    - *Process depends on File:* If the process reads the file.
    - *File mapping into memory:* Same rule as new thread.
- *Process/Filename Dependency:*
    - *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.

Backtracker proposes a propagation algorithm to track the intrusion effects offline, after attack detection, as follows. First a dependency graph is initialized with a set of compromised objects identified by administrator. Then, Backtracker reads the log from the most recent entry until the intrusion moment. For each process, if the process is dependent upon a object, e.g. file, present on graph then remain objects dependent on the process are also added to graph.
The low abstraction level implies a big and complex dependency graph with many objects and processes. Also, objects shared between many processes,e.g. */tmp/* or */var/run/utmp* which tracks login/logout of users, are likely to produce false dependencies leading to false positives. False positives obfuscate the attacker's actions or lead to legitimate data lost. Therefore, Backtracker proposes a *white-list* filter that ignores common shared files. However, this technique rely on administrator knowledge and introduce false negatives if attackers perform on white-listed objects. Backtracker do not perform any proactive task to remove or recover from intrusions.

Taser [37]: Taser is an intrusion recovery system for operating systems which relies on Forensix [38] to log system interactions. It removes the intrusion effects from current state using forward-recovery with versioning. The dependence graph, which defines files to remove, is generated on recovery-time using rules similarly to Backtracker.
Taser relies on Forensix [38] to log system calls associated with files, processes and sockets. In order to determine the intrusion effects, Taser builds a dependency graph using a set of rules similar to rules of Backtracker. Since these rules result in large number of false dependencies, which mark legitimate objects as *taint*, Taser provides a white list mechanism and establish 4 optimistic policies that ignore some dependencies. 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 system.
The recovery phase is started with the set of corrupted objects determined by an administrator or an IDS. The provided set of objects can either be the source or the result of an attack. In the latter case, Taser, like Backtracker, transverses the dependency graph in reverse causality order to identify the set of attack source objects, named $S_{intrusion}$ that compromised the provided files. At the *propagation phase*, Taser transverses the dependency graph from the attack source set $S_{intrusion}$ and adds all contaminated objects to the set $S_{contaminated}$. Then it loads a previous version of marked files from a system snapshot and selectively replays the legitimate write operations that happen on affected files after the snapshot. Based on *selective undo* concept, legitimate files remain unchanged. However, read dependencies are not considered because Taser do not record the process level execution. Unlike *taint propagation via replay* technique in Retro [39], the processes are not re-executed. The recovery phase stops after reload the affected objects and re-execute write dependencies that happen after the snapshot. Goel et all.[40] propose an extension based on *taint propagation via replay* where applications are re-executed with a pre-tainted version of the file. If the output of the process during replay is different from the original output, then the dependent applications are also tainted.
Taser use the rules to, statically, determine the affected files and remove their effects. The dependency based exclusively on rules "can mistakenly mark legitimate operations as tainted, so data is lost".

Retro [39]: Retro is an intrusion recovery system designed for operating system. It provides the capability of remove the files affected by a set of identified attacking actions. Retro performs forward-recovery using versioning. It restores the corrupted files to a previous version and it re-executes actions dependent on them.
During record phase, the kernel module of Retro creates periodic checkpoints of file system and records system-wide interactions between objects and processes. The object definition encompasses TCP sessions, files, directories and user terminal. Every system call from executed processes are record with their input, output and object dependencies: accessed or modified objects. The dependence graph of Retro is defined by arguments and output objects of system calls (Fig. 2). This graph is finer-grained than Backtracker since dependencies are established per system call instead of per process.

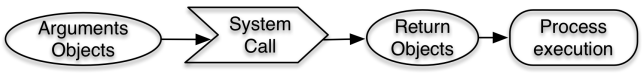

Fig. 2: Dependency graph due to system call in Retro

During the recovery phase, Retro requires the administrator to identify $S_{intrusion}$, processes or system calls which caused the intrusion. First, it removes system calls from the intrusion replacing them with null actions on graph. Retro rollback every compromised input and output objects of those system call to a previous version.The system calls, which corrupted objects depend from, are re-executed to update their values. Then, the processes, which invoke the system calls, are re-executed and the output is updated. Forward system calls are re-executed while their inputs are different from the original execution. Propagation is done by the output of system calls with different execution. This technique is known as *taint propagation via replay*. The recovery process terminates when propagation stops. Since Retro re-executes processes with system call granularity, the process re-execution may stop earlier. Since Retro do not checkpoint the state of processes, the system must be restart to remove non-persistent states and processes must be re-executed from beginning. This issue has a significant overhead specially on long-run processes as web servers e.g. Apache.
Since Retro re-executes 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 pty. 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.
The efficiency of Retro is result of log every system call input/output and avoid redo if the input is similar. However, storage intensive applications imply a significative CPU overhead and log size to track all system calls. Internal behaviour updates on processes, e.g. software patching, are not supported because Retro stops the propagation and skips the re-execution if the inputs to a process during repair are identical to execution during record phase. Retro requests the user to solve external inconsistencies on network and terminal. Retro can not recover from an intrusion whose log files have been garbage collected. The size of log can obfuscate the intrusion actions to administrator. The process of clean the false dependencies can be too long to provide a dependable system.

Dare [41]: Dare extends Retro to recover from intrusions in distributed systems. It adds the dependencies through sockets. The machines involved in a network session add socket objects to the dependency graph. Each socket object is globally identified by network protocol, source and destination IP and ports and one ID which is exchanged in every packages during connection.
The recovery phase begins when the administrator identifies the set of intrusion actions and initiates the recovery process in compromised machine. The local process is similar to Retro except on network system calls handlers. Compromised network sessions must be re-executed since their input depends on destination server. Therefore, prior to invoke the system call for network session establishment, Dare invokes a remote method on receiver Dare daemon to rollback the network session. The receiver rollback the dependent objects to the version before session establishment and re-executes 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.
Dare supports distributed and parallel re-execution but affected machines must be offline i.e. the propagation algorithm shutdown the service during the repair phase. Machine failures lock the recovery process. The solution is limited to clusters where every operating system runs a Dare and Retro daemon.

Bezoar [36]: Bezoar proposes an approach to recover from hijacking attacks in virtual machines (VM). The checkpoint is performed by VM forking using copy-on-write. This checkpoint technique encompass the entire system: processes and kernel spaces, resources, file system, virtual memory, CPU registers, virtual hard disk and memory of all virtual external devices. Bezoar tracks down how data from network propagates in memory. During recovery, Bezoar rewinds the VM to a checkpoint previous to penetration. Then it replays the system execution while ignoring all network packets from malicious sources. Any external change consequent from repairing actions is externalized.
The recovery process using rewind is longer than Retro or Taser because all external requests are re-executed. Since there are not mechanisms to track the penetration, the administrator must identify every intrusive network connection. Bezoar requires system outage during replay and do not provide any external consistency warranties.

Repairable File System (RFS) [42]: RFS is designed to recover compromised network file systems. The novelty on RFS comparing with the previous introduced systems is the approach client-server architectures. RFS is composed by a network file system (NFS) client module and a server NFS module.
The client's module tracks system calls using ExecRecorder [43] and establish the dependence between processes and NFS requests. Request to NFS server are marked with request ID and client ID. On 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 recovery phase, the server defines the contaminated processes using the client logs. A process is contaminated if match a set of rules similar to Backtracker. RFS introduces the notion 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 legitimate if reads a correct block from a contaminated file. RFS uses the version server to rollback only the affected files. Despite the strict dependency policies issues as Backtracker, RFS is resilient to log attacks. However, legitimate clients need to store their log.

Summary: Dependencies on operating system layer are established by rules based on BackTracker. These rules are vulnerable to false positives and false

negatives. While Taser and RFS recover from intrusion removing the effects only on corrupted files, Retro and Dare propagate the effects of read dependencies: it re-executes processes with read dependencies from changed objects.
Operating system layer solution are vulnerable to low-level attacks inside kernel because they only log system calls. Since log daemons are installed in the machine where attacks are performed, attackers can compromise the recovery system. The recovery guarantees are limited by system administrator capability to detect the attack and pinpoint the intrusion source. Administrators must identify the root cause on system logs in order to fix. However, the process of remove false dependencies from a big log is too long to provide a intrusion tolerant system.

3.5 Recovery on Database Level

A vast number of database management systems (DBMS) recover from faults loading a clean snapshot. However, this approach removes not only the malicious effects but also legitimate transactions after the snapshot. Recovery approach on database is analogous to operating system. Dependencies are establish between transactions. Compromised transactions are determined from a initial set of bad transactions using read-write rules. "Transaction T_j is dependent upon transaction T_i if there is a data item x such that T_j reads x"[44] 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" [44]. 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 if corrupted transactions did not exist e.g. a corrupted transaction removes a row and the next transaction issues a transaction that would read that row and write other dependent data. Xie et all. [45] propose to keep a copy of deleted rows in a separated table to track the transaction which deleted the rows. Dependencies are determined issuing the query on original and delete track tables to add the relation with transaction that deleted rows which would be affected by present query otherwise. professor, dou aqui um detalhe extra de dependencia que e ja um bocado redo e entro em detalhes de implementacao, devo ser mais sucito removendo os detalhes de implementacao?
The dependency rules require to extract the read and write set of each transaction i.e. the set of entries that each transaction access or modifies. The following proposals use different methods to extract them.

ITDB [44][46][30]: Intrusion Tolerant Database (ITDB) is an intrusion recovery system for databases. It performs forward-recovery with compensation: compromised data entries are restored to the latest legitimate value using compensation transactions. ITDB uses the set dependency rules described above. The rules require to extract the read and write set of transactions. Since most of SQL DBMS only keep write logs, Liu et all. propose a pre-defined per-transaction type template to extract the read set of parsed SQL statement. This approach is application dependent since updates on application queries require updates in

their templates.
Liu et all. propose an *online recovery* service. First, it reads the log in forward direction, i.e. from intrusion moment to present. The write sets from bad transactions are stored in a *dirty set*. For each good transactions, ITDB keeps the write set until transaction commits or aborts; if the transaction commits after read some entry that belongs to the *dirty set*, then every entry in the write set of transaction is also considered in the set dirty and the transaction must be undone. Repaired entries are tracked to prevent compensation of later transactions from restore a repaired data entry to its version after the attack.
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 access to dirty data entries. Since identification of dirty entries requires log analysis, Liu et all. proposes a *multi-phase damage container* technique to avoid damage spread through new requests during the recovery phase. This damage confinement method denies read accesses to records, which have been modified after intrusion, and aborts active transactions which read these records. Although it speeds-up the recovery phase, it also confines undamaged objects and decreases the system availability during recovery period. The throughput asymmetry between recovery and user flows can be neutralized increasing the priority of repair request in the proxy. The availability is not compromised anymore but users may read corrupted data and propagate the damage slower.
ITDB architecture includes an intrusion detection system. The IDS is application aware and acts on the transaction level. Liu et all. propose isolation in terms of users: when a suspicious transaction is reported by IDS, every write operation from the suspicious user is done within isolated tables.
Since ITDB do not re-execute read dependent transactions, it ignores the fact that legitimate executions can be influenced by the attack indirectly [45]. Liu et all. solution lacks on re-execution of compromised transactions after compensation. They propose a theoretical model based on possible workflows and versioning. Their solution supports concurrent recovery but it aborts new transactions that read compromised data items and incurs on degraded performance.

Phoenix [47] : Phoenix is a forward-recovery system that uses versioning to remove the intrusion effects. While Liu et all. [44] rely on statement templates and read the log in recovery time, Phoenix changes the DBMS code to extract read dependencies and proposes a run-time algorithm to check dependencies between transactions.
In order to restore a previous version and track dependencies, Phoenix records every update to a row in a new row version and stores the transaction ID (TID) in a row field. Transactions read the latest row version written by a transaction that did not abort. Phoenix modifies the PostgreSQL DMBS code to intercept read queries during their execution and extract the TID of each accessed row. The logged data is used to update the dependency graph.
The recovery process is splinted into two phases: propagation and rollback. On

propagation phase, Phoenix identifies the set of affected transactions from a root set of corrupted transaction using the built dependency graph. During rollback, affected transactions are changed to *abort* on transaction status log of PostgreSQL instead of transaction compensation as Piu et All. Since PostgreSQL exposes only the version of the latest non-aborted transaction, the row is restored and the effect of corrupted transactions is removed. Phoenix do not include a checkpoint mechanism so the recover period is limited by storage size.

Summary: In conclusion, recovery systems for SQL database differ on methods to track the read and write sets and to restore previous values. Phoenix relies on PostgreSQL execution to obtain read information and generate a runtime dependency graph. On other hand, ITDB tracks the read requests information from SQL statements and it uses a read and a write log to undo the effects. Phoenix is more Barga2004accurate but it incur on bigger overhead during normal operation.
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. More, an attack can compromise just an independent part of the transaction and legitimate data is undo.

3.6 Recovery on Application Level

Most part of current web applications, like the services deployed in PaaS environments, are build on separated tiers: computation and storage. Following works assume database as persistent layer. While previous section works establish dependency between transactions using their read-write sets, they ignore the transaction dependency that occurs on application level. For example, an application can read a record A through transaction 1, compute a new value B based on A and write the value B through transaction 2. Worst, applications may keep internal state or communicate using a different method than database. However, computation layer is often state-less, requests are independent and database is the only mean of communication between requests (Section 4.1).

Data Recovery for Web Applications [33] : Goel et all propose a recovery system that selectively removes intrusion effects on web applications which store their persistent data in a database tier. Since each user request may involve multiple transactions, it captures dependencies on user, session, request, program, row and field levels. The proposal uses a tainting algorithm and compensation transactions. It writes previous values of tainted rows to remove effects of contaminated operations.
During record phase, the read and write set of each transaction is logged. The read set is extracted using a SQL query parser. Every transaction is mapped to origin request.
The recovery phase splits into 2 phases: tainting and undo. The tainting process

runs in an offline sandbox. Professor, nem a tese explica se o sandbox e feito com uma replicacao da base de dados ou copy-on-write, refiro isso ou ignoro? First, the administrator identifies malicious transaction or requests. Then, every transaction posteriori to intrusion moment is compensated to create a snapshot. The dependence graph between transactions is determined using database logs [44]. The graph defines the maximal set of dependent requests. Goel et All modify the PHP-interpreter to reduce the false dependencies between transactions: variables that read tainted rows or fields are also tainted; rows or fields written by tainted variables are tainted. Like *taint propagation via replay* in Retro [39], the requests that read tainted entries are also re-executed. However, it only aims to determine the set of tainted entries and their correct value in an isolated database. The undo phase restores every tainted entry in the current database using compensation transactions in reverse serialization order.
The legitimate operations do not reflect undo operations on INSERT or DELETE statements [45]. The proposal does not perform a redo phase since it assumes "that the damage caused by the tainted requests is small compared to the work done by the untainted requests". However, undo every operation after intrusion to create a snapshot has an equivalent overhead. The overhead of keep versions and checkpoints is avoided but transaction compensation implies a longer recovery phase. False positives are avoided using the taint propagation mechanism. Finally, it propose a white list and optimistic dependency mechanism to prevent false positives. Like in Taser, these mechanisms can generate false negatives which keep intrusion effects.

Poirot [48]: Poirot is a system 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 do not recover from intrusions but proposes a tainting algorithm for application code. During 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 system is patched and Poirot identifies the changed code blocks and requests dependent on it during normal execution. The affected requests are re-executed. During the re-execution phase, each function invocation is forked into 2 threads: the patched e non-patched version [49]. 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.

Warp [50]: Warp is patch based intrusion recovery for single server web applications backed by SQL databases. Unlike previous approaches, Warp allows administrators to retroactively apply security patches without track down the source of intrusion and supports attacks on user browser level. Warp is based on Retro's [39] rollback and re-execute approach. Kim et All. propose a prototype based on PHP and PostegreSQL. During normal execution, Warp records all JavaScript events that occur during each page visit. 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 on browser level. On server side, Warp records every requests received and forwards the request to the PHP runtime. During request dispatch, Warp records non-deterministic functions and a list of invoked source code files. Warp stores database queries input/output and tracks the accessed table partitions using a query parser. At end, the HTTP response is logged and packed with all execution records.
During repair phase, the administrator fix the source code flaws and provides the list of modified files. Given the patch, Warp computes the set of requests that executed those files [48] [49]. These requests are the root cause of changes during re-execution. Each request is re-executed using a server-side browser to perform the same user original actions. The repair server intercepts non-deterministic function calls during replay and return the original logged value. Also, database read queries are re-executed only if the set of affect rows is different or its content was modified as result of either rollback or re-execution. Write queries are re-executed rolling-back the affected rows to a previous version and replaying the query. Each row,which have a different result after re-execution, is taint and all requests dependent on the row are also re-executed on browser level. Warp executes *taint propagation via replay*. Finally, the HTTP response is compared with original and any difference is logged. If responses are different, the following dependent user interactions on browser are replayed and all conflicts are queued and handled by users later.
Warp proposes a time-travel versioned SQL database. Each row is identified using a row ID and includes a *start time* and *end time* columns that establish its valid period using a wall clock timestamp. This allows selective rollback and read previous rows versions. However clock synchronization limits the usage of distributed databases.
Warp support concurrent repair using 2 integer columns to define the *repair generation*. During repair, the current generation ID incremented to fork the database. User requests are perform on current generation while recovery requests are perform on next generation. After repair the requests from the old generation, the server stops to apply the requests issued after the repair process begin.
Despite concurrent recovery and time-travel versioning, Warp database do not support checkpoint. Therefore, Warp requires to keep record of every logs that occur during the recoverable period. Moreover, the query re-write approach and 5 extra columns per table is a considerable overhead. Since Warp do not support foreign keys, it is not transparent on database. It also depends on client browser extension to log and modify every request. Warp is designed for single machine applications: it do not support multi-application server architectures nether distributed databases since versions are timestamp based. Finally, Warp do not account operating system attacks e.g. a shell access through a web site form vulnerability.

Aire [51] : Aire is an intrusion recovery system for loosely coupled web services. It extends the concept of local recovery by Warp [50] tracking attacks across services. While Dare [41] aims to recover a server cluster synchronously using Retro [39] on operating system, Aire performs recovery on asynchronous third-party services using Warp. It supports downtime on remote servers without locking the recovery. To archive this goal, pendent repair requests are queued until the remote server is recovered. Since clients see partial repaired states, 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*. 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 [50], during normal execution, Aire records 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 dawn the recovery process. Second, it undo the local database rows that might have been affected by attack requests. Unlike Warp, Aire do not delete past actions, it creates a new branch instead. Third, Aire uses the log to identify the affected queries and re-executes their requests. This process is similar to Warp but past incorrect external request or responses are detected. To replace either a request or a response, Aire sends a request to source or destination to replace it and starts a local repair process on remote server. If the remote server is offline, the repair request is queued. Since servers can start a remote recovery process to continue the global system recovery process, clients may see an inconsistent state. The distributed algorithm will start successive repairs and converge.
Aire approach using eventual consistency targets a specific kind of application that solve conflicts. Moreover, Aire recover third-party web-services which must have an Aire daemon. Aire require the administrator to pinpoint the corrupted requests. Finally, Aire, as Dare [41], requires system stop during recovery process which is equivalent to service failure.

Undo for Operators[55]: Undo for Operators allows operators to recover from their mistakes, software problems or data corruption on Email Services with file system storage. The design is base on "Three R's: Rewind, Repair and Replay" [31] where operator loads a system-wide snapshot previous to intrusion, repairs the software flaws and replays the user-level requests to roll-forward. As opposed to selective redo, the rewind approach removes any corrupted data without identification. Undo for Operators propose a proxy interposed between the application and its users. The proxy intercepts service requests. This architecture supports upgrade or replacement of operating system or application on computation layer to fix software flaws. However, the proposed architecture is

protocol dependent: the proxy supports IMAP e SMTP. Undo Operators define the concept of *verb*. Each protocol operation has its own *verb* class. Verbs are objects to encapsulate the user interactions and expose a generic interface for undo system.
During 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 on operation and arguments. Thus, the dependency mechanism is application dependent. For example, email send (SMTP protocol) are commutable and independent because their order is not relevant on email delivery. On the other hand, the order of delete (Expunge) and List (Fetch) on 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 establish a serialization ordering but it has a significant performance overhead on concurrently-arriving interactions and requires protocol knowledge.
The recovery phase process as follows. First the operator determines the corrupted verbs and fix their order adding, deleting or changing verbs. Second the system is rewind, i.e. a system-wide snapshot is loaded to remove any corrupted data. Third the operator patch the software flaws of SO or application. Finally, the requests are redo to rebuild the system state.
External inconsistencies may come out during the redo phase. These inconsistencies are detected comparing the re-execution and the original responses. Different responses trigger a compensation action to keep a external consistent state. Again, these actions are application dependent.
Since full system is rewind to a previous state, Undo for Operators do not support concurrent repair and user request execution. Moreover, it relies on protocol knowledge to establish dependencies and compensation actions. Therefore, any protocol change requires change the supported verb set. Intrusions can use corrupted requests which are not encompassed on known verb classes and cause a system fail.

Summary: Goel et all. and Warp establish dependencies using the request read-write set on database and use taint via redo. Brown establish dependencies using the knowledge about protocol operations. Unlike Goel et all., Warp and Brown et all. support application repair. While Goel et all. ignores external consistence, Warp detects inconsistencies on responses and replays the user interaction on browser and Brown et.all use compensation actions based on protocol-specific knowledge.

3.7 Evaluation of Technologies

The previous sections describes various systems that use different approaches to recover from intrusions. While lower level recovery systems incur of bigger overhead and may obfuscate the attack, application level solutions are semantically rich but do not track lower level intrusions. The level of abstraction defines the

record data: Operating System (OS) solutions are based on system calls and file system; Database (DB) solutions track inter-transaction dependencies; Application (App) level solution track transactions, requests and execution code.
As shown in the table 1, most of solutions require administrator to pinpoint the initial corrupted set of actions using an external mean e.g IDS. The remain solutions track the requests which invoked modified code files. While pinpoint actions incurs on false positives, track the invoked code files requires changes to the interpreter [**?**].

System	**Recover**	**Record**			**Intrusion Detection**	**Undo**	**Taint via Replay**	**Online Recovery**	**Externally Consistent**
		Sys. Call	Transaction	Request					
[37]Taser	OS	✓			Actions	Versioning			
[39][41] Retro/Dare	OS	✓			Actions	Versioning	✓		✓
[46] ITDB	DB		✓		Actions	Compensation		✓	
[47] Phoenix	DB		✓		Actions	Versioning		✓	
[33] Goel et All.	App		✓	✓	Actions	Compensation	✓		
[50][51] Warp/Aire	App		✓	✓	Code	Versioning	✓	✓	✓
[55] Brown et All.	App		✓	✓	Code	Checkpoint			✓

Table 1: Summary of current recovery systems

The described solutions varie according to *How to remove effects of intrusions?* and *How to recover a legitimate state?* (Table 2). Rewind solutions remove all effects loading a snapshot. Brown [55] loads a snapshot created by a checkpoint. The system must stop to perform a request consistent checkpoint i.e., all requests on checkpoint are completed ver como e que ele faz isso. Goel [37] create a snapshot compensating every action. This approach incurs on less storage overhead. However, the snapshot loading process during the recovery phase is longer and proportional to the number of requests after the snapshot. It only reverts direct effects of logged actions therefore effects of actions, which were not registered, remain on storage.
Forward recovery solutions keep previous versions of data entries or apply actions to revert their values. The first approach requires bigger storage overhead but it has data entry granularity instead of transaction. Tenho de rever como e que as transaction base fazem compensacao apenas de parte da query.

The recover method, i.e. which data entries are affected and have to be changed, is distinct (Table 2. Forward recovery systems [37] [47] [44] [50] [39]

How recovers? \ How removes?		**Rewind (Snapshot)**		**Forward**	
		Checkpoint	Compensation	Versioning	Compensation
Rewind	Redo All	Brown			
	Taint and update		Goel		
Forward	Remove Only			Taser/Phoenix	IDTB
	Taint Propagation			Warp	Retro

Table 2: Methods to remove intrusion effects and to recover legitimate states

undo only affected entries. Therefore, the application must track the dependencies between requests and undo their actions. Dependency rules are a major challenge. They can be established using database read-write [**?**], the invoked code files [**?**] or tainting [37]. On database level, the rules do not track the dependencies on application so they are imprecise. On one hand, strict rules incur on false negatives therefore legitimate data may be removed. On the other hand, optimistic rules can be explored and hide intrusions. Moreover, legitimate operations are not affected because *insert* or *delete* rows during normal execution or recovery phase are not detected. Despite Xie et all. [45] technique to detect deleted rows, remove only solutions [**?**] do not update legitimate actions which were affected. Detect which queries are affected by a row insert during the recovery phase requires to re-execute every query that encompass the modified table. In order to support software changes and read dependencies, the affected requests need to be re-executed while their input change: taint propagation via replay [**?**].
Rewind solutions [50] [55] do not incur on legitimate data losses since every request is re-executed and any dependencies are detected. Goel re-executes every request in a sandbox and update only the modified data entries in the current database. Brown re-executes every request on current database. However, re-execute every request is expensive and requires compensation mechanisms to keep external consistency.

The proposals [46][**?**][50][51] support online recovery where recovery phase do not require system downtime. This characteristic is required to archive intrusion tolerance however it allows intrusion to spread during the recovery period, except using multiphase containment [46] which compromises the availability during recovery period. The

The values externalized are different after the recovery process. The intrusion effects are removed. Brown et all [55] have an extensive study on this area [**?**]. They propose application dependent actions to compensate from recovery inconsistencies. The remain projects ignore or notify the administrator of externalized differences. The solutions of distributed systems Aire and Dare perform repair propagation invoking the local recovery methods.

4 Architecture Design

The following sections describe the proposed architecture. It starts by an overview of Platform as a service, followed by the details of the design and, at the end, the design choices are discussed.

4.1 Platform as a Service

Platform as a Service (PaaS) is a cloud computing model for automated configuration and deployment of services. It provides an abstracted and well-tested environment where developers can deploy their service applications written in high-level languages, e.g. Python, without concern with lower-level details. PaaS is designed in a modular and server-oriented manner providing a well-defined interface to services, e.g. data storage, that ca be provided by a third-party or easily updated.

PaaS deployment units are bare-metal servers and/or virtualized operating systems running on hypervisors (Xen [**?**] or KVM [**?**]) which can be managed by an Infrastructure as a Service framework (Openstack [**?**], AWS EC2 [**?**], Eucaliptus [**?**]). FALTA REFERIR AQUI O ISOLAMENTO
Os principais componentes sao: Load Balancer: Encaminha os pedidos dos utilizadores para os servidores baseado na disponibilidade de cada servidor. Node Controller: Por nó, monitoriza o estado do node e reporta, gere a configuracao do nó, inicia e tear-down de novas instancias. Cloud Controller: Gere os nós de computacao inicializando novos nós. Auto-scaling: Scaling decisions based on monitorization data Monitoring: Recebe os dados de cada node para saber a ocupacao do sistema. Application Instances: Onde as aplicacoes correm de modo isolado. Database Node: Pode ser um nó único (MySQL), um load balancer e um set de slaves ou só um set de slaves. Authentication Manager: Provide user and system authentication

Os módulos podem estar contidos na mesma máquina. O load balancer recebe os pedidos e encaminha-os para os Application Instances. Os

4.2 Architecture

5 Evaluation

The evaluation of the proposed architecture will be taken experimentally, building a prototype. This evaluation aims to verify the system and prove its scalability on large implementations. We evaluate our proposal by how effective and how efficient it is. In detail, we intend to measure the follow:

rever *Expected results* Our solution should help to recover from 5 of OWASP Top 10 Application Security Risks: Injection Flaws, Broken Authentication,Security Misconfiguration, Missing Function Level Access Control and Using Components

with vulnerabilities [56]. Shuttle should archive the proposed goals and its results should match evaluation metrics bounds that allows Shuttle usage on real environments. Section 5 presents further details.

- **Recording:** We want to quantify the *recording overhead* imposed measuring the *delay* per request and system *throughput*, *resource utilisation* 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.
- **Redo Metrics:** We aim to measure the effectiveness by *precision*, from recovered database entries, how many were required - equivalent to false negatives, and *recall*, from entries that required change, how many were changed, equivalent to false positives. We plan to efficiency measuring the recover time with and without stop the system and check if these values are usable on enterprise solutions.
- **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 system survivability.
- **Concurrency:** We want claim that our solution provides the same results as original execution if there are no changes on final state if the requests and system code remain identical even supporting parallel and concurrent requests.
- **Recovery Performance:** We target to analyze performance optimizations due to parallel requests. Recovery time depends on request and checkpointing frequency.

Estamos interessados em saber o false positive rate, true detection rate, detection latency and survivability

The expected performance metrics bounds,e.g. time to recovery, are highly dependable on attack type, damage, frequency, delay to detect and time the repair speed, the arrival rate of new requests and the required system integrity to archive survivability during the recovery process. We will evaluate our solution with different workloads.
We will develop a demo application that will be deployed on PaaS and a multi-threaded and multi-host program where each thread acts as a real-world user loading the application. For each user, we will programmatically measure response time, throughout and data consistency comparing the Shuttle's overhead. The prototype evaluation is branched in Private Cloud and Public Cloud. Due to public cloud costs, we will implement a test prototype on private cloud and evaluate the final prototype on public cloud. On private cloud we will survey the follow technologies:

- **Hypervisor:** Xen Server or KVM;
- **IaaS:** Openstack or Eucalyptus;
- **PaaS:** AppScale, Openshift and Apache Stratos;
- **Database:** HBase, Cassandra or Voldemort;

The complete evaluation will be performed on Amazon Web Service IaaS to overcome limited resources. This evaluation will share the same PaaS and Databases and scale to medium enterprise size.
Qualitative tests will be done with intrusion scenarios proposed on related works to compare Shuttle effectiveness against previous solutions.

6 Schedule of Future Work

Future work is scheduled as follows:

- January 16 - Fev 15: Write Project Draft and deploy AppStack, Openstack and OpenShift on local cloud
- Fev 15 - Fev 28: Project Report Done
- March 1- March 15: Solution Deployment on local cloud and testing
- March 15 - April 10: Solution Deployment on public cloud (AWS), experimental evaluation of the results.
- April 10 - April 30: Write a paper describing the project
- May 1 - June 15: Finnish the writing of the dissertation
- Jun 15, 2014: Deliver the MsC dissertation

The solution architecture will be implemented and evaluated in phases, named sprints. The first sprint will design the auditors, checkpoint database and redo. The sprint goal will be a basic system without PaaS and concurrency issues to prove the concept and extract new issues. The next sprint will deploy this basic system on PaaS. The later sprint will handle request parallelization and consistency. The latest sprint is focus on optional goals as request changes and simple user interface. More details about schedule are available in Appendix A

7 Conclusion

In this report we have surveyed the most relevant techniques to recover from intrusions. First we introduced the concepts of dependability, faults, intrusion prevention and tolerance. Then we discussed the intrusion recovery goals and approach.
Several system were presented to illustrate the current state of the art of intrusion recovery for operating systems, databases and multi-tier applications. Conclusoes disso Finally, we survey the Platform as a Service (PaaS) architecture.
We proposed a design for Shuttle, a intrusion recovery system for PaaS, that aims to make PaaS deployed applications secure and operational despite intrusions. Shuttle recovers from software flaws, corrupted requests and unknown intrusions. It also support system corrective and preventive maintenance. The major challenges in this design are to implement a concurrent and consistent database checkpoint and establish accurate requests dependencies, contain the damage spreading and repair fast to avoid critical levels of integrity.

Shuttle will remove intrusion effects on service instances, help to track affected requests from a set of externally provided list of malicious requests, support software patching and finally concurrent rollback of database tier and legitime request replay. We will propose, to the best of our knowledge, the first intrusion recovery using Redo for PaaS and the first concurrent recovery solution on distributed key-value database.

Acknowledgments I'm 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).

8 Appendix

A Detailed Schedule

Aqui vou detalhar mais sobre o meu calendario com objectivos mais detalhados

B Platform as a Service Architecture

Aqui vou colocar uma imagem grande com a arquitectura de um PaaS porque acho relevante, acredito que nem todos os leitores vao ter uma ideia nitida de como e a arquitectura do PaaS

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. L. Vaquero, L. Rodero-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
4. Amazon elastic beanstalk. [Online]. Available: http://aws.amazon.com/pt/elasticbeanstalk/
5. Google app engine. [Online]. Available: https://developers.google.com/appengine/
6. [Online]. Available: https://www.heroku.com/
7. [Online]. Available: https://www.openshift.com/
8. 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
9. [Online]. Available: http://www.cloudfoundry.com/
10. [Online]. Available: http://stratos.incubator.apache.org/
11. 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/
12. P. Veríssimo, 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
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-Classes/csse372/201310/Readings/WhySWFails-Charette.pdf
14. A. Avizienis, J. Laprie, and B. Randell, *Fundamental concepts of dependability*, 2001. [Online]. Available: http://www.malekinezhad.com/FOCD.pdf
15. 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
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. 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
18. 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
19. V. Stavridou, "Intrusion tolerant software architectures," *... & Exposition II, 2001 ...*, 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932175

20. 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
21. 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
22. 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
23. G. Candea and A. Fox, "Designing for high availability and measurability," *Proc. of the 1st Workshop on Evaluating and . . .*, no. July, 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.2535&rep=rep1&type=pdf
24. C. Studies, D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, K. Emre, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," pp. 1–16, 2002.
25. A. Bessani, P. Sousa, and M. Correia, "Intrusion-tolerant protection for critical infrastructures," 2007. [Online]. Available: http://docs.di.fc.ul.pt/handle/10455/3054
26. A. Bessani, M. Correia, and P. Verissimo, "Intrusion Tolerance: The "killer app" for BFT Protocols (?)," *homepages.gsd.inesc-id.pt*, pp. 1–3, 1999. [Online]. Available: http://homepages.gsd.inesc-id.pt/ mpc/pubs/position-bft.pdf
27. A. Brown and D. Patterson, "Embracing failure: A case for recovery-oriented computing (roc)," *High Performance Transaction Processing . . .*, pp. 3–8, 2001. [Online]. Available: http://www.csd.uwo.ca/courses/CS9843b/papers/autonomic_ROC.pdf
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+Mana
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. 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
32. "John Knight Willow."
33. 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
34. F. Cornelis and A. Georges, "A taxonomy of execution replay systems," *. . . on Advances in . . .*, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.9129&rep=rep1&type=pdf

35. 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
36. 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
37. 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
38. 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
39. T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion Recovery Using Selective Re-execution."
40. 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/
41. 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
42. 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
43. 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
44. 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
45. 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
46. 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
47. 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
48. 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

49. 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
50. 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
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. A. B. Brown and D. A. Patterson, "Undo for Operators: Building an Undoable E-mail Store," 2003.
56. 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