Recovery from Security Intrusions in Cloud Computing

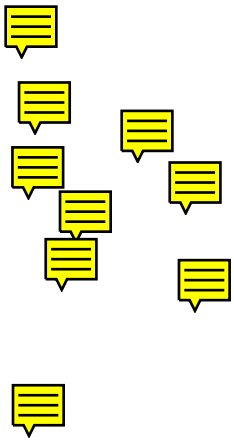
Dário Nascimento
dario.nascimento@tecnico.ulisboa.pt

Instituto Superior Técnico
Advisor: Professor Miguel Pupo Correia

Abstract. We introduce a novel service for PaaS systems that gives system administrators the power to recover its applications. The motivation for this work is the increasing number of intrusions and critical applications in Cloud, particularly on the emerging of its Platform as a Service (PaaS) model and the perspective of recovery from security intrusions. While security research focus on prevention and distributed systems research on fault tolerance, we introduce a new service where security faults 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 the cloud computing model for automated configuration and deployment of applications. While the Infrastructure as a Service (IaaS) model became relevant to provide computation resources on demand [1,2], PaaS intents to reduce the cost of scale software deployment and maintenance. It defines an environment for execution and deployment of applications in *containers*. Containers provide software stacks e.g., Python+MySQL, where developers deploy their applications for users. Containers are bare-metal machines, virtual machines or Linux cgroups [3]. PaaS provides a set of programming and middleware services to support application design, implementation and maintenance. For example, load-balancing, automatic server configuration and storage APIs. It ensure developers to have a well tested and an integrated deployment framework [4]. PaaS environments are provided either by cloud providers [5,6,7,8] or open-source projects [9,10,11]. PaaS systems are developing fast and their success will not be established not only by pure peak performance but also by qualities of their services: correctness, security, consistency and recovery.



The number of critical and complex applications in cloud environments, particularly using PaaS, is increasing rapidly. However, since nowadays most of

customers' and companies' critical applications and valuable information is migrating to cloud environments, the value of deployed applications is superior and vulnerability exploitations are more attractive and profitable, rising the risk of intrusion. When a vulnerability exploitation succeeds, intruders provoke system faults. Faults may cause system failure and, consequent, downtime that has significant business losses [12]. Recovery solutions are needed to remove faults and restore the proper behavior. They may also provide fault tolerance if they prevent system failures without exposing system downtime [13].

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 [14]. More, attackers can spend years developing new ingenious and unanticipated attack methods having access to what protects the application while the guardian has to predict new methods and solve attacks in few minutes to prevent intrusions and mitigate vulnerabilities. On other hand, 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. 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.

Instead of trying to prevent every single intrusion to increase the mean-time-to-failure(MTTF), we focus on recover when they happen. We try to tolerate them and reduce the mean-time-to-repair(MTTR). Data backup solutions roll back intrusion effects but require extensive administrator effort to re-execute legitimate actions. Our goal is to design, implement and evaluate *Shuttle* [1], a intrusion recovery service for PaaS. Shuttle prevents information losses and recovers from intrusions on 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 compromise their availability. Confidentiality flaws are out of work scope. - deve ficar aqui? Since intrusions recovery normally requires extensive human intervention to restore the application state, we believe a service that provides intrusion recovery without exposing a downtime, removing the corrupted data and restoring application integrity is a significant asset.

The rapid and continuous decline in computation and storage costs in cloud platforms makes affordable to store user requests, use database checkpoints and re-execute previous user requests to recover from intrusions. 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

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

[15]. Shuttle leverage the resources scalability in PaaS environments to record and re-execute user requests. Shuttle uses clean PaaS containers images to renew the application containers and re-executes user requests in parallel to recover a consistent application persistent state removing any suspicious intrusion. The container image may include software updates to fix previous flaws. Shuttle only requires a valid input record and a intrusion-free container to recover a consistent persistent state. Shuttle will provide intrusion recovery by design of a new service for PaaS systems. This service will be available for PaaS' customers usage without installation, configuration and their applications source code will remain identical.

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 concepts 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 providing an intrusion recovery system for applications deployed 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 following faults:

- *Software Flaws:* Computing or database containers are compromised due to software vulnerabilities and require software patches to fix.
- *Corrupted requests and/or data values by malicious or accidental input mistakes:* 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 also *supports system corrective and preventive maintenance* since it supports software updates to prevent or remove intrusions and allow operators to try new configurations or software versions without affects in the application behavior perceived by users.

In order to archive these goals we plan to:

- *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:* Reflect software updated in the application persistent state.
– *Recovery without stopping the application:* Recover the application without exposing users to application downtime. However, corrupted states may be expose during the recovery phase.
– *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:* Database checkpoint, without downtime, t 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 application.

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 for recover in the levels where PaaS applications are attacked: operating system, database and application.

3.1 Fault Characterization

Dependability of a computing system, which is built by the application and respective deployment environment, is the ability to deliver service that can justifiably be trusted. The service delivered by a computing system is its behavior as it is perceived by its users. 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 i.e., the state is correct or has been repaired after intrusion. Three of the core concepts in service dependability are: fault, error and failure.

Faults are remote events e.g., configuration mistake or unauthorized data modification. Activated faults leads to errors. *Errors* are inconsistent parts of system state. System *failures* occur when an error is exposed by the service and deviates it from deliver the system specification function. Faults are flaws in software or hardware domain that have been introduce either *accidentally* or *maliciously* during the system development, production or operation phases

[16,17]. In the context of this work we consider human-made faults, which are accidental, deliberate non-malicious or malicious, that occur development, production and operation. In particular, we target the design faults due to software flaws and interaction faults from input mistakes, attacks and intrusions [17].

Intrusions are malicious faults resulting from vulnerability exploitation designed by human attackers. Intrusion can omissive, suspending a system component, and/or assertive, changing a component to deliver a service with a not specified format or meaning [18]. In order to develop dependable system, delivering a resilient service, we need a combination of intrusion forecast, prevention, detection, mitigation, tolerance, recovery (Fig. 1).

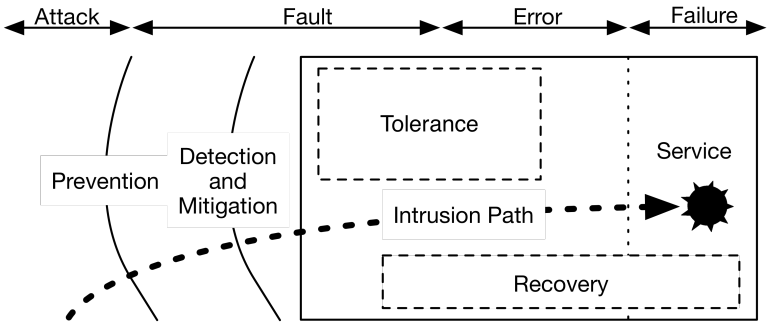


Fig. 1: Intrusion path across the system

Intrusion forecast and prevention is realized by design and it 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 [14,16]. Also system administrators, as humans, can make security configuration mistakes or users may grant access to attackers [19]. 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.

Intrusion detection and mitigation monitors the system events to detect suspicious actions that may be connected with an intrusion. Intrusion detection systems (IDS) [20] are signature-based, which match system events against a database of known attacks events, or anomaly-based, which look for anomalies in statistic patterns. However, Intrusion detection systems turns the system attack-aware but not attack resilient. Intrusion detection systems may enable mitigation techniques e.g., block the suspiciousTCP session, but attackers may

perform the intrusion before intrusion detection systems detects it.

Intrusion tolerance, or *survivability*, 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 penetration [21]. Instead of trying to prevent every single intrusion, these are allowed, but tolerated: the system has mechanisms that prevent the intrusion from generate a system failure [13]. Intrusion tolerance hide errors effects using redundancy mechanisms or detecting, processing and recovering faults.

The Intrusion tolerance redundancy mechanisms involve classic fault tolerance concepts. Classic redundancy and Byzantine fault tolerant protocols are based on state replication mechanisms [22,23,24] and prevent a single compromised component from leading to an immediate system failure. In Disaster recovery solutions [25], the state is replicated to a remote site to recover from catastrophes. Replicas can be implemented in different manner to create diversity i.e, explore the orthogonality between the vulnerabilities of different system implementations to reduce the risk of common failures. Replicas of system components may also be rebooted periodically to rejuvenate and restore their soft-state and security assets e.g., keys, creating a proactive recovery mechanism. [26,27,28,29,30]

The above classic fault tolerance methods are designed for arbitrary faults. However intrusion are systematic and intentional malicious faults designed by human attackers to leverage design, deploy and operation vulnerabilities. Most of fault tolerance mechanisms do not prevent intentional faults from attacks to software flaws or at application level e.g., using valid user requests. Moreover, state replication mechanisms facilitates the damage spread from one site to many sites as they copy data without distinguish between legitimate and malicious sources. Even when replicas are updated after a defined period or to a remote site to provide a safety backup, the damage will spread if not detected timely. Most of proactive recovery techniques rejuvenate only the application soft-state but intrusions can also affect the persistent state. Therefore, intrusions may transverse these mechanisms and cause a system failure. The alternative approach for intrusion tolerance is to detect, process and recover the intrusion. Intrusion tolerance is also 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 application to correct behavior. It aims 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, reducing MTTR with recovery methods can improve the application availability considerably [31]. Intrusion recovery techniques archive the goals of intrusion removal and, if timely without exposing downtime, intrusion tolerance. repete o final do ultimo paragrafo para tentar ligar mas fica um pouco isolado... In order

to recover from intrusion and recover the application to a consistent behavior, intrusion recovery solutions detect the intrusion, manage the exploited vulnerability and remove the intrusion effects changing the persistent state to the state that would be produced without intrusion.

The first phase of intrusion recovery concerns the intrusion detection. The detection delay should be minimized because intrusion effects spread during intrusion achievement and detection. Automated intrusion detection systems (IDS) are used to detect intrusions or suspicious behaviors. However their threshold must be carefully selected. The threshold establish the trade-off between false positives and false negatives. High thresholds initiate less intrusion recovery mechanisms but many intrusions may not be identified. On other hand, low threshold detects more suspicious behaviors but incur on false alarms which may result in legitimate data losses. Since human intervention may be needed to prevent false alarms, the detection phase can have a significant delay. IDS are out of document scope but we argue that recovery systems should support the execution of recovery processes in background without externalization to users. The recovery process should avoid downtime during false alarms. Intrusion recovery solutions also should provide tools to help administrators to review the system behavior and determine which weaknesses were exploited. Some recovery solutions also support preventive intrusion recovery where system flaws are detected but its exploitation is not detected.

After intrusion or system flaws detection, the system vulnerabilities must be managed. Vulnerabilities are identified, classified and mitigated [32]. 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 also applied during system usage to prevent future intrusions.

The latest phase is remove the intrusion effects. Intrusions affect the system integrity and/or its confidentiality. Recover from the effects of confidentiality intrusions 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* [33]. Recover from integrity intrusions is necessary to change the application to a correct behavior. The application availability, reliability and safety is defined by its integrity. 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 [34]. Different system have different minimal bounds for availability and integrity to provide a fair service during intrusion recovery. The following section introduces approaches to recover system integrity reverting the intrusion effects, changing the system state to a coherent state and restoring the correct behavior.

3.3 The Recovery Process

As discussed in previous section, intrusion recovery solutions revert the intrusion effects and restores the system to a correct behavior. Here we specify this

process formally using the notion of history and outline the distinct approaches for intrusion recovery.

A history H models a system execution composed by a set of actions. A generic set of actions A_x writes a set of objects D_x. A history H operates a set of objects D. We specify $A_{intrusion}$ as the set of actions whereby the attacker compromises the system during the intrusion, A_{after} as the sequence of actions that occur after the intrusion begin and A_{legal} as the set of legitimate actions after intrusion. Notice that $A_{after} = A_{intrusion} \cup A_{legal}$. A simple intrusion recover system, like backup, aims to obtain the set of objects associated with the set legitimate of actions C where $C = H - A_{intrusion} : H \cap A_{after} = \emptyset$ i.e., get a set of objects without intrusion and any action that occur after the intrusion. More advanced systems can perform $C = H - A_{intrusion} : A_{after} \subseteq H$ i.e., remove the values of objects written by actions of intrusion and keep actions performed after intrusion. We define $A_{tainted} : A_{tainted} \subseteq A_{legal}$ as the subset of legitimate actions from A_{legal} that access objects in the set $D_{intrusion} \cup D_{tainted}$, i.e. objects written by intrusion actions or actions which have been tainted. Therefore, $D_{legal} \neq D - D_{intrusion} \wedge D_{legal} \neq D - (D_{intrusion} \cup D_{tainted})$ i.e., remove the objects written by intrusions and tainted actions is necessary but not enough to obtain the set of objects that would be produced by the subset of legal actions A_{legal}. The set of objects D_{legal} is obtained from the set of actions $C = H - A_{intrusion}$ where $A_{tainted}$ have a different execution since is required that $D_{tainted} = \emptyset \wedge D_{intrusion} = \emptyset$. Therefore, we prove that intrusion recovery system require process, named **redo**, where every action from set C is re-executed to obtain a legitimate object set that excludes the effects of intrusion.

The set of actions prior to intrusion, $C = H - A_{intrusion}$, can include an extensive number of actions. Each action takes a variable but not null time to perform. Therefore the redo process takes an excessive amount of time. We define the subsets $D_{checkpoint}(t) \subseteq D$ and $A_{checkpoint}(t) : A_{checkpoint} \subseteq H$ as the subsets of object values and actions executed before the begin of a checkpoint operation at instant t. The checkpoint operation copies the value of the object immediately or on the next write operation. If an attack is posteriori to t then $A_{after} \cap A_{checkpoint} = \emptyset \implies (A_{intrusion} \cup A_{tainted}) \cap A_{checkpoint}(t) = \emptyset$ i.e., the checkpoint is not affected by intrusion. The system can redo only $H - A_{checkpoint} - A_{intrusion}$ using the object set $D_{checkpoint}$ as base. A **snapshot** is a checkpoint that includes every action and object in system before the instant t. A **version** is a checkpoint of a single object value and the actions that operated on it before the instant t. Defining $A_{posteriori}(t)$ as the set of actions after instant t, a snapshot or version can be obtained performing a **compensation** process where the complement of every action after t is performed in inverse order.

Recovery systems need to record the system actions during the normal system execution, or **record phase**. In order to perform redo, system action do not need to be idempotent but their re-execution must be deterministic. The record

phase should record the value of every non-deterministic behavior to turn their re-execution into a deterministic process.

Since objects are shared between actions, we can establish dependencies between actions which can be visualized as a graph. The nodes of *action dependency graph* represent actions and the edges indicate dependencies though shared objects. The **object dependency graph** establish the dependency between objects through actions. Dependency graphs are used to order actions re-execution[35], get the set of actions affected by a object value change [36], get the set of actions *tainted* by intrusion attacking actions [37] or resolve the set of objects and actions that caused the intrusion using a set of known *tainted* objects [38]. 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 dependencies extraction method. The abstraction level outlines the recoverable attacks. In the next paragraphs, we will briefly describe 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

Intrusion recovery in an Operating System is done according to its persistent state, the file system. By removing intrusions effects in file systems, recovery systems have the best hope of revert the Operating System to a flawless state. System calls, files, sockets and processes establish the dependencies at Operating System layer. First, we introduce the concept of dependence at Operating System 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 [38]: 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. Backtracker uses dependencies between files and processes to resolve every file and process that could have compromise the provided files. The following rules establish these 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 [39] 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 graph is initialized with a set of compromised objects identified by administrator. Then, Backtracker reads the log of actions from the most recent entry until the intrusion moment. For each process, if the process is dependent from a file or process currently present in the graph then the remain objects dependent from the process are also added to graph.

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 actions of attackers 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 [40]: Taser removes the intrusion effects from current state loading *selectively* a legitimate version of tainted files from a file system snapshot and performing only the correct actions. Taser relies on Forensix [41] to log system interactions and generates a dependence graph, which defines files to remove, at recovery-time using rules similarly to Backtracker.

Taser relies on Forensix [41] to log the names and all the arguments of system call operations related to process management, file system operations and networking. 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 not only a white list mechanism but also establishes four 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"[40]. However, attackers can leverage these optimistic policies to penetrate the Operating 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 $A_{intrusion}$, that compromised the provided files. After, at *propagation phase*, Taser transverses the dependency graph from the attack source set $A_{intrusion}$ and adds all tainted objects to the set $D_{tainted}$. Taser defines a *selective redo* process where it selectively replays legitimate operations

only on tainted objects. It loads an initial version of the tainted objects from the file-system snapshot and sequentially replays the legitimate modification operations of tainted objects since the snapshot. Non-tainted files remain unchanged. Since Taser use rules to determine the affected files and remove their effects, it can mistakenly mark legitimate operations as tainted and induce to legitimate data losses. More, Taser do not update the objects dependent from tainted objects. Therefore, Goel et al.[42] 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. However, long running applications take a long time to re-execute. aqui nao detalho imenso sobre taint propagation via replay porque e um trabalho extensao e a ideia esta mais bem explicada no Retro, que apresento abaixo... ha problema?

Retro [43]: Retro provides the capability of removing files affected by a set of identified attacking actions. It restores the corrupted files to a previous version from a file system checkpoint. Retro defines the *taint propagation via replay.*

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 not only files and directories but also TCP sessions and user terminal. System calls of executed processes are recorded 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.

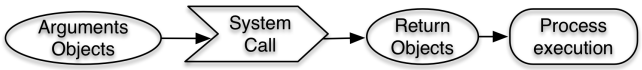


Fig. 2: Dependency graph due to system call in Retro

During the recovery phase, Retro requires the administrator to identify $A_{intrusion}$, processes or system calls which caused the intrusion. First, it removes the intrusion system calls from the graph replacing them with null actions. 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. Retro uses the *taint propagation via replay* method where forward system calls are re-executed 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 re-executes processes with system call granularity, the process re-execution may stop earlier. However, 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.

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 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. The later work, *Dare* [44], 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 is similar 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.

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 behavior 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. 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 by attackers or operator mistakes. The size of log can obfuscate the intrusion actions to administrators. The process of cleaning the false dependencies can be too long to provide a dependable system. Dare supports distributed 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.

Bezoar [39]: 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 how the data from network connections propagates in memory. Bezoar do not

provide mechanisms to track intrusions therefore the administrator must identify every intrusive network connection. During recovery, Bezoar rewinds the VM to a checkpoint previous to penetration. Then, it replays the system execution ignoring all network packets from identified 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. Bezoar requires system outage during replay and do not provide any external consistency warranties.

Repairable File System (RFS) [45]: 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 module tracks system calls using ExecRecorder [46] 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 propagate the effects of read dependencies. However, propagate the effects based on input changes requires to store the input of every action.

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, due to low abstraction level, the dependency graph and log are too big and complex to remove false dependencies promptly and provide a intrusion tolerant system. deveria pegar nas criticas que vou fazendo no final de cada trabalho e mover para aqui?

3.5 Recovery at 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"[47] 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" [47]. 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. Since user mistakes are often deletes because wrong query arguments, this is a relevant issue. Xie *et al.* [48] propose to keep a copy of deleted rows in a separated database table to track the transaction which deleted the rows. Dependencies are determined issuing the query on original and delete tracking tables to add the dependency between the transaction and the deleted rows which would be affected.
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 and restoring the previous values.

ITDB [47][49][34]: Intrusion Tolerant Database (ITDB) performs intrusion recovery in databases using compensation: compromised data entries are restored to the latest legitimate value using compensation transactions. ITDB uses the set of dependency rules described above. These rules require to extract the read and write set of transactions. Since most of SQL DBMS only keep write logs, Liu *et al.* 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 al.* propose an *online recovery* service, i.e., the application remains available during the recovery process. First, it reads the log from intrusion moment to present. It stores the write sets of bad transactions 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 using inverse operation. 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 al.* 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. However it 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 al.* 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 leave out the fact that legitimate executions can be influenced by the removed values [48]. Liu *et al.* propose a theoretical model based on possible workflows and versioning. However, predict every possible workflow requires extensive computational and storage resources. Their solution supports concurrent recovery but it aborts new transactions that read compromised data items thus incurs on degraded performance.

Phoenix [50] : Phoenix removes the intrusion effects using a versioned database. While Liu *et al.* [47] rely on SQL statement templates 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.

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 al.*. 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: 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 incur on bigger overhead during normal operation but it is query independent. While Phoenix requires to keep every version in memory, ITDB requires to define an inverse transaction to get the correct version.

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 at 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 [37] : Goel *et al.* propose a recovery system that selectively removes intrusion effects on web applications which store their persistent data in a sql database. 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 and 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. First, the administrator identifies malicious transaction or requests. Then, every transaction posteriori to intrusion moment is compensated to create a clean database snapshot. The dependence graph between transactions is determined using database logs [47]. The graph defines the maximal set of dependent requests. Goel *et al.* 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 [43], the requests that read tainted entries are also re-executed. This process determines the set of tainted entries in an isolated database. After, during the undo phase, the system 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 [48]. The usage of compensation transactions avoids the overhead of keeping versions and checkpoints but it implies a longer recovery phase and the knowledge of inverse transaction. 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 [51]: 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 [52]. 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 [36]: 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 [43] rollback and re-execute approach. Kim *et al.* propose a prototype based on PHP and PostgreSQL.

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 [51] [52]. 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 has a different result after re-execution, is taint and all requests dependent on the row are also re-executed at browser level i.e., Warp uses *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 five 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 [53] : Aire is an intrusion recovery system for loosely coupled web services. It extends the concept of local recovery by Warp [36] tracking attacks across services. While Dare [44] aims to recover a server cluster synchronously using Retro [43] 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 [54] [55]. 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 [56] and re-applies legitimate changes. At end of local repair, Aire moves the current branch pointer to the repaired branch.

Like Warp [36], 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 [44], requires system stop during recovery process which is equivalent to service failure.

Undo for Operators[35]: 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" [57] 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 exposes an interface that establish the order between requests and their dependencies.

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, compensation actions and sort the re-execution. 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 al.* 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 al.*, Warp and Brown *et al.* support application repair. While Goel *et al.* 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.

The described solutions constrast according to *How to remove effects of intrusions?* and *How to recover a legitimate state?* (Table 2). Rewind solutions [35]

System	Recover	Record			Intrusion Detection	Undo	Taint via Replay	Online Recovery	Externally Consistent
		Sys. Call	Transaction	Request					
[40]Taser	OS	✓			Actions	Versioning			
[43][44] Retro/Dare	OS	✓			Actions	Versioning	✓		✓
[49] ITDB	DB		✓		Actions	Compensation		✓	
[50] Phoenix	DB		✓		Actions	Versioning			
[37] Goel *et al.*	App		✓	✓	Actions	Compensation	✓		
[36][53] Warp/Aire	App		✓	✓	Code	Versioning	✓	✓	✓
[35] Brown *et al.*	App		✓	✓	Code	Checkpoint			✓

Table 1: Summary of current recovery systems

remove all effects loading a snapshot loading a previous snapshot.The system must stop to perform a request consistent checkpoint i.e., all requests on checkpoint are completed. Goel [40] 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.

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

The recover method i.e., which data entries are affected and have to be changed, is distinct (Table 2. Forward recovery systems [40] [50] [47] [36] [43] 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 [40]. 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 al.* [48] 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 [36] [35] 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 [49][**?**][36][53] 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 [49] which compromises the availability during recovery period. The

The values externalized are different after the recovery process. The intrusion effects are removed. Brown et al. [35] 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.

Propagation, known also as backtracking[38] or tainting [37], 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.

Rewind recovery: [57] 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* [37] 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.

4 Architecture

The following sections describe the proposed architecture. It starts by an overview of a generic Platform as a service architecture, followed by the design choices for Shuttle.

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, generic and well-tested set of programming and middleware services where developers can design, implement, deploy and maintain their service 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 a pay-per-usage extremely scalable software infrastructure.

PaaS tenant application deployment units are the containers. Containers are performance and/or functionality isolated environments. Containers are bare-metal servers, virtualized operating systems running on hypervisors e.g., Xen[59], KVM[60], or lightweight in-kernel accounting and isolation of resources usage

using Linux cgroups [3]. These containers are managed directly or through a IaaS framework (e.g., openstack [61], AWS EC2 [62], Eucalyptus [63]). The tenant applications deployed in PaaS are designed to scale horizontally. Each container is isolated and distributed states are avoided. The application state is maintained by database or session cookies.

A base PaaS framework is composed by the following components:

- **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**: Retrieves the metering data from each node. The Load balancer uses this information to perform requests routing while the cloud controller automatically decides when to scale.
- **Containers:** The isolated environment where tenant applications runs.
- **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:** Provide user and system authentication.

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

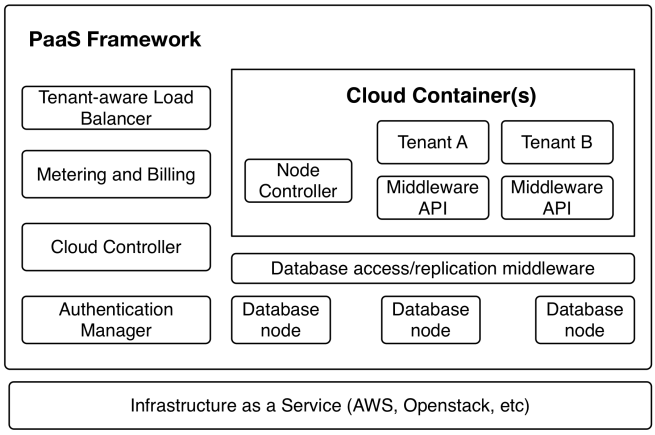


Fig. 3: Generic Architecture of PaaS

4.2 Shuttle

While previous work aim to perform intrusion in applications with a single database, Shuttle aims to recover intrusions in PaaS applications. Since typical

PaaS applications are designed to support high usage loads, our main contribution is its scalability and transparency for application developers. Shuttle logs and re-executes user requests to recover from intrusions.

The PaaS system including **Shuttle** is composed of following modules (Figure 4): devo colocar os modulos antes da ligacao entre eles ou depois? estou um pouco perdido aqui

- **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 that spreads the requests through each web server according to their usage.
- **Web Servers:** The web servers are the HTTP application servers that serve user requests. Each application uses the database library that contains the **database client auditor**.
- **Database Servers:** The database servers run the database software that stores the application persistent state. Each server contains a **database server auditor** that logs the requests dependency at database level.
- **Request Storage:** The user requests and redo metadata is stored in a scalable database.
- **Redo Nodes:** The redo nodes are HTTP clients that read previous requests from the requests storage and send them to the web servers.
- **Manager:** The manager is the coordinator of Shuttle.

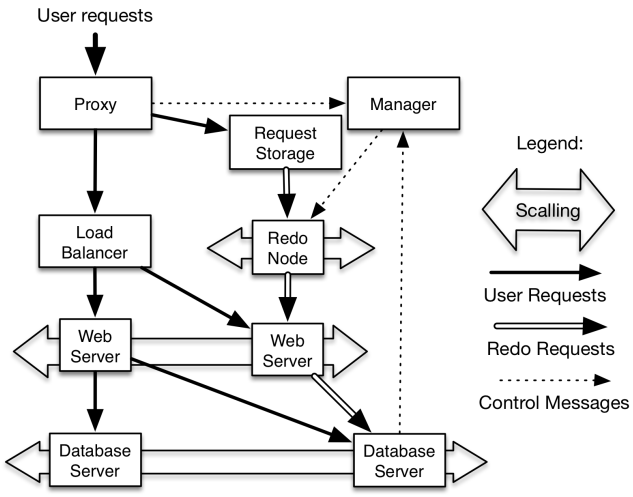


Fig. 4: Overview of the proposed system

Shuttle will perform a selectively redo the affected data entries

target take a rewind approach where the system is restored to a clean state and every request is re-executed. Since PaaS applications are oriented to high frequency of requests, our architecture supports parallel recovery and reduces the recovery time using the dependency between requests. Brown et. all [35] rely on application knowledge to establish the dependency between requests. Our work aims to support any application deployed on PaaS therefore we rely on database dependencies during the original execution [47] to establish the dependency between requests.

manager It coordinates the distributed database snapshot, collects the dependency information from auditors and orders the redo nodes to re-execute the necessary requests.

During the execution phase, the *Request Auditor* logs every every user request in memory. The *Database Auditor* logs the database invocations from the computation tier. The logged data si periodically flushed the data to the *Log Analyser*. The system administrator setups the *Checkpoint Manager* to perform period distributed snapshot of the NoSQL database.

The redo phase is initialised when intrusions, suspicious or mistakenly behaviours are detected or the application software requires an update. The set of malicious requests, if any, is identified. These requests are removed from the original request sequence. Then the *Redo Coordinator* loads the snapshot on each database node and new PaaS instances are initiated with the code fixed. The new system is intrusion-less because the snapshot and instances are clean and the code is fixed. The requests are re-executed in parallel in a different database version. Since users still use the corrupted computation instances and database, the system remains online with a depreciated behaviour but without exposing a downtime. After recovery, the users requests are switched to the state fixed system.

One of the main challenges is to provide a distributed snapshot consistent with your computation level transactions. Only completed requests can be in the snapshot otherwise the redo phase will be inconsistent. The other challenge is to reduce the recovery time to acceptable values. We explore the fact that PaaS applications have loosely couple requests that share data only by database requests. Therefore, we use the database dependencies to compute parallel sequence of executions. Dependencies between requests are not trivial, particularly when their execution is modified by code or request ordering corrections.

The elected database to backend the application will be a Key-Value NoSQL database based on Dynamo DB [54]. These database are common on PaaS because they scale properly and provide a simple put,get,delete API. This API avoids the delay of parsing SQL queries.

NoSQL Key-Value storage Voldemort, which is an opensource implementation of Amazon Dynamo DB [54] by Linkedin, and the users requests will be stored in the NoSQL database Cassandra.

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. 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 [64]. Shuttle should archive the proposed goals and its results should match evaluation metrics bounds that allows Shuttle usage on real environments. We evaluate our proposal by how effective and how efficient it is, i.e the false positive and false negative rates, the recovery latency and the survivability. In detail, we intend to measure the follow:

- **Recording:** We want to quantify the *recording overhead* imposed measuring the *delay* per request and system *throughput*, *resource 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 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 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 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. We will check how it can improve the performance of redo process.
- **Cost:** We will measure the monetary cost of intrusion recovery using public cloud providers.

The expected performance metrics bounds, like the time to recovery, are highly dependable on: attack type, damage and frequency; delay to detect and repair time; request arrival rate; system integrity required to archive survivability during the recovery process.

We will develop a demo web application that will be deployed on PaaS. Since independent user sessions would be trivial to re-execute, the application has highly dependent interactions between requests from different users. The application is a Java Spring[**?**] implementation of Question and Answering (QA) system, like Stack Overflow [65] and Yahoo! Answers [66]. Spring is one of most used Java enterprise web frameworks and it is compatible with most of current PaaS systems. To highlight the Shuttle scalability, the application will store its state in the modified NoSQL database.

We will develop a 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 it will measure response time and throughput of each user. To evaluate the effectiveness, the tool will analyze the consistency of application responses during the recovery phase. These results will be extracted from intrusion scenarios similar to the ones proposed in the related works. These scenarios aim to compare Shuttle effectiveness against previous solutions. 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 private cloud and evaluate the final prototype on public cloud. First, our prototype will be deployed in a PaaS system provided by AppScale [9] and OpenShift [8]. These opensource PaaS solutions will run over Openstack. Later, we will perform the final evaluation running the system over Amazon Web Service IaaS or Google Cloud Platform to overcome our limited resources and scale to medium enterprise size scenarios.

6 Schedule of Future Work

Future work is scheduled as follows:

- January 16 - Feb 15: Write Project Draft and deploy AppScale [9] and OpenShift [8] in a local Openstack [61] cloud
- Feb 28 - May 4: Project Report Done
- March 1 - June 1: Solution Implementation
- Jun 20 - Jul 15: Solution Deployment on local cloud and testing
- Jul 15 - Aug 15: Solution Deployment on public cloud (AWS), experimental evaluation of the results.
- April 20 - Jul 10: Write a paper describing the project
- Jul 10 - Set 15: Finnish the writing of the dissertation
- Set 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.

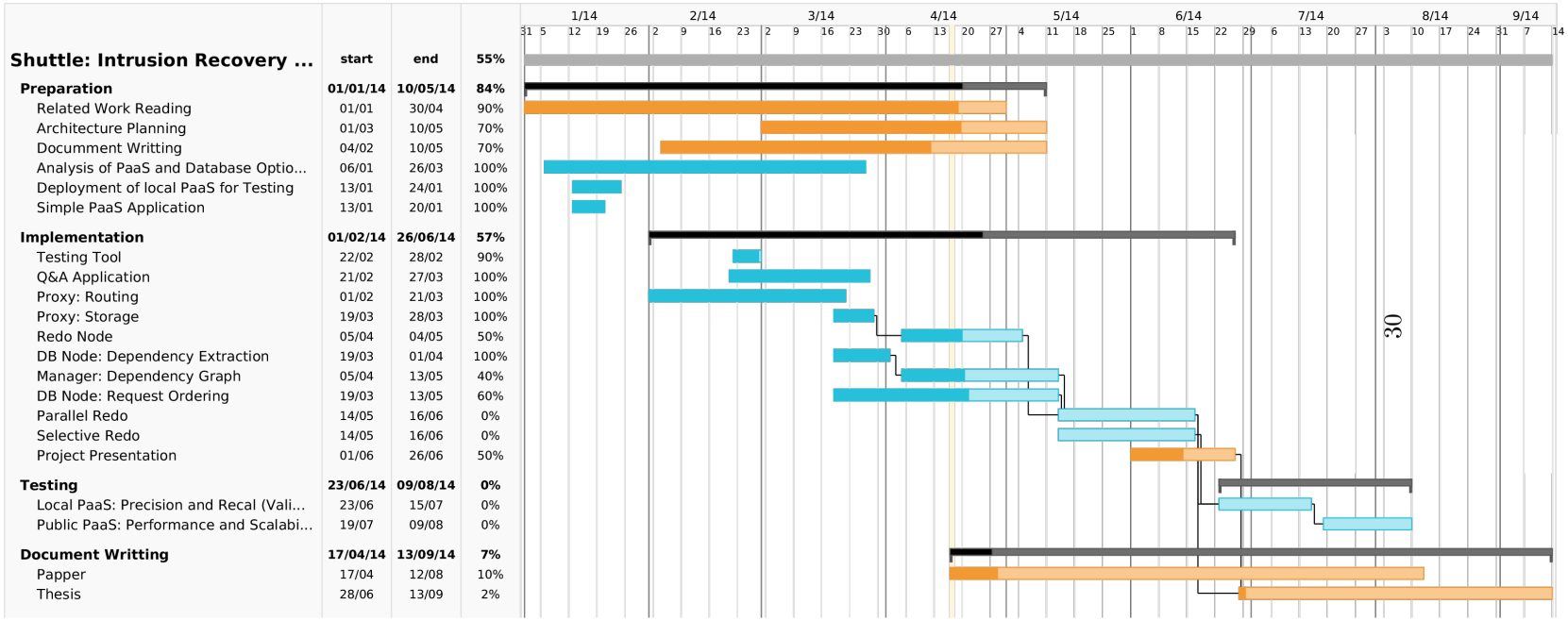
Several system were presented to illustrate the current state of the art of intrusion recovery for operating systems, databases and multi-tier applications. Conclusoes disso Finally, we survey the Platform as a Service (PaaS) architecture.

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 for 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).

A Detailed Schedule



Shuttle: Intrusion Recovery ...	start	end	55%
Preparation	**01/01/14**	**10/05/14**	**84%**
Related Work Reading	01/01	30/04	90%
Architecture Planning	01/03	10/05	70%
Documment Writting	04/02	10/05	70%
Analysis of PaaS and Database Optio...	06/01	26/03	100%
Deployment of local PaaS for Testing	13/01	24/01	100%
Simple PaaS Application	13/01	20/01	100%
Implementation	**01/02/14**	**26/06/14**	**57%**
Testing Tool	22/02	28/02	90%
Q&A Application	21/02	27/03	100%
Proxy: Routing	01/02	21/03	100%
Proxy: Storage	19/03	28/03	100%
Redo Node	05/04	04/05	50%
DB Node: Dependency Extraction	19/03	01/04	100%
Manager: Dependency Graph	05/04	13/05	40%
DB Node: Request Ordering	19/03	13/05	60%
Parallel Redo	14/05	16/06	0%
Selective Redo	14/05	16/06	0%
Project Presentation	01/06	26/06	50%
Testing	**23/06/14**	**09/08/14**	**0%**
Local PaaS: Precision and Recal (Vali...	23/06	15/07	0%
Public PaaS: Performance and Scalabi...	19/07	09/08	0%
Document Writting	**17/04/14**	**13/09/14**	**7%**
Papper	17/04	12/08	10%
Thesis	28/06	13/09	2%

Fig. 5: 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. 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
5. Amazon elastic beanstalk. [Online]. Available: http://aws.amazon.com/pt/elasticbeanstalk/
6. Google app engine. [Online]. Available: https://developers.google.com/appengine/
7. [Online]. Available: https://www.heroku.com/
8. Openshift. [Online]. Available: https://www.openshift.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
10. [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/
13. 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
14. 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
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. Avizienis, J. Laprie, and B. Randell, *Fundamental concepts of dependability*, 2001. [Online]. Available: http://www.malekinezhad.com/FOCD.pdf
18. 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
19. 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

20. 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
21. V. Stavridou, "Intrusion tolerant software architectures," *... & Exposition II, 2001 ...*, 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932175
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. L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks (1976)*, vol. 2, no. 2, pp. 95–114, May 1978. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/0376507578900454
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
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
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. 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
28. 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.
29. 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
30. 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
31. 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
32. 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
33. 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
34. 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

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

49. 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
50. 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
51. 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
52. 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
53. 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
54. 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
55. W. Vogels, "Eventually consistent," *Communications of the ACM*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1435432
56. S. Chacon, *Pro Git*. Apress, 2009.
57. 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
58. "John Knight Willow."
59. [Online]. Available: Xen: http://www.xenserver.org/
60. [Online]. Available: KVM: http://www.linux-kvm.org/
61. [Online]. Available: openstack: https://www.openstack.org/
62. [Online]. Available: AWS EC2: https://aws.amazon.com/pt/ec2/
63. [Online]. Available: Eucalyptus: https://www.eucalyptus.com/
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
65. [Online]. Available: http://stackoverflow.com/
66. [Online]. Available: https://answers.yahoo.com/