



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 their applications. The motivation for this work is the increasing number of intrusions and critical applications in Cloud, particularly on the emergence of the Platform as a Service (PaaS) model ~~and the perspective of recovery from security intrusions~~. While security research addresses prevention and distributed systems research 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 is being much used for obtaining computation resources and services on demand [1,2], PaaS intends to reduce the cost of scale software deployment and maintenance. This model defines an environment for execution and deployment of applications in *containers*. Containers provide software stacks, e.g., Python+MySQL, where developers deploy their applications ~~for users~~. Containers can be bare-metal machines, virtual machines or process isolation mechanisms such as Linux cgroups [3]. A PaaS provides a set of programming and middleware services to support application design, implementation and maintenance. Examples of this services are load-balancing, automatic server configuration and storage APIs. This usually ensures that the developer has 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]. The success of PaaS systems will ~~also~~ be established by the qualities of their services: correctness, security, consistency and recovery.



The number of critical and complex applications in 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. Therefore, the risk of intrusion is higher because exploitation of vulnerabilities is more attractive and profitable. When an attacker exploits an vulnerability successfully, there is an intrusion, which can be considered to be a fault. Faults may cause system failure and, consequently, downtime that has significant business losses [12]. Recovery solutions are needed to remove faults and restore the proper behavior.

Prevention and detection of malicious activity are the priorities of most security processes. However, preventing vulnerabilities by design is not enough because software tends to have flaws due to complexity and budget/time constraints [13]. More, attackers can spend years developing new ingenious and unanticipated attack methods having access to what protects the application while the guardian has to predict new methods and solve attacks in few minutes to prevent intrusions and mitigate vulnerabilities. On the 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.

The approach followed in this work consists in recovering the state of applications when intrusions happen, instead of trying to prevent them from happening. Intrusion recovery does not aim to substitute prevention, but to be an additional security mechanism. Similarly to fault tolerance, intrusion recovery accepts that faults occur and have to be processed. Data backup solutions roll back intrusion effects but require extensive administrator effort to re-execute legitimate actions. Our goal is to design, implement and evaluate *Shuttle*¹, a intrusion recovery service for PaaS. Shuttle prevents information losses **penso que previna indirectamente pois guardo os pedidos, se a informacao da base de dados for perdida, pode ser recuperada refazendo os pedidos mas nao falei muito disto** 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 compromising their availability. Shuttle cannot avoid information leaks, so confidentiality is out of the scope of this work. Intrusions recovery normally requires extensive human intervention to remove the intrusion and restore the application state. We believe that a service that removes the intrusion effects and restores



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

the applications integrity, without exposing a downtime during the process, is a significant asset to system administrators of PaaS applications.

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, which are mechanisms we will use 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 [14]. Shuttle leverage the resources scalability in PaaS environments to record and re-execute user requests. Shuttle uses clean PaaS container images to renew the application containers and re-executes user requests in parallel to recover the application persistent state integrity 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 the persistent state integrity, i.e., the persistent state allows the applications to behave according to their specification. substitui consistent por integridade, a explicacao fica boa assim ou demasiado suscinta? 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 explains the goals and expected results of our work; Section 3 presents the fundamental concepts and previous solutions; Section 4 describes briefly the architecture of PaaS frameworks and the proposed architecture for intrusion recovery system; Section 5 defines the 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 have been compromised due to software vulnerabilities.
- *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 *supports software updates* to prevent future intrusions and allows operators to try new configurations or software versions without effects in the application behavior perceived by users.

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

- *Remove intrusion effects:* Remove corrupted data in operating system, database and application level instances and update affected legitimate actions.
- *Remove selected malicious requests:* Help administrator to track the intrusion producing the set of actions affected by an externally provided list of malicious actions.
- *Support software update:* The application persistent state, after the recover process using a new software version, must be consistent with software updates. Como posso explicar concisamente que se o administrador actualizar o software e executar o processo de recovery, o estado persistente vai ser recriado e por isso vai ser consistente com a nova aplicacao
- *Recovery without stopping the application:* Recover the application without exposing users to application downtime.
- *Determinism:* Despite parallel execution of requests, the results of re-execution are the same as the result of original execution if the application source code and requests remain equal. encaixei aqui no sentido de tal como o low runtime overhead e without stopping, sao requisitos para os main goals que estao no inicio, devo retirar?
- *Low runtime overhead:* The recording of operations or state for recovery purposes should have a negligible impact in the runtime performance.
- *NoSQL Database Snapshot:* Create database snapshots to reduce the number of actions to recover and the recovery time. como poderia explicar esta linha sem explicar que vou refazer pedidos? posso colocar so: Create database snapshots to reduce the recovery time?
- *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 [15,16]. 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 [16].

Intrusions are malicious faults originating 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 [17]. In order to develop dependable system, delivering a resilient service, we need a combination of intrusion forecast, prevention, detection, mitigation, tolerance, recovery, and service (Fig. 1).

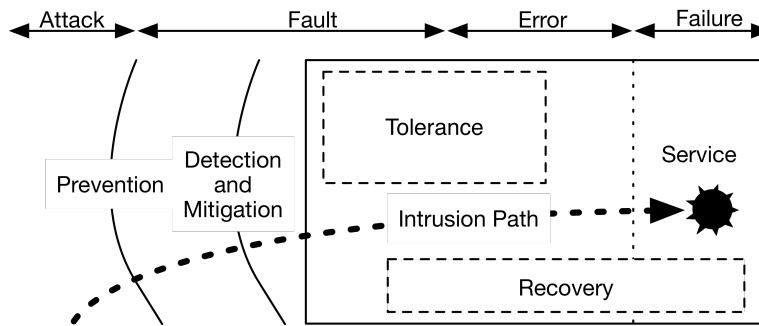


Fig. 1: Intrusion path across the system

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 [13,15]. Also system administrators, as humans, can make security configuration mistakes or users may grant access to attackers [18]. Moreover attackers can spend years developing new ingenious and unanticipated intrusions having access to what protects the system while the guardian has to predict new methods. Due to this asymmetry, it is arguably ~~impossible to protect all vulnerabilities by design. Therefore the vulnerabilities of prevention mechanisms can be exploited successfully leading to an intrusion.~~

Intrusion detection and mitigation monitors the system ~~events~~ to detect suspicious actions that may be connected with an intrusion. Intrusion detection systems (IDS) [19] 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 suspicious TCP 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 [20]. Instead of trying to prevent every single intrusion, these are allowed, but tolerated because the system has mechanisms that prevent the intrusion from generate a system failure [21]. Intrusion tolerance 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 the 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 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 are the sum of integrity and availability of each data entry and system component weighted by their importance [34]. Different systems 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 restore 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 begin. Notice that $A_{after} = A_{intrusion} \cup A_{legal}$.

A simple recover system, like backup, aims to obtain the set of objects $D_{recovered}$ that is associated with the set legitimate actions of $A_{recovered}$, where $A_{recovered} = H - A_{intrusion} : H \cap A_{after} = \emptyset$, i.e., get a set of objects without intrusive nether any action that occur after the intrusion. More advanced systems can perform $D_{recovered} = D - D_{intrusion} : D_{after} \subseteq D$, i.e., remove the values of objects written by intrusive actions and keep the values of 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. **tainted** objects are written by intrusion actions or tainted actions. Therefore $D_{legal} \neq D - D_{intrusion}$ and $D_{legal} \neq D - (D_{intrusion} \cup D_{tainted})$, 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 without the subset $A_{intrusion}$. The set of objects D_{legal} is obtained from the set of actions $A_{recovered} = H - A_{intrusion} : A_{legal} \subseteq H$. Since $A_{recovered} \cap A_{intrusion} = \emptyset \implies D_{tainted} = \emptyset \wedge D_{intrusion} = \emptyset$, $A_{tainted}$ would have a different execution. Therefore, we prove that intrusion recovery solutions require a process, named **redo**, where actions from set $A_{recovered}$ are re-executed to obtain a legitimate object

set that excludes the effects of intrusion.

The set of actions prior to intrusion, $A_{before} = H - A_{after}$, can include an extensive number of actions. Each action takes a variable but not null time to perform. Therefore, redo A_{before} may take 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 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.

Rewind [35] designates the process redo every action in $H - A_{checkpoint} - A_{intrusion}$. However, it may still take a long time. **Selective redo** [36], which is an alternative approach, redo only $A_{tainted}$ to update the objects in D .

Recovery solutions have two distinct phases: **record phase** and **recovery phase**. The record phase is the system usual state where the system is running and the solution records the system actions. 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 actions input and the value of every non-deterministic behavior to turn their re-execution into a deterministic process. The **recovery phase** can have three phases: determine the affected actions and/or objects, remove these effects and redo the necessary actions to recover a consistent state. Recovery solutions support **runtime recovery** when record and recovery phases occur simultaneously. They do not require system downtime to recovery.

Since objects are shared between actions, we can establish dependencies between actions. Dependencies can be visualized as a graph. The nodes of an **action dependency graph** represent actions and the edges indicate dependencies through shared objects. The **object dependency graph** establish the dependency between objects through actions. Dependency graphs are used to order actions re-execution[37], get the set of actions affected by a object value change [38], get the set of actions tainted by intrusion attacking actions [39] or resolve the set of objects and actions that caused the intrusion using a set of known tainted objects [40]. **Taint algorithm** aims to resolve the tainted objects $D_{tainted}$ from a source set of intrusive actions $A_{intrusion}$ or objects $D_{intrusion}$ using the dependency graph. **Taint propagation via redo** [41] algorithm, which extends the previous algorithm, finds out $D_{tainted}$ replaying the

actions that have input changes during the redo phase. Taint propagation via redo algorithm restores the values of $D_{intrusion}$ redoing only the legal actions that output $D_{intrusion}$; then it redo the actions dependent from $D_{intrusion}$, updating their output objects. While the forward actions have different input, they are also redo and their output is updated. fica mal aqui? e uma tecnica especifica e dificil de explicar suscita...

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

3.4 Recovery at Operating System Level

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. First, we introduce the concept of dependency at Operating System level. Then we present two intrusion recovery systems that use dependency rules and tainting propagation via replay. Finally, we present proposals for recovery in computing clusters, virtual machines and network file systems.

BackTracker [40]: 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. It 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 [42] 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 tainting algorithm to track the intrusion effects of fine, 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. The result is a dependency graph where nodes are objects and processes that were involved to write the compromised objects.

Objects shared between many processes, e.g., `/tmp/` or `/var/run/utmp` which tracks login/logout of users, or objects, which are not used by processes to generate the output, are likely to produce false dependencies leading to false positives. False positives obfuscate the actions of attackers or lead to legitimate data loss. Therefore, Backtracker proposes a *white-list* filter that ignores common shared files. However, this technique relies in the administrator knowledge and introduces false negatives if attackers perform on *white-listed* objects. Backtracker do not perform any proactive task to remove or recover from intrusions but provides the common rules for intrusion recovery in Operating Systems.

Taser [43]: Taser removes the intrusion effects from current state loading *selectively* a legitimate version of tainted files from a file system snapshot. It performs only the legitimate actions that modified the tainted files after the snapshot. Taser relies on Forensix [36] 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 [36] to audit the system actions during the record phase and determine the tainted objects at recovery phase. Forensix logs 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 *object 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 *tainted*, 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” [43]. However, attackers can leverage these optimistic policies to penetrate the Operating System.

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

objects of the attack, $D_{intrusion}$, and transverses the graph adding all tainted objects to the set $D_{tainted}$.

Taser removes the effects of tainted loading a previous version only of tainted objects from a file-system snapshot. To recover a coherent state, Taser performs *selective redo* replaying sequentially the legitimate modification operations of tainted objects since the snapshot. Non-tainted files remain unchanged.

Taser do not update the objects dependent from tainted objects. The redo process only recovers a consistent state for the originally tainted objects. Therefore, Taser ignores a set of actions that would read the modified version and have a different execution and output. The problem is concerned in [44]. More, 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.

Retro [41]: 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 snapshot and performs selective redo using *taint propagation via redo*.

During record phase, the kernel module of Retro creates periodic snapshots of the system and records system-wide interactions between objects and processes on a dependency graph. 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 is defined by arguments and output objects of system calls and associated to the process execution (Fig. 2). This graph is finer-grained than Backtracker graph since dependencies are established per system call instead of per process and it contains more information than the homologous in Taser.

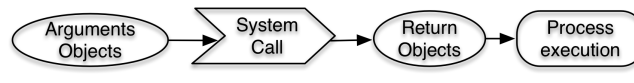


Fig. 2: Dependency graph due to system call in Retro

During the recovery phase, Retro requires the administrator to identify $A_{intrusion}$, processes, system calls or $D_{intrusion}$ objects which caused the intrusion. First, it changes the graph replacing the intrusive system calls with null actions. Then it performs *taint propagation via redo*. It rollback the objects in $D_{intrusion}$, which have been modified by $A_{intrusion}$, to a previous version from a snapshot. Then, the system calls, which are dependent from the restored objects, are redo and their output objects are updated. The forward system calls are also redo while their inputs are different from the original execution. The propagation is done

thought the output of system calls with different execution and the recovery process terminates when propagation stops.

Since Retro redo processes with system call granularity instead of process granularity, the redo process may stop earlier. However, since Retro do not checkpoint the state of processes, the system must be restart to remove the current non-persistent states and processes must be re-executed from beginning to load their non-persistent state and perform their system calls with the correct state. 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* [45], extends Retro to recover from intrusions in distributed systems. It adds the dependencies through sockets. The machines involved in a network session add socket objects to the dependency graph. Network protocol, source and destination IP and ports and one ID, which is exchanged in every package during connection, globally identify each socket object. 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 significant 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 amount of log entries 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, as Retro, the affected machines must be off-line because its propagation algorithm shutdown the service during the repair phase.

Bezoar [42]: 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 encompasses the entire system: processes and kernel spaces, resources, file system, virtual memory, CPU registers, virtual hard disk and memory of all virtual external devices. Bezoar tracks how the data from network connections propagates in memory. Bezoar does 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 does not provide any external consistency warranties.

Repairable File System (RFS) [46]: 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 includes a network file system (NFS) client module and a server NFS module.

The client module tracks system calls using ExecRecorder [47] and establishes the dependence between processes and NFS requests. Requests 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 it matches 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 remain legitimate if they read a correct block from a contaminated file. RFS uses the version server to rollback only the affected files.

RFS is resilient to log attacks but legitimate clients need to store their log. More, RFS, as Taser, does not update the files dependent from tainted files and can remove legitimate files due to false positive dependencies.

Summary: Dependencies 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 propagates the effects of read dependencies. However, Retro propagates the effects based on input changes recorded to store the input of every action and re-execute the processes since log.

Operating system layer solutions 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 recover. However, due to low abstraction level, the dependency graph and log are too big and complex to remove false dependencies promptly and provide an intrusion tolerant system. a palavra sumario nao fica bem, e mais uma conclusao de subcapitulo mas nao sei se devo usar conclusion, comments, analyze...

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 established between transactions.

Compromised transactions are determined from an 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 " [48] 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" [48]. 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. 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.* [49] 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 these sets and restore the tainted entries.

ITDB [48][50][34]: Intrusion Tolerant Database (ITDB) performs intrusion recovery in databases using compensation and supports *runtime recovery*, i.e., the database service remains available during the recovery process. ITDB uses the set of dependency rules described above and extracts the read set parsing the SQL statements.

During the record phase, ITDB audits the read and write sets of each transaction. Since most of SQL DBMS only keep write logs, Liu *et al.* proposes 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.

At recovery ~~phase~~, ITDB reads the log of read/write sets from the intrusion moment to the present and generates ~~the~~ $D_{tainted}$ adding the elements from the write sets of tainted transactions to $D_{tainted}$. For each good transactions, ITDB keeps the write set until transaction commits or aborts; if the transaction commits after some entry that belongs to $D_{tainted}$, then every entry in the write set of transaction is also considered in the set dirty and the transaction must be compensated. The compensation of a transaction is performed using the inverse modification of the original transaction to restore the previous value. Repaired entries in $D_{tainted}$ are tracked to prevent compensation of later transactions from restore a repaired data entry to its version after the attack. After this process, the latest legitimate value of $D_{tainted}$ entries is recovered.

If the intrusion propagation is faster than the recovery process then the recovery phase is endless because the damage will spread through new transactions. To prevent damage spreading, ITDB blocks the read 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 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. The solution supports concurrent recovery but aborts new transactions that read compromised data items thus incur on degraded performance.

ITDB architecture includes an intrusion detection system. The IDS is application aware and acts at 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.

ITDB recovery phase does not contain a redo phase where it re-executes read dependent transactions. Therefore, it points out the fact that legitimate executions can be influenced by the removed values [49]. Liu *et al.* propose a theoretical model based on possible workflows and versioning. However, predict every possible workflow requires extensive computational and storage resources.

Phoenix [51] : Phoenix removes the intrusion effects using a versioned database. While Liu *et al.* [48] 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 transactions 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 modifies the DBMS source code to obtain read set and generate a dependency graph. On the other hand, ITDB extracts the read sets parsing the SQL statements and uses the log and compensation transactions to remove the intrusion effects. Phoenix is query independent but keeps every version in memory. ITDB is application dependent because it defines 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 removed.

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 [39] : Goel *et al.* propose a recovery system that selectively removes intrusion effects on web applications that store their persistent data in a SQL database. Since each user request may involve multiple transactions, it captures dependencies at user, 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, a monitor logs each user request and response, every read/write set of associated database transactions and the modified database tables and rows. Each transaction is ordered, since the database uses serializable isolation, and stamped with an id that establish the redo order.

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 dependency graph between transactions is determined using database logs [48]. Taser also proposes a white list and optimistic dependency policies to prevent false positives. 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. Goel *et al.* use *taint propagation via redo*: the requests dependent from data written by intrusive actions are redo and the outgoing edges in the dependency graph are also redo. The process determines the set of tainted database rows in an isolated database. At undo phase, the system applies a compensation transaction that copies the modified entries from the sandbox to the current database.

Poirot [52]: 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 software is updated to fix its flaws. 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 updated e non-updated version [53]. 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 [38]: Warp is a patch based intrusion recovery solution for single server web applications backed by SQL databases. Unlike previous approaches, Warp allows administrators to retroactively apply security updates without track down the source of intrusion and supports attacks on user browser level. Warp is based on Retro [41] taint propagation via redo approach and removes the intrusion effects using a versioned database. The Warp prototype uses 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 appli-

cation. During request dispatch, Warp records non-deterministic functions and a list of invoked PHP 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.

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. These allows selective rollback and read previous rows versions. Warp support concurrent repair using two integer columns to define the *repair generation*. At repair phase, the current repair generation ID is incremented to fork the database. User requests are performed in the current generation while recovery requests are perform in the next generation. After redo the requests retrieved until the begin of the repairing process, the server stops and applies the remain requests.

During repair phase, the administrator updates the application software to fix its flaws. Warp computes the set of requests that executed modified files [52] [53]. These requests are the root cause of changes during re-execution. To update the database and reflect the patch, each user request is redo using a server-side browser. The modified PHP interpreter intercepts non-deterministic function calls during replay and returns the original logged value. Also, database read queries are replayed only if the set of *affect rows* is different or its content was modified. Write queries are replayed loading a previous version of the rows and replaying the query. Each row, which has a different result after re-execution, is now tainted and all requests that read the row are also redo at browser level. Finally, the HTTP response is compared with original and any difference is logged. If responses are different, the following dependent user interactions on browser are replayed.

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

Warp modifies the query to write the required metadata. However, storing 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 application deployed in multiple servers nether distributed databases since versions are timestamp based.

Aire [54] : Aire is an intrusion recovery system for loosely coupled web services. It extends the concept of local recovery by Warp [38] tracking attacks across services. While Dare [45] aims to recover a server cluster synchronously using Retro [41] 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 [55] [56]. The model allows clients to observe the result of update operations in different orders or

delayed during a period called *inconsistency window* until the remote server recovers. Aire considers the repair process as a concurrent client. To repair key-value database entries, Aire creates a new branch [57] and re-applies legitimate changes. At end of local repair, Aire moves the current branch pointer to the repaired branch.

Like Warp [38], 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 at end.

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 [45], requires system stop during recovery process which is equivalent to service failure.

Undo for Operators [7]: 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” [35] 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 support MAP e SMTP. Undo Operators define the concept of *verb*. Each protocol operation has its own *verb* class. A verb is object to encapsulate the user interactions and exposes an interface to 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 establishes 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 fixes 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 defined per verb 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 repair during runtime 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 establishes 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 consistency, Warp detects inconsistencies on responses and replays the user interaction on browser and Brown *et al.* all use compensation actions based on protocol-specific knowledge.

Finally, Warp do not account operating system attacks, e.g., a shell access through a web site form vulnerability.

Goel *et al.* The usage of compensation transactions avoids the overhead of keeping versions and checkpoints but implies longer recovery phase and the knowledge of inverse transaction.

3.7 Evaluation of Technologies

The previous subsections describe various systems that use different approaches to recover from intrusions. The level of abstraction outlines the recorded elements. Operating System solutions record system calls and file system, Database solutions track inter-transaction dependencies and Application level solution track transactions, requests and execution code. The log at operating system level is more detailed but may obfuscate the attack in false positive prevention



techniques. At database and application level, the log is semantically rich but do not track low level intrusion.

While most of solutions require administrator to pinpoint the initial corrupted set of actions using an external mean, e.g., IDS, Warp track the requests which invoked modified code files. Pinpoint actions incurs on false positives and negatives due to administrator mistakes. Tracking the invoked code files requires changes to the interpreter but avoids false positives and negatives.

The tainting algorithm, which determines $D_{intrusion}$ and $D_{tainted}$, is performed statically using the original execution dependencies recorded in a dependency graph or dynamically replaying the legitimate actions which have a different input in replay phase than in original execution. The reflects the changes during the redo phase, therefore clearly better but requires to store the input of every action. An alternative proposed by Brown *et al.* sorts the requests using application dependent and the redo phase will reflect the changes. This approach is possible because every request is redo with a known order. The tradeoff between perform taint propagation via redo or redo every request is equivalent to a tradeoff between storage and computation. In the worst case, taint propagation via redo will redo the same number of requests than Brown *et al.* approach.

Warp supports application source code changes tracking the original code invocations and comparing output when the application functions are re-invoked. Brown *et al.* support application changes using application dependent compensation actions to resolve conflicts.

At recovery phase, the solutions remove the intrusion effects and recover a consistent state. Three of the possible options to remove the intrusion are: snapshot, compensation and versioning. Versioning, which is a per-entry and per-write snapshot, is finer-grained and allows the reading previous versions without redo the actions after the snapshot. However, the storage requirements to keep the versions of every entry in a large application can be an economic barrier. In [39], the snapshot is built reverting transactions modification in the current values. This approach does not require storing the copy of values but requires more computational resources to create a snapshot.

To recover a consistent state, the intrusion recovery solutions should redo the legitimate actions dependent from intrusion actions (Section 3.3). While Brown *et al* propose to redo all legitimate user requests sorted by an application-dependent algorithm, the remain solutions use tainting via redo to selectively redo only the dependent actions. The late approach may hide indirect dependencies [49] but recovers faster. More, in Brown *et al*, the replay of every request generates conflicts that must be solved in order to archive a consistent state. Warp queues the conflicts for later solving by users. The proposals [50, 38, 54] support online recovery where recovery phase do not require system downtime.

System	Stores	Intrusion Pinpointing	Tainting Algorithm	Supports Code Changes
[43] Taser	System Input	Tainted objects	Graph	-
[41][45] Retro/Dare	System Call Input and Output	Intrusion actions	Taint via redo	-
[50] ITDB	Read/Write set using SQL parsing	Tainted Objects	Set expansion (graph)	-
[51] Phoenix	Read/Write set using DBMS modification	Tainted Objects	Graph	-
[39] Goel <i>et al.</i>	User, session, request, code execution, database row	Intrusive requests	Graph and Taint via redo	✗
[38][54] Warp/Aire	Client side browser interactions, requests, user sessions, invoked PHP files	Source code patch	Taint via redo	✓
[37] Brown <i>et al.</i>	Requests	Request modification	Application dependent dependencies	✓
Shuttle	User requests, session and read/write set using DBMS changes	Source code patch or Requests modification	Graph and Taint via redo	✓

Table 1: Summary of storing and intrusion tracking options

This characteristic is required to archive intrusion tolerance however it allows intrusion to spread during the recovery period. In [50], the access to tainted data is denied. However, it compromises the availability during recovery period.

We conclude that a solution that combines Brown *et al* and Warp can recover from intrusions in PaaS applications. **devo justicar aqui ou deixar para a architecture?**

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

System	Previous state recovery	Effect removal	Redo phase	Runtime recovery	Externally consistent
[43]Taser	Snapshot	Remove intrusion requests; load legitimate entries value from a snapshot	No		
[41][45] Retro/Dare	Snapshot	Remove intrusion requests; load legitimate entries value from a snapshot	Tainting via redo		
[50] ITDB	Transaction Compensation	Compensate tainted transactions	No	✓	
[51] Phoenix	Row versioning	Abort tainted transactions	No		
[39] Goel <i>et al.</i>	Transaction Compensation	Snapshot using transaction compensation	Tainting via redo		
[38][54] Warp/Aire	Row versioning	Load previous entry version	Tainting via redo	✓	✓
[37] Brown <i>et al.</i>	Snapshot	Load Snapshot	Redo all requests sorted by application semantics		✓
Shuttle	Snapshot	Load Snapshot	Redo tainted requests sorting by original execution read/write sets	✓	✓

Table 2: Summary of ~~consistent~~ state recovery options

access to data storage, middleware solutions and other services hiding lower level details and providing a pay-per-usage ~~extremely~~ scalable software infrastructure. IaaS 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[58], KVM[59], 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 [60], AWS EC2 [61], Eucalyptus [62]). The tenant applications deployed in PaaS are designed to scale horizontally. Each container is isolated and distributed states are avoided. Database or session cookies maintain the application state.

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.

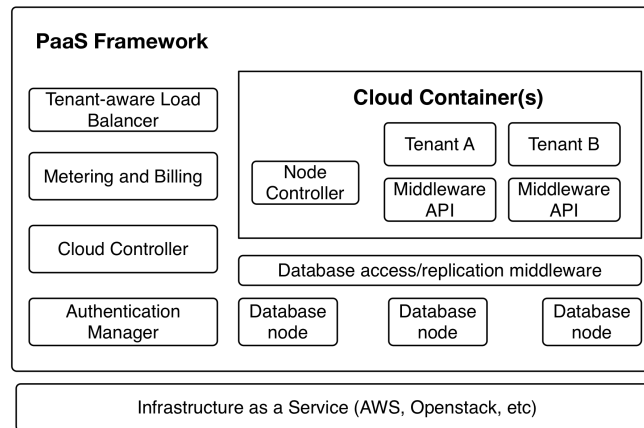


Fig. 3: Generic Architecture of PaaS

4.2 Shuttle

While previous work aims to perform intrusion 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 a scalable intrusion recovery service that is transparent for application developers. Shuttle logs and re-executes user HTTP requests to recover from intrusions.

Our solution is designed using a proxy architecture that logs the user HTTP requests without modification of PaaS components. Shuttle is composed of following modules (Figure 4):

- **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 interceptor**.
- **Database Servers:** The database servers runs a NoSQL key-value storage software that stores the application persistent state. Each server contains a **database server auditor** that logs the requests dependency at database level.

- **Request Storage:** 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 server.
- **Manager:** The manager is the coordinator of Shuttle.

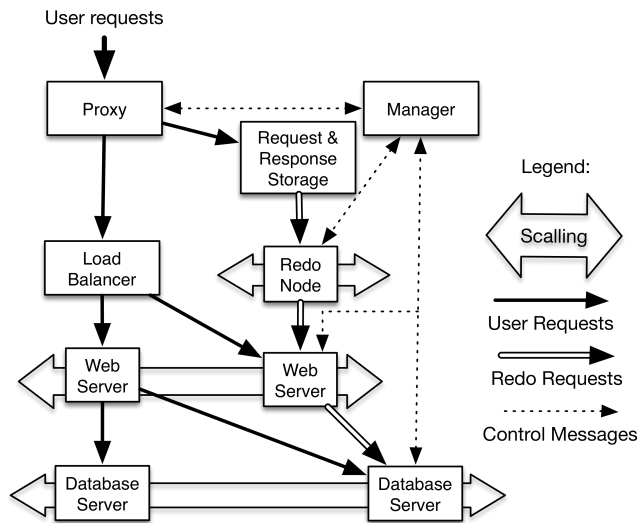


Fig. 4: Overview of the proposed system

The PaaS architecture fits the transparency and scalability requirements. It supports to scale database, computation and redo nodes, includes a load balancer. More, nearly all deployed application use HTTP protocol and most of developers use a provided storage API, which can hide the Shuttle service.

Shuttle targets integrity intrusions that are created through malicious or accidental HTTP request mistakes or exploitation of software flaws in applications or their containers. Therefore, Shuttle aims to support intrusion pinpointing through source code patching or modification of user request sequence or content.

During the record phase, the *proxy* logs the user HTTP requests, add a timestamp to its header as request id (RID) and forward to the load balancer. These requests are stored in an external distributed storage system asynchronously. The access to database servers is intercepted by a *proxy* that serializes the access for each key allowing multiple-reads or a single write. The access order in each database server is recorded including the RID and sent to the *manager*. The response is also stored in the external storage.

The redo phase is initialized when intrusions, suspicious or mistakenly behaviors are detected or the application software requires an update. The set of malicious requests, if any, is identified and removed or modified from the original request sequence by the administrator. Then the *manager* loads the snapshot on each database node and new PaaS instances, which may include code fixes, are initiated. The new system is intrusion-less because the snapshot and instances are clean and the code is fixed. The requests are replays in parallel using a set of HTTP clients named *redo nodes*. These requests are redo in a different database version. Since users still use the corrupted computation instances and database, the application remains online, perhaps with a degraded behavior, without exposing downtime to users. After recovery, the users requests are switched to the system with fixed state.

The challenge of reduce the amount of requests during the redo phase will be analyzed during the development phase. como posso dizer que vou pensar no caso mais tarde, nao e prioridade

One of the main challenge to reduce the recovery time to acceptable values. We aim to use two techniques: snapshot and parallel redo.

Perform a snapshot in a distributed NoSQL Key-Value storage is not trivial since it must be consistent with the computation tier user requests. User requests may include multiple database accesses. The snapshot must include only completed user requests, otherwise the redo phase will be inconsistent because Shuttle would redo a request in a snapshot that contains part of its old persistent state. Therefore, the *manager* leverages the request timestamping to schedule a snapshot. The administrator defines a future instant t where the snapshot will occur and the manager forwards this information to every database node. The requests, which have a timestamp bigger than t , will write in a new version and read the latest. The remain requests will write in the previous version and read only versions written by requests with timestamp lower than t . This mechanism is based on copy-on-write algorithm.

Previous solutions leverage the request serialization provided by snapshot isolation in SQL DBMS to sort the redo operations. However, PaaS perform multiple *transactions* accessing independent keys in NoSQL database nodes. Therefore, we need a new approach to serialize the requests and key access during redo. More, the keys accessed during redo may change due to code updates or request modification. While *in* [37] use the application protocol knowledge to establish the dependency between requests, our work aims to support any application deployed on PaaS. Therefore, we use the captured database key access list to generate a dependency graph. The graph reflects the dependencies between original execution. False dependencies in this graph do not cause data lost or inconsistent state since they are used only to order the requests redo as *in* [48].

Shuttle can not execute requests in serial order. It not only would have performance degradation but also lock the replay phase because requests are originally executed in parallel and may depend from each other. On the other hand, PaaS applications have loosely couple requests that share data only by database re-

quests. Therefore, Shuttle will perform as many requests in parallel as possible to reduce the duration of the redo phase. Yet, the processing order in the computation tier neither the time between accesses to the database is deterministic. So Shuttle uses the original key access list to order the access to each key. This method is dead-lock free because if the code and requests remain equal. Otherwise, some request may not access the same keys or write/read the same content during replay creating request starvation and locking the redo process. Therefore, the database client interceptor will fetch the list of keys accessed by the original request execution and simulate the access to the remaining keys to unlock the requests, which are waiting in the key.

The elected database to backend the prototype is the key-value storage VolDEMORT [?]. VolDEMORT is an open-source implementation of Amazon Dynamo DB [55] in use by LinkedIn [?]. Key-value storage databases are common in PaaS environments because they scale properly and provide a simple put,get,delete API. We leverage this simple API to avoid the delay and mistakes of parsing SQL queries.

The external users requests and responses will be stored in the scalable NoSQL database Cassandra [?]. The database can be replicated to a remote site to prevent catastrophic disasters. Shuttle uses the externally stored user requests to recover from any kind of intrusion in the computation and database nodes. Since PaaS defines containers to run each module, Shuttle will collaborate with the coordinator of the PaaS framework to launch new containers using intrusion-free container images, which may include software updates. The old containers, which may contain unknown intrusions, are shutdown. The software updates may aim to prevent future intrusions or fix a discovered flaw [52,38]. Shuttle updates the application persistent state to reflect the updates removing the intrusion.




By redo the user requests in parallel using intrusion-free containers and a clean database snapshot, we have the best hope to recover from security intrusions in PaaS applications.


5 Evaluation

The evaluation of the proposed architecture will be done 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 [63]. 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  *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 [64] and Yahoo! Answers [65]. 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 [60] 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: Finish 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

~~Security~~ intrusion recovery systems recover the application state when intrusions happen, instead of trying to prevent them from happening. ~~They provide more reliable systems.~~ A number of research projects depicted their application in Operating Systems, databases and web applications. However,  of these projects provide scalable solutions for applications deployed in multiple servers and backed by a NoSQL database.

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

Shuttle will remove intrusion effects and recover a consistent persistent state of application that is deployed in PaaS. We propose, to the best of our knowledge, the first intrusion recovery for PaaS and 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

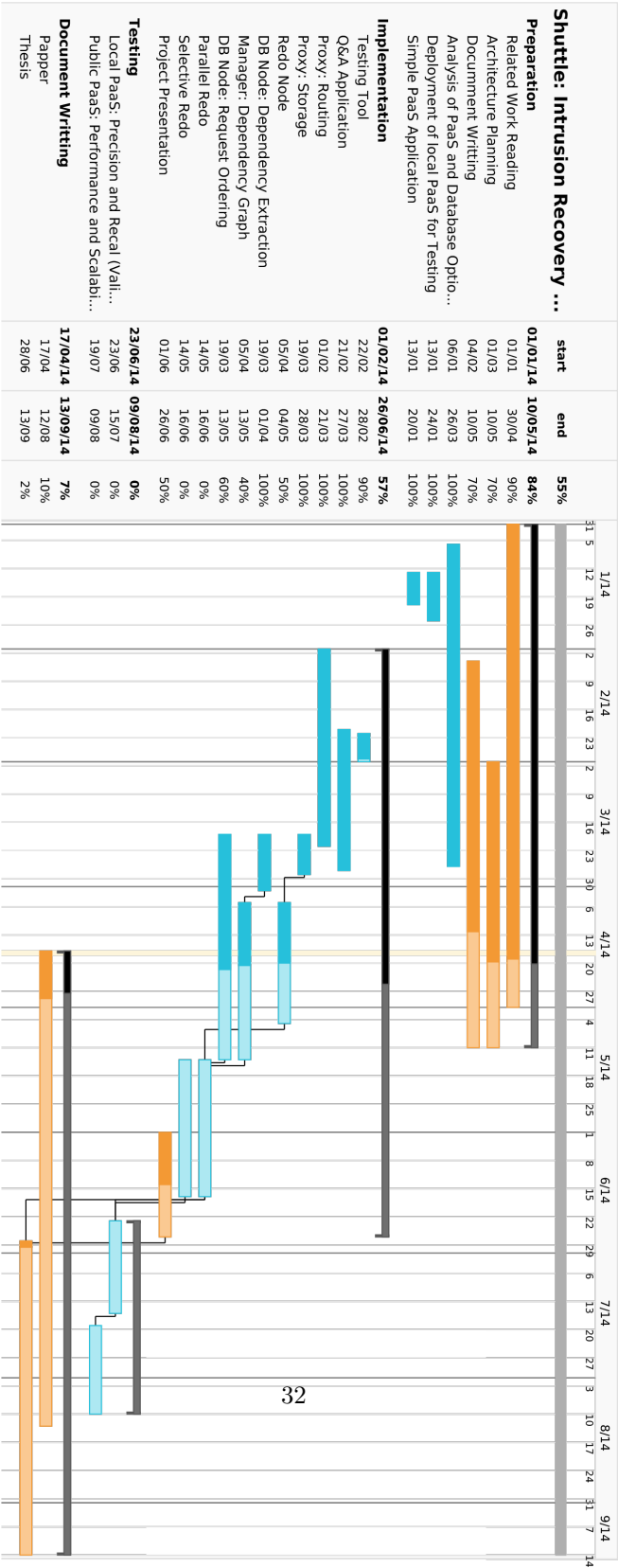


Fig. 5: Project Gantt Schedule



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. Roderio-Merino, and R. Buyya, “Dynamically scaling applications in the cloud,” *ACM SIGCOMM Computer ...*, vol. 41, no. 1, pp. 45–52, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1925869>
5. Amazon elastic beanstalk. [Online]. Available: <http://aws.amazon.com/pt/elasticbeanstalk/>
6. Google app engine. [Online]. Available: <https://developers.google.com/appengine/>
7. [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. R. Charette, “Why software fails,” *IEEE spectrum*, no. September 2005, pp. 42–49, 2005. [Online]. Available: <http://www.rose-hulman.edu/Users/faculty/young/OldFiles/CS-Classes/csse372/201310/Readings/WhySWFails-Charette.pdf>
14. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, “A view of Cloud Computing,” *Communications of the ...*, 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1721672>
15. C. 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>
16. A. Avizienis, J. Laprie, and B. Randell, *Fundamental concepts of dependability*, 2001. [Online]. Available: <http://www.malekinezhad.com/FOCD.pdf>
17. D. Powell, “Failure mode assumptions and assumption coverage,” *Fault-Tolerant Computing, 1992. FTCS-22. Digest of ...*, pp. 1–18, 1992. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=243562
18. A. Brown and D. Patterson, “To err is human,” *Proceedings of the First Workshop on ...*, 2001. [Online]. Available: <http://roc.cs.berkeley.edu/talks/pdf/easy01.pdf>
19. M. Roesch, “Snort – Lightweight Intrusion Detection for Networks,” *LISA*, 1999. [Online]. Available: <http://static.usenix.org/publications/library/proceedings/lisa99/full-papers/roesch/roesch.pdf>

20. V. Stavridou, "Intrusion tolerant software architectures," ... & Exposition II, 2001 ..., 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932175
21. P. Verissimo, N. Neves, and M. Correia, "Intrusion-tolerant architectures: Concepts and design," *Architecting Dependable Systems*, vol. 11583, 2003. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-45177-3_1
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. 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>
 36. 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>
 37. A. B. Brown and D. A. Patterson, "Undo for Operators: Building an Undoable E-mail Store," 2003.
 38. 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>
 39. 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>
 40. 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>
 41. T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion Recovery Using Selective Re-execution."
 42. 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>
 43. 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>
 44. 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/>
 45. 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>
 46. 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>
 47. 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>
 48. 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>

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