

# 1

## Context and Related Work

In this chapter, we give background information on relevant concepts and techniques that are touched in this thesis. We start ~~with a taxonomy~~ by introducing the main concepts of dependability in Section 1.1. This includes a discussion of intrusions and methods to avoid or tolerate them. Section 1.2 introduces the main intrusion recovery techniques. Each of the following sections describe a number of relevant proposals for recovery in the levels where **PaaS!** (**PaaS!**) applications are attacked: operating system, database and application. Finally, Section 1.3 discusses the contributions of these works in the context of this dissertation.

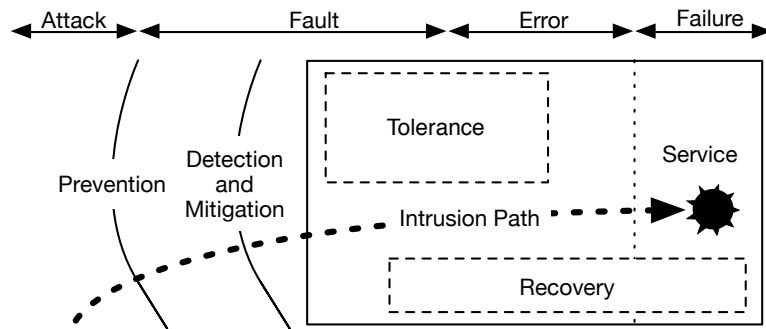
### 1.1 Dependability Concepts

The *dependability* of a computing system is the ability to deliver a trustworthy service [? ]. In particular, the concept of dependability encompasses the following attributes: *availability*

- Availability: service readiness for authorized users; *confidentiality*
- Confidentiality: absence of unauthorized disclosure of information; *safety*
- Safety: absence of catastrophic failures; *reliability*:
- Reliability: continuity of correct service; *integrity*
- Integrity: absence of improper system state.

Three core concepts in dependability are: fault, error and failure. **nada mudou para baixo**  
A *fault* is the cause of an error. The source of a fault belongs to the software or hardware domain and it can be introduced either *accidentally* or *maliciously* during the system development, production or operation phases [? ? ]. Faults can deviate the system from its specified behavior leading to errors (Fig-Figure 1.1). *Errors* are the part of the system state that may cause a subsequent failure. A system *failure* occurs when errors become observable at the system interface. In the context of this work, we consider faults which are generated by humans in an accidentally or deliberate, malicious or non-malicious way. In particular, we target software flaws faults and interaction faults from input mistakes, attacks and intrusions [? ].

An *intrusion* is a malicious fault resulting from an intentional vulnerability exploitation. As originally proposed generically for faults [? ? ], intrusions can be omissive, suspending a system component,



**Figure 1.1:** Intrusion path across the system

and/or assertive, changing a component to deliver a service with a not specified format or meaning. In order to develop a dependable system, delivering a resilient service, we can use a combination of intrusion forecast, prevention, detection, mitigation, tolerance and recovery (Fig-[Figure 1.1](#)).

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

*Intrusion detection and mitigation* mechanisms monitor the system to detect suspicious actions that may be connected with an intrusion. However, **IDS!** (**IDS!**) turn the system attack-aware but not attack-resilient, that is, they cannot maintain the integrity and availability of the system in face of attacks [?]. **IDS!** (**IDS!**) may not detect unknown vulnerabilities. Moreover, attackers may use encrypted actions or use legitimate requests.

*Intrusion tolerance* is the last line of defense against attacks before the system failure occurs ([Fig- Figure 1.1](#)). Intrusion tolerance is the ability of a system to keep providing a, possibly degraded but adequate, service during and after an intrusion [?]. Instead of trying to prevent every single intrusion, these are allowed, but tolerated. The system has mechanisms to prevent the intrusion from generating a system failure [?]. Intrusion tolerance mechanisms hide the errors effects using redundancy mechanisms or using intrusion recovery systems, which detect, process and recover from intrusions.

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

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

## 1.2 Intrusion Recovery

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

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

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

The third phase, and the one that this work is about, consists in removing the intrusion effects. Intrusions affect the application integrity, confidentiality and/or availability. To recover from availability or confidentiality violations is out of the scope of this document. However, we argue that the design of the applications should encompass cryptography techniques which may reduce data relevance and protect the data secrecy [? ].

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

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

### 1.2.1 Formalization of the Recovery Process

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

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

A recovery service aims to set the state of a service to  $O_{recovered}$  at the end of the recovery process. The set  $O_{recovered}$  shall be composed of objects as if their state was defined exclusively by a set of legitimate actions  $A_{recovered}$ . Objects of the subset  $O_{recovered}$  represent a new *intrusion-free* and



Figure 1.2: Set of actions of an application execution

*consistent state*. A state is consistent if it is valid according to the application specification. A state is intrusion-free if it is created only by legitimate actions. If the application respects the specification (correctness), then changing from  $O$  to  $O_{recovered}$  performs service restoration [? ], i.e., restores the application service to a correct behavior.

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

We define the set of *tainted* actions,  $A_{tainted}$ , and the set of tainted objects,  $O_{tainted}$ , at a certain instant in the following way: if an action belongs to  $A_{intrusion}$ , then it belongs to  $A_{tainted}$ ; if an object belongs to  $O_{intrusion}$  then it belongs to  $O_{tainted}$ ; if an action in  $A_{legal}$  reads an object in  $O_{tainted}$ , then that action belongs to  $A_{tainted}$ ; if an action in  $A_{tainted}$  writes an object value in  $O_{legal}$ , then that object belongs to  $O_{tainted}$  (Figure 1.2). Therefore,  $A_{tainted}$  includes  $A_{intrusion}$  but typically also actions from  $A_{legal}$  that were corrupted by corrupted state. Also,  $O_{tainted}$  includes  $O_{intrusion}$  but typically also objects from  $O_{legal}$  that were corrupted by corrupted state. Then, the set of object values written only by non-malicious actions is not the same as the set of objects obtained after removing the objects written by malicious or tainted actions, i.e.,  $O_{legal}$  written by  $A \cap A_{intrusion} = \emptyset$  is not equal to  $O_{legal}$  written by  $D - D_{intrusion}$  or  $O_{legal}$  written by  $D - D_{tainted}$ . In other words, to remove the objects written by intrusions and tainted actions is necessary but not enough to obtain the values of the set of objects that would be produced only by legitimate actions.

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

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

The sequence of actions,  $A_{before}$ , performed before the intrusion, i.e.,  $A_{before} = A - A_{after}$ , can be extensive. Each action takes a variable but not null time to perform. Therefore, to replay  $A_{recovered} : A_{before} \subseteq A_{recovered}$  may takes an excessive amount of time. We define the subsets  $O_{snapshot}(t)$  and  $A_{snapshot}(t) : A_{snapshot} \subseteq A$  as the subsets of object values and actions executed before the begin of a snapshot operation at instant  $t$ . The snapshot operation copies the value of the object immediately or on the next write operation. If the attack is subsequent to  $t$ , then  $A_{after} \cap A_{snapshot} = \emptyset \implies (A_{intrusion} \cup A_{tainted}) \cap A_{snapshot}(t) = \emptyset$ , i.e., the snapshot is not affected by intrusion. For that reason, the service can replay only  $A - A_{snapshot} - A_{intrusion}$  using the object set  $O_{snapshot}$  as base.

The most recent works, which are explained in the following sections, define two distinct approaches to update the set of object  $O$  to  $O_{recovered}$  because of changes in the execution of  $A_{tainted}$ : *rewind* and *selective replay*. The selective replay approach loads only the previous versions of the tainted objects,  $O_{tainted}$ , and replays only the legitimate operations, which were tainted,  $A_{tainted} \notin A_{intrusion}$ , to update the objects in  $O$ . The  $O_{legal} \notin D_{tainted}$  remain untouched. The alternative approach, *rewind* [? ], designates a process that loads a system wide snapshot previous to the intrusion

moment and replays every action in  $A - A_{snapshot} - A_{intrusion}$ . However, this process can take a long time.

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

Recovery services have two distinct phases: *record phase* and *recovery phase*. The record phase is the service usual state where the application is running and the service records the application actions. In order to perform replay, the application actions do not need to be idempotent but their re-execution must be deterministic. The record phase should record the actions input and the value of every non-deterministic behavior to turn their re-execution into a deterministic process. The recovery phase can have three phases: determine the affected actions and/or objects, remove these effects and replay the necessary actions to recover a consistent state, as already explained in Section 1.2. The recovery services that support *runtime recovery* do not require application downtime because the record and recovery phases can occur simultaneously.

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

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

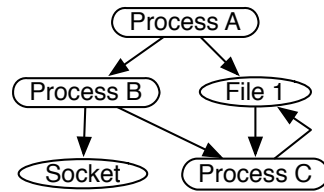
Dependencies are established during the record phase or at recovery time using object and action records. The level of abstraction influences the record technique and the dependency extraction method. The abstraction level outlines the recoverable attacks. In the next paragraphs, we explain the relevant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

## 1.2.2 Recovery at Operating System Level

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

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

Backtracker proposes a tainting algorithm that does tainting analysis offline, after attack detection, as follows. First a graph is initialized with an initial set of compromised processes or files,  $O_{tainted}$ ,



**Figure 1.3:** Dependency graph generated by BackTracker

identified by the system administrator. Then, Backtracker reads the log of system calls from the most recent entry until the intrusion moment. For each process, if it depends on a file or process currently present in the graph then the remaining objects dependent from the process are also added to the graph. The result is a dependency graph (Fig-Figure 1.3) with the objects, including  $O_{intrusion}$ , which the compromised objects depend from. The following dependency rules establish the graph edges:

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

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

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

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

The recovery phase is started with a set of tainted objects provided by an system 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,  $O_{intrusion}$ , which compromised the provided objects. After, at *propagation phase*, Taser transverses the dependency graph from the source objects of the attack,  $O_{intrusion}$ , to the current moment, adding all tainted objects to the set  $O_{tainted}$ .

Taser removes the intrusion effects loading a previous version of the tainted objects from a file-system snapshot,  $O_{snapshot}(t)$ . Then, to recover a coherent state, Taser performs *selective replay*, i.e., it replays, sequentially, the legitimate write operations of the tainted files since the snapshot. Non-tainted files remain unchanged. Since Taser does not checkpoint the state of processes neither the input of each system call, the system must be restarted to remove the current non-persistent states and processes must be replayed from the beginning to load their non-persistent state and perform their system calls with the correct state. This issue has a significant overhead specially for long-run processes as web servers.

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

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

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

During the recovery phase, RETRO requires the system administrator to identify  $A_{intrusion}$ , processes, system calls or  $O_{intrusion}$  objects which caused the intrusion. First, it removes the malicious system calls from the graph. Then it performs *taint propagation via replay*. To do so, it loads a previous version, from a snapshot, of the objects in  $O_{intrusion}$ . Then, the system calls, which are dependent from the restored objects, are replayed and their output objects are updated. The forward system calls, which depend from-on the updated objects, are also replayed while their inputs are different from the first execution. The propagation is done thought the output of system calls with different execution. The recovery process terminates when propagation stops. Since RETRO records the system call input, it can replay processes with system call granularity instead of process granularity, so the replay process may stop earlier. However, this mechanisms forces that the process re-execution has the same sequence of system calls as its first execution. Therefore, the process source code nether its sequence of actions can not change.

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

The efficiency of RETRO comes from avoiding to replay the actions if their input remains equal. However, the invoked system call must remain the same. Therefore, RETRO does not support a distinct process execution. RETRO requires human intervention to solve external inconsistencies. *Dare* solves it but it is limited to clusters where every operating system runs a *Dare* and RETRO daemon.



RETRO can not recover from an intrusion whose log files have been garbage collected or deleted. Dare supports distributed re-execution but, as RETRO, the affected machines must be offline because its propagation algorithm shutdown the service during the repair phase.

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

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

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

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

During the recovery phase, the server defines the contaminated processes using the client logs. A process is contaminated if it matches a set of rules similar to Backtracker [? ]. RFS adds the concept of contaminated file and contaminated file block. If a file is contaminated, all its blocks are contaminated. The reverse is not true, i.e., processes remain non-contaminated if they read a legitimate block of a contaminated file. RFS uses the version server to rollback only the affected files.

**Summary:** Dependencies at the operating system level are established by rules based on Back-Tracker [? ]. These rules are vulnerable to false positives and false negatives. While Taser [? ] and RFS [? ] recover from intrusions removing the effects only in corrupted files, RETRO [? ] removes the values written by  $A_{intrusion}$  and replays the forward system calls while their input changes. If  $A_{intrusion}$  is identified properly, then RETRO does not have false positives.

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

### 1.2.3 Recovery at Database Level

A vast number of database management systems (DBMS) support recovery by loading a snapshot (or full-backup) of the database, possibly patched with blocks of data modified since the snapshot was taken. However, this approach does not distinguish between malicious and legitimate requests. The recovery approach we are interested in the document is applied to databases similarly to what was seen in Section 1.2.2 for operating systems. While the operating system dependencies are established by the system calls, the database dependencies are established by transactions. First, we define the main dependencies in the database systems. These rules are used by most of the recovery services for databases and applications (Section 1.2.4).

Compromised transactions are determined from an initial set of bad transactions using read and write rules. "Transaction  $T_j$  is dependent upon transaction  $T_i$  if there is a data item  $x$  such that  $T_j$  reads  $x$ " [? ] and  $T_i$  performs the latest update on  $x$ . 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" [? ]. This dependency chain is broken if a transaction performs a blind write, i.e., the transaction writes an item without read it first.

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

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

**ITDB [? ? ? ]:** Intrusion Tolerant Database (ITDB) performs intrusion recovery in databases using compensating and supports *runtime recovery*, i.e., the database service remains available during the recovery process. ITDB uses the generic set of dependency rules mentioned in the beginning of Section 1.2.3 and extracts the read and write set parsing the **SQL!** statements.

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

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

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

An alternative approach concerns the throughput asymmetry between the recovery and the user flows. The asymmetry can be neutralized if the repair request priority is increased. Then, the availability is not compromised anymore but users may read tainted data and propagate the damage slower.

The ITDB architecture includes an **IDS!** (**IDS!**). The **IDS!** is application aware and acts at transaction level. Liu *et al.* propose isolation in terms of users: when a suspicious transaction is reported by the **IDS!**, every write operation from the suspicious user is done in isolated tables.

The ITDB does not perform transaction replay during the recovery phase. Therefore, it ignores that legitimate executions can be influenced by the updated values. Liu *et al.* propose a theoretical model [? ] based on possible workflows and versioning. However, predict every possible workflow requires extensive computational and storage resources.

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

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

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

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

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

The data item granularity affects the runtime performance overhead and the accuracy of dependency tracking. Coarser granularity, e.g., row, results in lower performance overhead but a higher probability of false dependence if two transactions read and write different portions of the same data item. Moreover, an attack can compromise just an independent part of the transaction and legitimate data is removed.

## 1.2.4 Recovery at Application Level

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

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

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

The recovery phase is as follows. First, the system administrator, using the logged data, identifies the malicious requests  $A_{intrusion}$ . Then, using a dependency graph, it determines the tainted requests. The dependency between transactions is established in a similar manner to the Section 1.2.3 but using table granularity instead of row granularity. Such coarse-grained approach may generate many false dependencies. Therefore Goel *et al.* use *taint propagation via replay*. Moreover, it proposes to modify the PHP-interpreter to reduce the false dependencies between transactions using a variable-level tainting. In other words, variables that read tainted rows or fields are also tainted; rows or fields written by tainted variables are tainted. This process increases the precision of the set of tainted requests. Finally, the compensation transactions of the tainted requests are applied in reverse serialization order on the current state of the database to selectively revert the effects of the database operation issued by the tainted requests.

**POIROT [? ]:** POIROT is a service that, given a patch for a newly discovered security vulnerability in a web application code, helps system administrators to detect past intrusion that exploited the vulnerability. POIROT does not recover from intrusions but proposes a tainting algorithm for application code. During the normal execution, every user request and response is stored. The log of each request includes the invoked code blocks. After the attack discover phase, the software is updated to fix its flaws. POIROT identifies the changed code blocks and requests dependent from them during the normal execution. The affected requests are replayed. During the re-execution phase, each function

invocation is forked into two threads: the updated and non-updated version [? ]. 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. These concepts are used in Warp [? ].

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

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

Warp includes a time-travel versioned relational database. Each row is identified using a row ID and includes a *start time* and *end time* timestamp columns that establish the row validity period. Warp reverts the intrusion effects, in a specific row, loading one of its previous versions. Warp supports concurrent repair using more two integer columns to define the begin and end of each *repair generation*. At repair phase, the current repair generation ID is incremented to fork the database. User requests are performed in the current generation while recovery requests are performed in the next generation. After replaying the requests retrieved until the beginning of the repairing process, the server stops and applies the remain requests.

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

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

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

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

Aire prevents the recovery process from locking due to remote servers downtime. To achieve this goal, the pending repair requests are queued until the remote server is recovered. Since clients may see a partial repaired state, Aire proposes a model based on eventual consistency [? ? ]. The model allows clients to observe the result of update operations in different orders or delayed during a period called *inconsistency window*, i.e., until the remote server recovers. Aire considers the repair process as a concurrent client. To repair key-value database entries, Aire creates a new branch [? ]

and re-applies legitimate changes. At end of local repair, Aire moves the current branch pointer to the repaired branch.

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

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

The Aire approach using eventual consistency requires the developer to concern the conflict resolution. Moreover, Aire recovers third-party web-services, which must have an Aire daemon. Aire requires the system administrator to pinpoint the corrupted requests. Finally, Aire, as Dare [? ], requires application downtime during the recovery process.

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

Undo for Operators proposes a proxy interposed between the application and its users. The proxy intercepts application requests. Since requests must be ordered to replay, Undo for Operators defines the concept of *verb*. Each protocol operation has its own *verb* class. A verb object encapsulates a single interaction (request/response) of the user and exposes an interface to establish the order between requests and their dependencies. However, the proposed architecture is protocol dependent. The proxy implementation only supports **IMAP!** (IMAP!) and SMTP. A new protocol requires a new set of verbs.

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

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

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

During the recovery process, Undo for Operators replaces the current system state by a system wide snapshot. Consequently, it does not support runtime recovery. Moreover, it relies on protocol knowledge to establish dependencies, compensating actions and to sort the requests during the normal execution and recovery phase. Therefore, any protocol change requires modifying the supported

verbs. Intrusions can use corrupted requests which are not defined as verb classes and cause a system fail.

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

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

## 1.3 Chapter Summary

Previous subsections describe various services that use different approaches to recover from intrusions. The level of abstraction outlines the recorded elements. Intrusion recovery services at the operating system level record the system calls and the file system. At the database level they track the transactions. At the application level they track the transactions, requests and code execution (Table 1.1). The log at operating system level is more detailed but may obfuscate the attack in false positive prevention techniques. At database and application level, the log is semantically rich but does not track low level intrusions.

Most of services require the system administrator to identify the initial set of corrupted actions or objects (Table 1.1). The system administrator may be supported by an **IDS!** (IDS!). The alternative, proposed in Warp [?], tracks the requests which invoked modified code files. The identification of the actions or objects can incur on false positives and negatives due to system administrator mistakes. Tracking the invoked code files requires to change the interpreter (JVM, Python, PHP) but avoids false positives and negatives.

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

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

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

**Table 1.1:** Summary of storing and intrusion tracking options

At recovery phase, the intrusion recovery services remove the intrusion effects and recover a consistent state (Table 1.2). Three of the possible options to remove the intrusion are: snapshot, compensating and versioning. Versioning is a per-entry and per-write snapshot. This mechanism is finer-grained than snapshot and allows the reading of previous versions without replay the actions after the snapshot. However, the storage requirements to keep the versions of every entry in a large application can be an economic barrier. The usage of actions compensation requires the knowledge of the actions that revert the original actions.

To recover a consistent state, intrusion recovery services should replay the legitimate actions dependent from the intrusion actions (Section 1.2.1). While Undo for Operators [?] propose to replay all legitimate user requests sorted by an application-dependent algorithm, the remaining services, which perform replay, use tainting via replay to selectively replay only the dependent actions. The late approach may hide indirect dependencies [?] but recovers faster. Undo for Operators [?], replays every request generating conflicts that must be solved in order to achieve a consistent state. Warp [?] queues the conflicts for later solving by users. Proposals [?] support runtime recovery (Section 1.2.1). This characteristic is required to support intrusion tolerance but allows intrusion to spread during the recovery period. To prevent that, ITDB [?] denies the access to tainted data. However, it compromises the availability during the recovery period.

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

**Table 1.2:** Summary of state recovery options



