Shuttle: Intrusion Recovery for PaaS

Dário Nascimento, Miguel Correia

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{dario.nascimento,miguel.p.correia}@tecnico.ulisboa.pt

Abstract—The number of applications being deployed using the Platform as a Service (PaaS) cloud computing model is increasing. Despite the security controls implemented by cloud service providers, we expect intrusions to harm such applications. We present Shuttle, a novel intrusion recovery service. Shuttle recovers from intrusions in applications deployed in PaaS platforms. Our approach allows undoing changes to the state of PaaS applications due to intrusions, without loosing the effect of legitimate operations performed after the intrusions take place. We combine a record-and-replay approach with the elasticity provided by cloud offerings to recover applications deployed on various instances and backed by distributed databases. The service loads a database snapshot taken before the intrusion and replays subsequent requests, in parallel as much as possible, while continuing to execute incoming requests. We present an experimental evaluation of Shuttle on Amazon Web Services. We show Shuttle can replay 1 million requests in 10 minutes and that it is possible to duplicate the number of requests replayed per second by increasing the number of application servers from 1 to 3.

I. Introduction

Platform as a Service (PaaS) is a cloud computing model that supports automated configuration and deployment of applications [1], [2]. PaaS offerings, such as Windows Azure, Google App Engine, and Force.com, provide an environment in which clients (*tenants*) build and run their application on a managed cloud infrastructure through a set of services. These services are paid-per-usage and turn the application easy to deploy and scale.

The number of applications running in cloud computing platforms, including those based on the PaaS model, is increasing rapidly. Many of these applications are critical for their companies, so the risk of intrusion is high. The recent case of the cloud-based Code Spaces service is conspicuous: hackers deleted most of its data and backups, leading to the termination of the service [3].

Cloud service providers (CSPs) implement several security controls. Most of these controls aim to prevent intrusions: access control, firewalls, intrusion detection and prevention systems, network access control, vulnerability scanning, etc. Despite the importance of these mechanisms, applications often contain design or configuration vulnerabilities that let intrusions happen [4] . Complexity and budget/time constraints, weak users passwords or bad security policies are known causes of these problems. The recent case of the bash bug (or Shellshock) shows that there are other reasons such as legacy software being used in ways that were unpredictable when it

was developed [5]. Much research has been done on mechanisms to tolerate Byzantine faults, including intrusions [6]. However, most of these techniques do not prevent application level attacks or user mistakes. For instance, if attackers steal legitimate user credentials, they are able to modify the state of the applications violating their security policy.

We assume intrusions can happen and their effects need to be removed from the applications' state. This removal is often done manually by system administrators who have to understand the parts of the state compromised directly by the intrusion or contaminated by operations that used compromised state, and clean the state manually. This process is error-prone, often takes long and causes application unavailability [7]. Intrusion recovery systems aim to automate these steps and mitigate these issues.

Previous intrusion recovery systems targeted operating systems [8], [9], [10], databases [11], [12], web applications [13], [14], [15] and other services [16]. Yet, none of them was designed for cloud applications, which are often deployed in multiple servers and use background databases. Furthermore, most cause downtime, which is undesirable in online services.

We present a novel *intrusion recovery* service for PaaS systems. Shuttle aims to make PaaS applications operational despite intrusions, helping tenants to recover their applications from software flaws and malicious or accidentally corrupted user requests without requiring application downtime during the process. When an intrusion is detected, tenants can use Shuttle, that is provided as a service by *Cloud service providers* (CSP), to remove intrusions' effects and recover the integrity of their applications. This paper concerns application availability and state integrity, not confidentiality.

Shuttle assumes a client-server model in which clients communicate with the servers in the cloud using HTTP / HTTPS. For each application deployed in the PaaS system, Shuttle records the requests issued by clients and creates periodic snapshots of the application database. After detection of the intrusion, Shuttle loads the snapshot that precedes the beginning of the intrusion and replays only the legitimate requests to recreate an intrusion-free application state. Requests are replayed asynchronously and, whenever possible, concurrently. The recovery process is deterministic because accesses to each data item are performed in the original order of execution.

Dependencies established at database level during the requests' first execution are used to create independent clusters of requests that can be replayed concurrently. We propose a

branching mechanism to keep the service executing incoming requests while doing replay.

Unlike previous intrusion recovery systems, Shuttle aims to be provided as a service to applications deployed in PaaS. Consequently, it can be well-tested and available without depending on being correctly setup by the application developers. We also leverage the elasticity of PaaS infrastructures to reduce the service costs and the recovery period. Specifically, Shuttle is designed to allocate more servers during the recovery period to accommodate the throughput of requests being replayed, and release them at the end, with a proportional impact on service costs. The rapid and continuous decline in computation and storage costs in CSPs makes affordable to store user requests, to use database snapshots and to replay previous user requests.

The contributions of this paper are the following: (1) a new intrusion recovery approach provided as a service integrated in a PaaS system and taking into consideration applications running in various instances backed by distributed databases; (2) a method to order the replayed user requests considering their accesses to databases; (3) accomplishing intrusion recovery without service downtime using a branching mechanism; (4) leveraging the resource elasticity and pay-per-use model in PaaS environments to record and launch multiple clients to replay previous non-malicious user requests as concurrently as possible to reduce the recovery time and costs; (5) a mechanism to do a globally transaction-consistent snapshot of NoSQL databases.

II. BACKGROUND

This section formalizes several approaches to perform intrusion recovery. An application execution is modeled as a set of actions A on a set of objects D. Actions are described by operations (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 define $A_{intrusion}$ as the subset of actions of A whereby the attacker compromises the application during the intrusion, A_{after} as the subset of actions that began after the intrusion began (including the first action of the intrusion), and A_{legal} as the subset of legitimate actions in A, i.e., $A_{legal} = A \backslash A_{intrusion}$.

Intrusion recovery services aim to remove the effects of malicious actions setting the application state to a state setup only by legitimate actions. The recovered application state shall respect the application specification (correctness) [17]. A *backup* mechanism is a basic recovery service that can set objects to the state they had before an intrusion began. The new state excludes the effects of the attacker's actions, but also the effects of any legitimate actions performed after the backup was done. This second aspect is undesirable, so intrusion recovery systems aim to avoid it.

An action is considered *tainted* at a certain instant if it is one of the attacker's malicious actions or if it reads an object written by a tainted action (called a *tainted object*). Since actions are contaminated by malicious actions through objects, to remove the state written by tainted actions it is necessary

but not sufficient to obtain the state produced by the legitimate actions.

Consider a hypothetical application execution. At a certain point in time after an intrusion, the application is stopped and the sequence of actions executed (A) is purged of the actions in $A_{intrusion}$. Then, the state is rewind to the beginning of the application execution and only the actions in $A_{recovered} = A \backslash A_{intrusion} = A_{legal}$ are re-executed. That re-execution is intrusion-free as $A \cap A_{intrusion} = \emptyset \implies A_{tainted} = \emptyset$. The set of tainted actions of the first execution that are not in $A_{intrusion}$ are re-executed but read different values (not imposed by the attacker) and have a different execution. The replay process restores the application to a correct state $D_{recovered}$, which is not compromised by the intrusion.

That generic approach is unfeasible without the use of snapshots. The sequence of actions performed before the intrusion $A_{before} = A \backslash A_{after}$ can be long and each action takes non-null time to execute, so replaying A_{before} may be unfeasible. Moreover, a log of all the actions executed may be too large. We define the subsets $D_{snapshot}(t)$ and $A_{snapshot}(t)$: $A_{snapshot}(t) \subset A$ as the subsets of objects and actions executed before the begin of a snapshot operation at instant t. If the intrusion happens after t, then $A_{after} \cap A_{snapshot}(t) = \emptyset \implies (A_{intrusion} \cup A_{tainted}) \cap A_{snapshot}(t) = \emptyset$, i.e., the snapshot is not affected by the intrusion. For that reason, the service can replay only $A \backslash A_{snapshot}(t) \backslash A_{intrusion}$ using the object set $D_{snapshot}$ as base.

There are two distinct replay approaches to update the set of object D to $D_{recovered}$, selective replay and full replay. A *version* is a snapshot of a single object value at an instant t. Versions can be recorded with the sequence of actions that write the objects before the instant t. The *selective replay* approach loads only the versions of the tainted objects, $D_{tainted}$, previous to the intrusion, instead of loading a previous version of every object [8], [14], [18]. Then, it replays only the legitimate actions, which were tainted, $A_{tainted} \backslash A_{intrusion}$, to update the objects in D. The state of the objects in $D_{legal} \backslash D_{tainted}$ remains unmodified. The other approach, *full replay* [16], loads a snapshot previous to the intrusion moment and replays every action in $A \backslash A_{snapshot}(t) \backslash A_{intrusion}$. This approach is slightly simpler than the other, but in general takes longer to execute.

Recovery services have two distinct phases: record and recovery. 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 (given the same initial state they produce the same final state). 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 be subdivided in three: determining the affected actions and/or objects, removing these effects, and replaying the actions necessary to recover a consistent state. In this paper we present a recovery service that supports *runtime recovery*, i.e., that allows the record and recovery phases to

occur simultaneously.

III. Architecture

We introduce Shuttle, an intrusion recovery service for PaaS. Our goal is to help PaaS tenants to recover from the following problems in their applications:

- *Software vulnerabilities:* non-authorized users compromise state by exploiting software vulnerabilities, e.g., allow invalid requests to be executed.
- *Malicious or accidentally corrupted requests:* users, authorized or not, compromise the application state accidentally or intentionally issuing valid requests.

For instance, two common attacks that can be used to compromise application state consist in: (1) attackers stealing valid users' credentials and using them to access their data; and (2) doing a SQL Injection attack by mixing SQL meta-characters with normal input and doing otherwise invalid queries to the database. Both attacks can be performed using apparently valid requests, so many prevention mechanisms fail to block them.

Applications supported by Shuttle can operate in one of two phases: *normal execution* and *recovery*. During *normal execution*, Shuttle records the data required to recover the application afterward: it does periodic database snapshots, logs user requests and database accesses. When an intrusion is identified, tenants use Shuttle to recover their applications staring the recovery phase.

The processes described in Section II lead us to define *how to remove the intrusion effects* and *how to recover a consistent state*. During the *recovery phase*, Shuttle removes the intrusion effects creating a branch of the system execution in which it loads a snapshot, which contains a state O before the intrusion began. It builds a consistent state replaying (re-executing) in the new branch, the legitimate requests logged during the *normal execution*, performing either selective or full replay (Section V). In the meantime, the incoming requests are executed in the previous branch. When ready, it sets the new branch as the single execution branch.

PaaS platforms offer services to build, deploy and manage applications. Shuttle aims to be integrated by *Cloud service providers* (CSP) into their PaaS architecture as a novel service. Services provided in PaaS are expected to be well-tested and available without setup because offered by CSP and shared by multiple tenants. We consider a minimal PaaS architecture to let Shuttle as generic as possible. We consider a client-server model in which clients access applications using the HTTP protocol [1]. HTTP requests are received by a load balancer that forwards them to web/application servers, which access a shared database. PaaS components are represented with solid line in Figure 1, while Shuttle components are represented with dashed line. The components of a PaaS platform with Shuttle are:
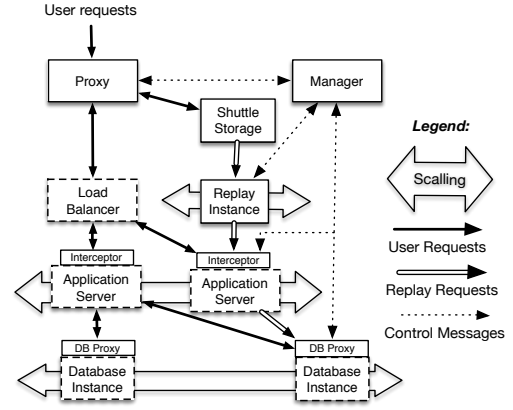
[1]Shuttle also supports HTTPS by ending the connections at the proxy.

Figure 1. Service architecture: The dashed line components are part of the PaaS architecture.

- *Proxy:* Logs every HTTP user requests, adds a unique mark to its header and forwards it to the load balancer. The proxy functionality might be part of the load balancer but conceptually it is a different component.
- *Load balancer:* Routes requests to different application servers taking into account their load (part of the PaaS platform).
- *Application servers:* The application (or web) servers are the components of the PaaS platform that run the application logic. This logic uses a library to access the database service. Shuttle uses a *database client interceptor* mechanism in this library to log the data items accessed per request.
- *Database instances:* A set of database servers used to store the application persistent state. Shuttle includes in each instance a *database proxy* that logs the requests that accessed each data item and determines the dependencies between requests.
- *Shuttle storage:* A scalable storage component that stores requests, responses and metadata.
- *Manager:* Retrieves dependencies and coordinates the recovery process.
- *Replay instances:* A set of HTTP clients that read previously executed requests from the Shuttle storage and invoke the application servers to re-execute the requests during the recovery process. These worker instances are coordinated by the manager.

PaaS offerings are supported by a computing infrastructure, often provided as a service (IaaS model), able to scale the application allocating new instances on-demand or automatically, to maintain the quality of service despite demand oscillations. This elasticity allows to allocate replay instances and to scale the application to attend the requests issued by them during the recovery process. Due to the common pay-per-usage model, these resources are payed only when a recovery process occurs. The remaining cost of the service comes from storing client requests and database snapshots. Our design aims to optimize the available resources to reduce the recovery period
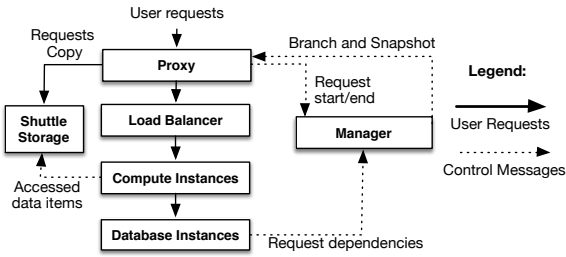
Figure 2. Interaction between components during normal execution

and costs.

Unlike previous works, our design encompasses distributed databases. We assume without loss of generality that applications store their state in distributed key-value stores, such as Dynamo [19], where the values are often accessed using a *create, read, update and delete* (CRUD) API. The simple API reduces the performance overhead to track accesses while the independence between keys turns Shuttle into a scalable service. Shuttle can be extended to support other NoSQL schemes.

The *Shuttle storage* keeps the content of the user requests and responses. Although we do not consider this aspect in the architecture, this store can be replicated to a remote site to allow tolerating catastrophic failures in a datacenter.

We consider the Shuttle components to be part of the trusted computing base since their integrity and availability are critical to recover the application. We assume that intrusions tamper the application data, which is stored in the database.

IV. NORMAL EXECUTION

Shuttle logs the data it needs to recover applications during the normal execution phase: user HTTP requests, application HTTP responses, database items accessed by each request and sequence of operations to each database item (Figure 2). In this section, we describe the normal execution phase following the path that a request takes to be processed.

A. Proxy

The proxy intercepts all user HTTP requests, except those to static contents (e.g., images), and adds a new header field named *Shuttle Request Data* (SRD). Each SRD contains three subfields: *Request ID* (RID), which is an unique timestamp; *Branch* and *Snapshot*, which define, respectively, the database branch and snapshot (Section IV-C) and a *restraint* flag, which is used to support runtime recovery (Section V-H).

The proxy also intercepts every application response, associates the response with the original request and adds a new timestamp to track the ending of the request execution. Requests, responses and their timestamps are stored in the *Shuttle Storage* using asynchronous I/O, which permits the operations to proceed before the transmission has finished.

B. Application server and database instances

In order to associate every database operation with the *Shuttle Request Data* (SRD) of the source HTTP request, we propose to modify the application server adding it an *interceptor*, which is invoked before and after the request processing. The database client library invokes the interceptor at begin of each operation to get the SRD and logs the accessed data items. The post-process interceptor stores the log of accessed data items in the *Shuttle Storage*. This approach avoids to modify tenants applications.

Every database operation involves calling the *database proxy*. The proxy logs the operation's *Request ID* (RID) and type (put, get, delete) and uses its *branch* and *snapshot* subfields to select the correct data item version. The sequence of operations to a data item defines its *operation list*. The access order is ensured by a read-write lock that serializes the access to the value of the data item allowing multiple reads but only a single write concurrently. This pessimistic concurrency control may decrease the system performance in comparison with a multi-version concurrency control, in which the RID would be written among the value and parsed during the read operation. Nevertheless, we used this form of concurrency control for the sake of simplicity.

The *operation list* and lock of each data item are stored in a Hash Table that supports concurrent accesses.

Periodically, each database instance iterates the operation list of every data item to establish the dependencies between requests. The manager retrieves, asynchronously, the start and end timestamps of each request, collected by the proxy, and its dependencies, collected by the database instances, to generate the dependency graph (Section V-B).

C. Snapshot

A *snapshot* is a set of versions of every data item in the database. Our service loads a snapshot to remove the intrusion effects and replays the later requests to recover an updated application state.

Performing snapshots in distributed key-value NoSQL databases like Dynamo is not trivial since snapshots have to be consistent with the user requests. We consider each user request may include multiple database accesses, each of them to multiple database servers, without using transactions. If Shuttle replays a requests on a snapshot that contains part of the persistent state written by a request during its first execution, the replay will be inconsistent. Therefore each snapshot shall be *global request-consistent* containing either all or none of the database updates made by every request [20].

The snapshot shall be non-blocking: applications shall not stop their execution while taking snapshots. For instance, a straightforward way to take a request-consistent global snapshot is to stop processing new requests, waiting until the currently executing requests finish, and then making a copy of each data item. However, this solution causes application downtime.

Our solution leverages the existence of a single load balancer and, consequently, a single proxy that adds a *Shuttle Request Data* (SRD) field to every request. Every SRD contains a RID (the instant when the request is retrieved). In order to create a snapshot, tenants define a future instant in time t when the snapshot will occur. The instant, named *Snapshot ID* (SID), identifies the snapshot. The manager passes the SID to every database proxy.

Database proxies use the SID to define the version of the data item used by the operations. Operations with *Request ID* (RID) lower than the scheduled snapshot instant (RID ¡ SID) access the version before the snapshot. Otherwise, the operations access the latest data item version. This mechanism splits requests to accomplish a request-consistent global snapshot, and allows tenants to schedule snapshots without application downtime.

We avoid blocking the application to copy the versions using a copy-on-write and incremental method: a new version is created only when the data is written for the first time in each snapshot. Since a data item may not be written in every snapshot, we associate a *version list* to every data item. A *version list* tracks on which snapshots the data item has been written.

A snapshot might become inconsistent if a request with RID greater than the snapshot instant SID read a version belonging to the snapshot SID and a concurrent request with RID lower than SID overwritten that version. Storing a new version and adding a flag on the version list solves the problem. Nevertheless, we expect this to happen only in rare occasions.

V. RECOVERY

Tenants initiate a recovery when they detect intrusions. When Shuttle enters recovery mode, it generates a list of requests to replay and asks for the PaaS controller to launch a set of *replay instances*. Shuttle may also ask for additional database and application server instances, or they may be launched automatically by the PaaS platform when it detects additional load, in case auto-scaling is supported. A non-tampered snapshot, which is previous to the intrusion instant, is selected. The multi-thread HTTP client of each replay instance fetches requests from the *Shuttle Storage* and sends them to the application servers, concurrently whenever possible. After replaying all requests issued before the beginning of the recovery, the manager sets the proxy state to *restraining mode* and commands the replay instances to replay the remaining requests (those issued after recovery began). Then, *restraining mode* is disabled. The following sections explain this process in detail.

A. Intrusion Identification

The recovery process starts when an intrusion is detected. Intrusions may tamper the database or the application server instances . In order to fix the vulnerabilities that may have lead to intrusions, Shuttle supports the following actions: 1) update the application software; 2) identify a set of tampered database items; 3) add, modify or remove logged requests; 4) launch cleaned database or application server instances.

If tenants update the application software, they have to ensure that the application's interface remains compatible with the requests that will be replayed. If the database is tampered using user requests, the tenant has to identify the malicious user requests. For instance, the tenant can provide the set of suspicious database items to Shuttle and it will resolve the set of requests that accessed the suspicious items after the estimated intrusion moment. Knowing the suspicious requests, the tenants shall use Shuttle to add, modify or remove the past requests to remove accidental or malicious behaves.

B. Dependency graph

A *dependency graph* consists of nodes that represent requests and edges that establish dependencies between them (Figure 4). Dependencies between requests are established using the following rules: 1) a request R_A is dependent upon request R_B if there is a data item x such that R_A reads x and R_B performs the latest update on x; 2) dependencies are transitive except when requests perform blind writes, i.e., requests write items without read them first [11]

Previous solutions for relational databases extract the dependencies using a pre-defined per-transaction type template [11] , or change the relational database management system code to extract read dependencies [12]. In contrast, Shuttle uses the database proxy to log the database accesses. Periodically, each database proxy traverses, in background, the *operation list* of each data item to collect the new accesses and to generate the dependencies between requests. The Shuttle manager processes the dependencies to update the dependency graph. An alternative approach is to pull the dependencies from each database node only before the recovery process and generate the dependency graph when needed. The dependency graph is implemented as a hash table. Keys of the hash table are the *Request ID* (RID). Each value of the hash table contains the requests that depend on the associated request, i.e., the requests that execute after this one. A scalable implementation can use a distributed hash table or a graph-oriented database.

The above method may lead to *false positives*, i.e., to flag dependencies that do not exist. For instance, a request may read a data item but not use it to compute the written value, so there is no real dependency. Although tracking variables used by each request during its execution might solve this particular case [13], it would require modifying the code interpreter (e.g., Zend Engine for PHP), which would constrain Shuttle to a set of specific languages. As our approach uses the dependencies to group the requests that can be executed concurrently, false dependencies imply a performance penalty but do not cause data loss or inconsistent state.

Complex queries on a relational database may lead to *false negatives*, for instance when a read operation would have been executed on a deleted data item if this data item had not been deleted before the request execution [21]. In contrast with SQL queries that access the data items that match a query, the CRUD interface of most key-value stores specifies, in a

deterministic and apriori manner, the data item that will be accessed. Shuttle logs every access, even when the data items do not exist, keeping the *operation list* of the deleted data items to track further operations.

C. Replay

Shuttle aims to support a large range of applications in which the user requests access a distributed database without using transactions. This assumption contrasts with previous recovery systems that collect the order requests based on the semantics of the application they consider (e.g., [16]) or leverage the serialization provided by snapshot isolation to do it (e.g., [13]). Moreover, requests executed concurrently during normal execution, may depend on each other, e.g., the first reads an item written by the second and the second reads an item written by the first.

We propose a new approach to order the requests for replaying that consists on sorting the requests per *start-end order*, instead of using a dependency graph. Requests are replayed ordered by their start instant. Moreover, if a request starts before the end of another request, then they were executed concurrently and they are also re-executed concurrently. Yet, re-execution of concurrent requests is not deterministic, e.g., due to multi-thread servers, messaging systems, etc. Therefore our novel approach uses the *operation list* to make parallel replay deterministic, by forcing operations to a data item during replay to follow the order established by its operation list (Figure 3a).

Modifications to the application code or to the sequence of requests may cause the application not to access the same sequence of data items or read/write the same content during the replay phase (Figure 3b). If an operation contained in the operation list is not performed, the following operations to the data item are blocked. To address this problem, at the end of each request execution, the *database client interceptor* fetches the list of data items accessed by the request on its first execution and compares them against the ones accessed during the replay process. The database client library invokes the *database proxy* with the keys that have not been accessed to unlock the remaining requests.

During replay there may be non-deterministic situations, whenever an access is not contained in the operation list. Consider the case of Figure 3. Figure 3a represents the first execution of two requests. During the recovery period, the intrusion was removed hence requests access different data items than in the first execution. If the *req. 2* started after the end of *req. 1*, the replay would consistent with the first execution (Figure 3c). An approach using only the dependency graph would have inconsistent results because the new dependency happens at recovery time.

The problematic scenario is that the two requests were executed concurrently during their first execution: the *start-end order* defines that *req. 1* and *req. 2* are replayed concurrently (Figure 3d). The access to A remains consistent with the first execution (*req. 2* after *req. 1*), since the accesses are constrained by the *operation list* of A. However, the final value

of B is unpredictable because *req. 1* may write B before or after *req. 2*. Since both requests did not access the data item B during their execution, the operation list does not establish an access order. Therefore, the *req. 1* and *req. 2* may execute in a arbitrary order. The order of these requests is as deterministic as if during the first execution: the operation of *req. 1* can execute before, between or after *req. 2*.

In order to turn the replay process more consistent with the first execution, we leverage semantic reconciliation, as in Dynamo [19]. The case represented in Figure 3d is equivalent to a concurrent update, where two parallel writes are performed on distinct database instances. Each request writes a distinct version resulting in conflicting versions of an item. Developers use the application-assisted conflict resolution interface to merge the versions. In this case, the following read operation would access the values written by the latest operation. In Figure 3d, *req. 2* could choose between 5 and 1.

The application shall be deterministic. An application is said to be deterministic if two subsequent executions with the same initial state and user inputs can be guaranteed to have the same final state and outputs. Five of the main sources of non-determinism are: shared memory, thread concurrency, random number generation, timestamps, and message exchanging. We assume requests to be independent thus they do not share memory, and that concurrent threads are independent. The API provided by Shuttle provides deterministic random number generation and timestamps using the *Request ID* (RID) as timestamp and pseudo-random number seed, so the replay of a request will use the same random numbers and timestamp (we consider a single timestamp per request to be enough for most applications). This mechanism is language independent. User requests and database accesses are ordered in a deterministic way using the operation list.

D. Clustering

Our preliminary experiments have shown that replaying requests concurrently can reduce the recovery period. We want recovery to take a fraction of the time elapsed since the snapshot from which recovery starts (e.g., if the snapshot was taken a week before, we want recovery to take much less than that period).

We address this problem grouping the requests into *clusters*. A cluster is a set of requests that have dependencies between them but not from/to requests in other clusters. Clusters are created when the recovery is about to start by inspecting the dependency graph. Since clusters are independent, they execute concurrently by different *replay instance* without synchronization. Requests within the same cluster, are performed in start-end order (Section V-C). Given that more requests are executed concurrently, Shuttle launches more application servers and database instances to process the replayed requests. Therefore, the replay phase throughput is bigger than during first execution and the recovery time is minimized. This mechanism is applicable if the graph dependencies remain unchanged during the recovery phase, i.e,. all replayed operations

(a) Ordered by operation list   (b) Operation unlock   (c) Consecutive requests   (d) Conflict
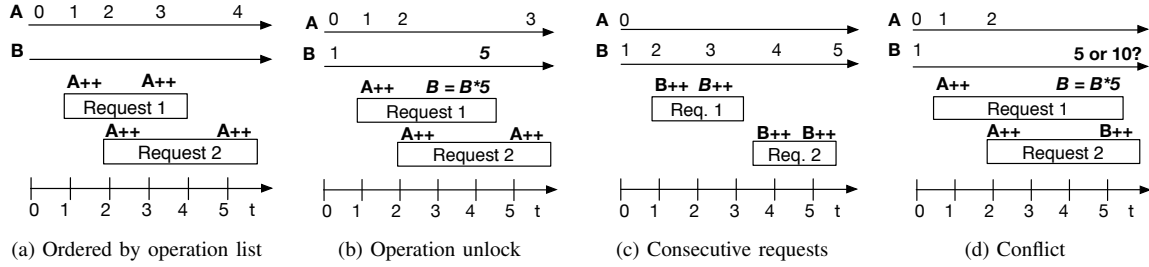
Figure 3. Replay two requests with different re-execution

are contained in the operation list but not all operations in the list must be replayed.

E. Full and Selective Replay

We propose two approaches for intrusion recovery: full replay and selective replay. Full replay consists in replaying every request done after the snapshot. Executing many requests takes considerable time, so this approach is adequate for issues detected reasonably fast after they happen, e.g., a few days or weeks. Selective replay re-executes only part of the requests so it is faster than full-replay. However, it requires tenants to provide a set of malicious actions (i.e., requests) $A_{intrusion}$. This set is used to deduce the set of tainted requests $A_{tainted}$. A request is said to be tainted if it is one of the attacker's requests or if it reads objects written by tainted request [8], [11], [12]. The selective replay process is as follows (full replay is simpler so we skip it):

1) *Determine the malicious requests* $A_{intrusion}$. Based on initial data such as user session compromised or data items accessed, the tenant determines the requests $A_{intrusion}$ used by the attacker to compromise the application. For instance, $A_{intrusion} = \{R.4\}$ (Figure 4).

2) *Use* $A_{intrusion}$ *to determine the set of tainted requests* $A_{tainted}$. For each request in $A_{intrusion}$, traverse the dependency graph in causality order and add these nodes to $A_{tainted}$ (in the figure $A_{tainted} = \{R.5, R.6, R.7, R.8\}$).

3) *Get the requests needed to obtain the values read by* $A_{tainted}$ *and their effects.* Instead of storing the input and output of every action or versions of every data item, we propose to replay the actions which $A_{tainted}$ depends on. The data item value is known at the snapshot instant so the algorithm transverses the graph in inverse causality order from each request in $A_{tainted}$ and stores the requests in A_{replay} ($A_{replay} = \{R.2\} \cup A_{tainted}$). A_{replay} is expanded by traversing the graph from each of its elements on causality order to determine the requests which can be affected by the re-execution of A_{replay} ($A_{replay} = A_{replay} \cup R.3$). Requests subsequent to the first snapshot after the latest malicious request may not be repeated if all their read operations read the value contained in the snapshot (version read by $R.8$ is stored in snapshot B).

4) *Determine the replay order.* The set A_{replay} is sorted on non-clustered *start-end order*.

5) *Load the previous data item versions* Shuttle loads the version, which is previous to the selected snapshot, of the
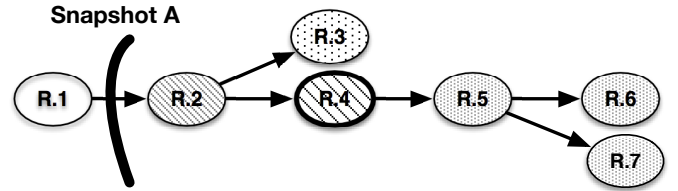

Figure 4. Dependency graph: $R.1$ is previous to a snapshot A; $R.3$ is dependent on $R.2$, which is replayed to get the read values; $R.4$ is a malicious request; $R.5, R.6, R.7$ are tainted;

data items read by the requests in A_{replay}.

6) *Replay the requests* Requests in A_{replay} are replayed. If an access is not contained in operation list, then a new dependency is established and the requests that accessed the data item during the first execution are also replayed as in *taint propagation via replay* [9]. For instance, $R.9$ is replayed if it reads an item written during recovery process but not during normal execution.

F. Consistency

An important aspect of a recovery system like Shuttle is the application consistency seen by users. For instance, if a user does an action based on data written by a malicious action, which result of the user action replay is consistent? Since users have a non-deterministic behavior, they may have to be notified if a recovery took place and their data was modified.

Shuttle does not execute requests that returned an error in the first execution. Similarly to other works in the area [16], we assume that these cases are compensated by the user when they happen. As only requests that did not return an error are replayed, Shuttle considers an inconsistency when a request returns an error or a response is different during replay. Shuttle provides the following API for the application programmer to define how inconsistencies are dealt with (Shuttle calls these functions in case they are launched by the tenant):

1) *preRecover():* invoked before the beginning of the recovery process.
2) *handleInconstency(request, previous response, new response, previous keys, new keys, action):* invoked when there is an inconsistency.
3) *postRecover(statistics, old version, new version):* invoked after the end of the recovery process.

The first function allows tenants to perform a set of actions before the beginning of the recovery process, such as
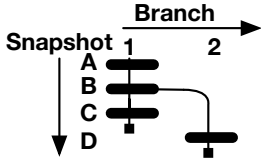
Figure 5. Tree model: 3 branches and 5 snapshots: branch 1 contains snapshots A,B,C; Branch 2 contains D; Branch 3 contains E.

notifying the operations team or taking a new snapshot. The second function takes as input the operation that caused the inconsistency as well as the response and keys accessed during the normal execution and during the recovery process. It also takes as argument the action to take. Currently we consider three possible actions: 1) ignore the inconsistency; 2) notify the user of the inconsistency; 3) execute another request. Using the *postRecover* function, the tenant has access not only to the statistics of the recovery process but also to an interface to compare the database values before and after the recovery process and the application responses, before exposing the data to the users.

Besides its users, an application may also interact with external services. The problem is treated in [15], [16].

G. Instance Rejuvenation

Attackers may exploit system vulnerabilities to tamper application server or database instances, affecting the application integrity or availability. Shuttle interacts with the PaaS controller to rejuvenate instances when they are compromised. This process terminates the instances and launches new ones. The PaaS controller initializes the new instances with updated machine images and deploys an updated version of the application code, which may include updates to fix discovered flaws or prevent future intrusions.

We assume new instances to be intrusion-free as the image can be updated to fix previous flaws and their persistent state is renewed. Instances can be launched in a remote site to recover from catastrophic disasters [22]. Tenants are responsible for ensuring that request dependencies remain correct and the updated version API is compatible, or for providing a script to update each request to the new API. Moreover, the selected snapshot has to be consistent according to the updated version specification or every request executed since the application's begin shall be replayed.

This process can be used in a proactive manner to renew instances to remove unknown intrusions [23] or to test new application versions to compare its results against the previous version, using the branching mechanism (Section V-H).

H. Recovery in Runtime

Shuttle is capable of doing recovery on runtime, i.e., without making the application unavailable for a period of time. To do so, each recovery process is considered to be a new branch, a model inspired in versioning systems such as *git* [24]. A *branch* is a sequence of snapshots (akin to the commits in git). Figure 5 presents an example with 3 branches and 5 snapshots.

Each recovery process creates a new branch forking a previous snapshot chosen by the tenant, either explicitly or im-

plicitly (by indicating the initial intrusion instant, selecting implicitly the preceding snapshot). Incoming user requests access only the data of the previous branch keeping the application available, while replayed requests access the created branch without compromising the availability of the application. In addiction, tenants can use the branching mechanism to test their intrusion recovery procedures in background, i.e, without exposing users to test issues.

Since tenants can select previous snapshots, we define a branch as a sequence of non-tampered snapshot, named *branch path*. For instance, the snapshots E, B, A compose the branch 3 (Figure 5). Every database instance knows the *branch path* of the previous branch and the newly created branch in use by the requests being replayed.

Since a novel data item version is created only when the data item is written for the first time during each snapshot, the data item may not have a version for each snapshot (Section IV-C). Therefore, the version to accessed by an operation is defined using the branch path and the *version list* of the data item: operations read the latest version present in the *version list* and in the *branch path* and write the latest version in the branch path. A new version is added to the version list on the first key access to each data item during the replay. This mechanism allows mapping the request to the correct version. Since the *version list* keeps a pointer to the latest version and this reference is updated, the complexity of getting the correct version is $O(1)$.

At recovery time, the manager sends the new *branch path* to every database instance. The new incoming users access the, perhaps corrupted, old branch while the requests being replace access the new branch. Therefore, the application remains online, perhaps with a degraded behavior, without exposing downtime to users.

At some point, when the recovery is finishing, the user requests have to start being issued to the new branch. To do so, after replaying the requests, the proxy flag *restraining* is set and every new request is marked with the *restrain* flag. Database accesses marked with *restrain* are delayed. After replaying the requests retrieved during the recovery process, the proxy sets the new branch in the subfield *branch* of SRD of the new requests, the *restrain* flag is disabled and the database nodes are notified to proceed the accesses. This mechanism delays the processing of some requests, but this has typically a duration of seconds, compared with a recovery process that may take many minutes or even hours.

VI. EVALUATION

In order to evaluate our approach, we integrated a Shuttle prototype with AppScale and Voldemort. AppScale [25] is an open-source version of Google App Engine. Voldemort [26] is an open source implementation of Dynamo [19], developed and in use by LinkedIn. Shuttle's prototype has been developed in Java (1400 lines of code for the proxy, 1800 for the manager, 300 for the interceptor, 900 for the replay instances and 1800 for the database proxy).

	#intrusion	#tainted	#replayed (sr)	#replayed (fr)
1a	106	0	< 605	> 38620
1b	58	14	< 379	> 38620
1c	48	52	< 253	> 38620
2a	4 338	0	-	> 38620
2b	18 286	1 278	-	> 38620
3	2 000	-	-	> 38620

Table I
ACCURACY OF THE RECOVERY PROCESS (NUMBER OF REQUESTS)

A. Application Example: Ask

We developed a *Questions and Answers* (Q&A) web application for PaaS inspired on Stack Exchange [2] (1700 lines of code) to evaluate Shuttle. The application represents a generic web application that accepts requests and stores the persistent state in the Voldemort database. Its implementation is independent of Shuttle, i.e., Shuttle does not require the application to be modified.

The application semantics implies the following dependencies: a) questions are independent; b) new answers depend from previous answers and votes to the same question; c) new comments depend from the commented answer; d) news vote depend from the voted answer. We selected subsets of a dump of the Stack Exchange database [3] to simulate real-world requests.

B. Accuracy

We evaluate Shuttle's ability to correctly recovery applications in different scenarios. We consider three classes of intrusion scenarios: malicious requests, software vulnerabilities and external channels (e.g. SSH connections). The selected data subset contains 100 000 requests originally performed from 31 July until 12 Sep. 2008: 6992 questions, 28993 answers, 2220 comments, 61795 votes and 3062 tags. Requests were sorted per date, establishing 92 939 dependencies.

At intrusion moment, Sep. 2nd, the database contains 4338 questions, 18286 answers, 422 comments and 38334 votes (61380 requests). The attack is detected in Sep. 12th, assuming a pessimistic delay of 10 days. During this period, the application retrieved 38 620 requests. Table I represents the summary of the accuracy tests. It contains the number of data items tampered by the intrusion (*#intrusion*) and the number of user requests that read data items written by tainted requests or malicious requests (without considering the intrusion requests). Recovery using *full replay* requires to replay every request from the latest snapshot before the intrusion instant until the detection instant: in this example at least 38 620 requests (*#replayed (fr)*). Selective replay only re-executes tainted requests, unless some data item versions need to be recreated. On the worst case, the system does not contain any snapshot and every data read by the tainted requests shall be recreated (*#replayed (sr)*).

Malicious Requests. In the first class of scenarios, we consider three cases in which an attacker has stolen an user

credential, then: a) deleted every question created by the user; b) deleted every user answer; or c) modified every user answer.

1a) The attacker deletes the user's 4 questions, performing 4 delete requests that remove 106 associated comments and answers. The tenant identifies the malicious requests through the user session and selects a snapshot previous to the intrusion instant. Users cannot access deleted questions, so no request is tainted. If Shuttle has a snapshot containing the deleted questions, then *selective replay* does not need to replay any request and merges the deleted questions on the current system state. If the latest snapshot is previous to the creation of the 4 questions, then *selective replay* replays 605 requests to recreate the deleted questions, their answers and votes. The result is merged with the current branch, rebuilding the deleted questions.

1b) Deleting the user's 48 answers implies that 58 data items are deleted and 14 answers and comments are tainted as they execute after the intrusion instant answering and voting without knowing some answers. If a snapshot containing the user answers exists, then the *selective replay* approach replays only 14 tainted answers and comments. Otherwise, it replays 379 requests: the total number of requests to recreate the tainted questions and then merge the result.

1c) 48 data items are modified while 52 requests are tainted because the users replays, votes and comments the modified questions after the intrusion instant. For recovery, the 52 tainted requests shall be replayed. If Shuttle does not have a snapshot containing the questions, then 253 requests have to be replayed to recreate them.

Software vulnerability. On the second class, we evaluate intrusion scenarios where software flaws allow attackers to modify the database without authorization. For instance, a code version added a flaw that allows *SQL-Injection*. We consider two independent scenarios where the attacker: a) deleted every question; b) deleted every answer.

In *2a)*, the deleting of every question removes 4 338 data items. In *2b)*, the questions are preserved but 1 278 answers, votes and comments are tainted as the user did not see the deleted answers.

Instead of identifying the requests that explored the vulnerability, the tenant patches the code to remove the application vulnerability. Tenants use the instance rejuvenation mechanism to shutdown current application containers and deploy new application version. After, they use the *full replay* to repeat all requests since the beginning of usage of the software version with the flaw. Requests that explored the vulnerability fail to execute and a consistent application state is recovered.

External channel. On the third class, we consider a case where the proxy does not log the attacker actions. In this case, an attacker used a SSH account created by exploring the shellsock, which is a bash vulnerability. The attacker stolen the database credentials and modified at least 2000 data items. Since these database operations are not logged, the dependencies are not established and the number of tainted

requests is unknown. However, even without logging the malicious actions, Shuttle recovers the application by loading a database snapshot previous to the estimated intrusion instant (Sep. 2nd) and performing *full replay*. The attack effects are removed because Shuttle loads a database snapshot instead of undoing every operation. As the malicious actions were not logged, they are not replayed and Shuttle recovers the application consistency.

The number of requests to replay is defined by the snapshot instant: on *full replay* Shuttle replays all requests performed after the intrusion instant, while on *selective replay* Shuttle replays the requests necessary to read the values of the entries before the intrusion and the tainted requests. While *selective replay* seems to have a big advantage comparing with *full replay*, which performs, in these scenarios, at least 38 620 requests, most real applications have more dependencies thus the number of tainted requests is bigger. For instance, if the order between questions with the same tag is considered as a dependency, the number of dependencies rises from 92 939 to 109 118 and the number of independent clusters decreases from 6992 to 56. We plan to further analyze the dependencies established by different applications.

C. Performance

We evaluate Shuttle's performance considering the throughput of the application, the size of the logs and the recovery time. We also estimate the cost of deployment of Shuttle on a public cloud provider *Amazon Web Services* (AWS). We run 6 AWS *c3.xlarge* instances (14 ECUs, 4 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 7.5 GB of memory, 2 x 40 GB Storage Capacity) connected by gigabit ethernet (780Mbps measured with *iperf*, 0.176ms round-trip time measured with *ping*). We use one client, one instance with Shuttle proxy and a load balancer (HAProxy), three WildFly (formerly known as JBoss) application servers and one Voldemort database. We consider a large data sample from the data of Stack Exchange with 50 000 requests (1432 questions, 3399 answers, 8335 comments, 36834 votes, 950 000 question views). We do not consider a particular scenario or replay scheme (full/selective), but define instead the number of requests recovered per experiment.

Performance overhead. We evaluate the overhead of Shuttle by measuring the throughput of the *Ask* application with and without Shuttle (Table II). We considered two workloads: (A) has 50% reads, 50% write and (B) has 95% reads, 5% write. Write operations insert questions, answers, comments and votes of the data sample, while the read operations access the latest inserted questions. Table II shows that Shuttle imposes an overhead of 13-20%, which seems reasonable considering the benefits of having it. We believe the main cause of overhead is the current proxy, which is not very optimized. The current version written in Java performs considerably better than a previous version in Python, but we expect to be able to do much better by rewriting it in C.

	Workload A	Workload B
Shuttle	6325 ops/sec [5.78 ms]	15346 ops/sec [3.62 ms]
No Shuttle	7148 ops/sec [5.07 ms]	17821 ops/sec [3.01 ms]
overhead	13% [14%]	16% [20%]

Table II
SHUTTLE OVERHEAD IN TERMS OF APPLICATION THROUGHPUT (OPS/SEC) AND RESPONSE LATENCY (MS)

In order to measure Shuttle's overhead on the database accesses, we used the *Yahoo! Cloud Serving Benchmark* (YCSB) framework [27]. We considered two workloads: (A) has 50% reads, 50% updates and (B) has 95% reads, 5% updates. Operations access 1KB records following a Zipfian distribution (Figure 6). Results show Shuttle has small impact on the latency of database accesses.

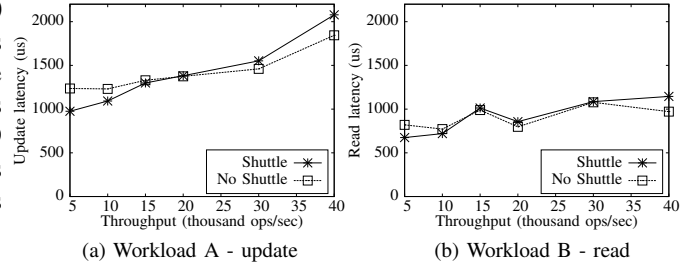(a) Workload A - update   (b) Workload B - read

Figure 6. Performance overhead on database

Recovery. We measured the recovery time using Shuttle to replay the sample of 1 million requests. While serial replay (1 cluster) takes approximately half hour (1717s), recovery with clusters takes only 9 minutes (544s) (Figure 7a).

We measured the recovery period with different number of instances on clustered mode (Figure 7b). The figure shows that Shuttle is scalable, in the sense that adding more servers allow reducing the time of recovery (3 servers allowed recovery in half the time of 1, 750 versus 400 s).

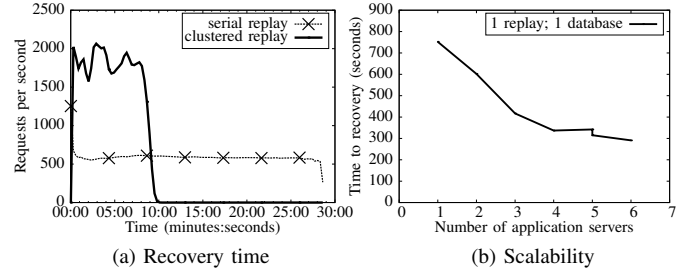
(a) Recovery time   (b) Scalability

Figure 7. Recovery time and scalability

We measured the duration of the restrain period considering two clients with a constant throughput of 400 requests/sec. The serial replay mode, due to implementation issues, is not capable of exhausting the application servers and takes almost one hour to recover (2953s from which 1100s restraining) (Fig. 8a). The clustered mode takes 10 minutes (635s), from which the restrain period represents 46 seconds. (Fig. 8b).

Space overhead. We measured the memory and storage overhead of 1 million requests, from which 95% were read question requests. Values at Table III represent the size of each component in memory . Requests and keys are stored
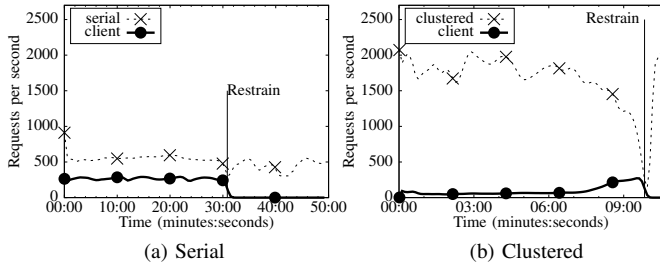
Figure 8. Restrain period in Serial and Parallel Recovery

in the external database while the dependency graph and the accesses are kept in the manager and database instances. No snapshot has been taken and the data is not compressed.

In the current implementation, the SRD represents a fixed overhead of 35 bytes per request.

	# objects	size (MB)
Shuttle Storage:		
Request	1 million	212
Response	1 million	8 967
Start/End	2 million	16
Keys	137 million	488
Total		9 684
Database node:		
Version List	14 593	1.4
Operation list	9 million	277
Total		282
Manager:		
Graph	1 million	718

Table III
STORAGE USED BY SHUTTLE

The main overhead are the responses, as we are storing them complete (the full HTML pages). Notice that Shuttle has to store the responses only if the tenant uses the API to solve inconsistencies (Section V-F). The size of the list of keys accessed by the request depends on the key length and the number of keys accessed. Each access implies an overhead of 13 bytes to record the request ID and the operation type in the version list. The snapshot does not impact the throughput but requires to track the new version, which implies a storage overhead of 10 bytes for each data item when it is written by the first time after a snapshot. The overhead can be reduced implementing the version list as a bitmap. The total database storage overhead encompasses synchronization mechanisms. Since the dependency graph is implemented as a double-linked graph, each entry in the dependency graph has 765 bytes to store not only the start/end instant of the request but also the requests which this request depends from and to (10 on average). Serialization mechanisms and compression techniques can reduce the storage overhead. For instance, the *lz4* of Cassandra, reduces th size of the Shuttle Storage in disk to 4.9 GB.

Monetary cost. Since the replay instances are allocated on demand and paid-per usage, the cost of Shuttle is dominated by the storage. For instance, the proportional overhead of generating 20 million requests per day. To store a quarter (20 billion requests) requires 1.432 TB to store the Shuttle storage, 1.436 TB for the graph and 564 GB in the database

instances. We propose to combine the DynamoDB to keep the last 24h of requests and the Glacier service to archive the data. Shuttle generates an average of 35 GB per day, which costs $8.75 per month to store in DynamoDB and $4.83 per-month for the provisioned capacity. The Glacier stores 3.433 TB so it costs $34.33 per month. Since Shuttle performs snapshots, tenants can remove the old snapshots tacking into account that Shuttle needs only a snapshot previous to the intrusion instant to recover the application.

Shuttle requires an extra instance to deploy the Shuttle manager. To recover the application, we used one *c3.xlarge* virtual machine as replay instance and two *c3.xlarge* instances to run the application servers to replay 1 million requests during 544 seconds. Considering a full-hour, these instances have an associated cost of $0.239 per instance-hour, which means a cost of less than $1 for the recovery. In this manner, Shuttle leverages the elasticity and pay-per-usage model of cloud computing to provide a cost-efficient intrusion recovery solution.

VII. RELATED WORK

Shuttle is an intrusion recovery system based on log-replay. While this approach has been applied in operating systems [8], [9], [10], databases [11], [12], and web services [14], [13], [15], our system is the only one that recovers from intrusions in applications deployed on PaaS platforms. The closest works to Shuttle are Aire [15], Warp [14], Goel [13], and Undo for Operators (UO) [16], although none of them does recovery in cloud environments.

Shuttle's full-replay approach is motivated by UO [16]. UO proposes to remove the intrusion effects using a snapshot and replaying every request posterior to the snapshot instant. UO considers monolithic applications, which are instantiated in the paper as an email server. Shuttle, on the contrary, considers a PaaS platform with both application server and database instances, supporting scalable applications of several kinds . UO sorts requests using knowledge of the application protocol. Developers must define, for each type of application request, the order between requests and their capability to be executed in parallel. In contrast, Shuttle uses the dependency graph to create clusters of request to execute in parallel and sorts the requests using the start-end order and database accesses.

Goel et. all [13] proposes a solution to recover from intrusion in web applications. It uses a modified PHP interpreter to determine the tainted requests. Goel reverts the effect of tainted requests applying compensating transactions on the current state of the database.

Warp [14] helps the administrators to retroactively patch security vulnerabilities. It stores every version of each data item and the version read by each request. It also captures the browser-side input at DOM level using a browser plugin and modifies the code interpreter to track the code files invoked by each request. Requests that invoked code files modified by the patch are considered tainted. Warp loads the version of the tainted data items and repeats these requests using a server-side browser. Forward requests are also replayed while

their inputs are different from the ones at first execution. Most of previous solutions store every data item version or action input and output [14], [15]. Shuttle incurs on smaller storage overhead because it gets the data item version from a snapshot and replays every legitimate request at least until the next snapshot, where the data item value is known.

There have been different approaches to remove the intrusion effects: Goel use transaction compensation to create a snapshot, Warp stores every data item version and UO uses snapshots. While the first does not remove the effects of unknown actions, the second requires a considerable storage overhead. We implemented a snapshot mechanism designed for distributed databases. While UO overwrites the current state copying an old snapshot, we use the *branch path* mechanism to select the accessible snapshots for a given request without copying data.

Warp supports recovery in runtime using two fields in the database table, whereas Aire uses a branching mechanism to permit recovery on loosely coupled web services. We use a branching mechanism to support runtime recovery allowing various recovery processes simultaneously.

Goel does not address external consistency issues. Warp [14] detects inconsistencies in responses and replays the user interaction using a browser. UO uses compensating actions based on protocol-specific knowledge. We propose an API that the application developers can use to deal with inconsistencies.

VIII. CONCLUSION

The paper presented Shuttle, an intrusion recovery service for PaaS. We described the design of a new architecture where a snapshot-based recovery system is provided as a service for PaaS tenants. Shuttle relies on a distributed database and the resource elasticity of PaaS environments to reduce the recovery time and costs. We introduce a novel dependency mechanism based on request start and end instants and list of accesses to order the requests during replay. Shuttle uses a branching mechanism to avoid service downtime during the recovery phase and permits to undo a recovery process. Our evaluation shows that Shuttle can replay 1 million requests in 10 minutes, costing less than $1.

REFERENCES

[1] L. Vaquero and L. Rodero-Merino, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM CCR*, vol. 39, no. 1, 2008.

[2] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM CCR*, vol. 41, no. 1, pp. 45–52, 2011.

[3] N. McAllister, "Code spaces goes titsup forever after attacker nukes its amazon-hosted data," *The Register*, Jun 2014.

[4] J. Williams and D. Wichers, "Owasp top 10 – 2013," *OWASP Foundation, April*, 2013.

[5] H. Sidhpurwala, "Bash specially-crafted environment variables code injection attack," September 2014.

[6] P. Veríssimo, N. Neves, and M. Correia, "Intrusion-tolerant architectures: Concepts and design," *Architecting Dependable Systems*, vol. 11583, pp. 3–36, 2003.

[7] A. Brown and D. Patterson, "To err is human," *Proc. of Workshop on evaluating and architecting system dependability*, 2001.

[8] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *SOSP*. ACM, 2005.

[9] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution." USENIX, 2010.

[10] T. Kim, R. Chandra, and N. Zeldovich, "Recovering from intrusions in distributed systems with dare," in *Proc. of APSys*. ACM, 2012, p. 10.

[11] P. Liu, J. Jing, P. Luenam, and Y. Wang, "The design and implementation of a self-healing database system," *Journal of Intelligent Information Systems*, vol. 23, no. 3, Nov. 2004.

[12] D. Pilania, "Design, implementation, and evaluation of a repairable database management system," in *ICDE*. IEEE, 2005.

[13] I. Akkus and A. Goel, "Data recovery for web applications," in *DSN*. IEEE, Jun. 2010, pp. 81–90.

[14] R. Chandra, T. Kim, and M. Shah, "Intrusion recovery for database-backed web applications," in *SOSP*. ACM, 2011.

[15] R. Chandra, T. Kim, and N. Zeldovich, "Asynchronous intrusion recovery for interconnected web services," in *SOSP*. ACM, 2013.

[16] A. B. Brown and D. A. Patterson, "Undo for operators : Building an undoable e-mail store," in *USENIX ATC*, 2003.

[17] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE TDSC*, vol. 1, no. 1, pp. 11–33, 2004.

[18] I. Akkus and A. Goel, "Data recovery for web applications," in *DSN*. IEEE, Jun. 2010, pp. 81–90.

[19] G. DeCandia, D. Hastorun, and M. Jampani, "Dynamo: Amazon's highly available key-value store," in *SIGOPS*, vol. 41. ACM, 2007, pp. 205–220.

[20] J. Lin and Mario A. Nascimento, "A survey of distributed database checkpointing," *Distributed and Parallel Databases*, vol. 319, pp. 289–319, 1997.

[21] M. Xie, H. Zhu, Y. Feng, and G. Hu, "Tracking and repairing damaged databases using before image table," in *FCST*, 2008.

[22] T. Wood and E. Cecchet, "Disaster Recovery as a Cloud Service," *2nd USENIX HotCloud Workshop*, 2010.

[23] P. Sousa *et al.*, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE TPDS*, vol. 21, no. 4, 2010.

[24] S. Chacon, *Pro Git*. Apress, 2009.

[25] N. Chohan *et al.*, "Appscale," in *Cloud Computing*. Springer, 2010.

[26] R. Sumbaly *et al.*, "Serving large-scale batch computed data with project voldemort," in *FAST, USENIX*, 2012.

[27] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *1st SoCC*, 2010, pp. 143–154.