Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

μVerum: Intrusion Recovery for Microservice Applications

DAVID R. MATOS (Member, IEEE), MIGUEL L. PARDAL (Member, IEEE), ANTÓNIO RITO SILVA (Member, IEEE), and MIGUEL CORREIA (Senior Member, IEEE)
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Lisbon, Portugal

Corresponding author: David R. Matos (e-mail: david.r.matos@tecnico.ulisboa.pt).

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO)

ABSTRACT Microservice architectures allow complex applications to be developed as a collection of loosely coupled components. The heterogeneous architecture of these applications makes the process of recovering from intrusions especially complex, error-prone, and time-consuming. Although there are several recovery mechanisms for monolithic applications, applying such mechanisms in microservices would not work due to the distribution of the components, the different technologies used by each service, and their scale. Moreover, it can be difficult to trace the services affected by an intrusion and which actions to revert. We propose μVerum, a framework for recovering microservices from intrusions that corrupt the application state. Our approach allows recovery of large-scale microservice applications by logging user requests and the operations that are propagated through several microservices. When a system administrator detects a faulty request, μVerum can execute compensating operations in each of the affected microservices. We implemented, evaluated, and made the code of μVerum available. Our experiments show that μVerum is able to revert the effects in an intrusion in one second while the application is running.

INDEX TERMS Microservices, Cloud Computing, Intrusion Recovery

I. INTRODUCTION

The development of complex web applications has shifted from the traditional monolithic architecture to a distributed architecture that allows developers to organize in small teams, each one responsible for one self-contained component (or small set of components) of the application, and/or simply to reuse existing components. Each component is called a *microservice* [1]–[5] and communicates with other microservices through network interfaces [6].

Microservice applications rely on network communication and, as such, require isolation mechanisms, such as firewalls. Despite the best protection efforts, malicious users may still be able to bypass security, for example, by exploiting a misconfiguration or a previously unknown bug [7], leading to an *intrusion*. When such an attack occurs, it can *corrupt the state of microservices*, i.e., modify data for the benefit of the attacker. Moreover, its effects will tend to propagate to other microservices. Using the built-in rollback mechanism of database management systems is not an option because these mechanisms aim to revert a database transaction that did not complete successfuly. This is not the case of an intrusion in a microservices application in which the malicious operation is an HTTP request that was completely executed, meaning that any database transaction associated with it was also completely executed and cannot be rollbacked. Another approach would be to revert the state of the application to a previous point in time prior to the attack (a checkpoint) and continue execution from that point forward [8] . However, this approach has some drawbacks: it requires consecutive and consistent checkpoints of the state and synchronization across multiple machines, which may be unfeasible in microservices; the application must be offline during recovery; and any legitimate operation after the attack is lost.

There is previous work on intrusion recovery for storage and databases [9]–[15], for monolithic web applications [16]–[18] and for other applications [10], [11], [19], [20]. Each of these intrusion recovery systems was designed for recovering intrusions on a specific kind of system (databases, web applications, etc.). *It is not possible to use these systems with a microservice application for these reasons:*

- Each microservice may be written in a different programming language and may use a different kind of database (SQL, NoSQL). This heterogeneity may require a different recovery scheme for each microservice;

- As opposed to a monolithic application, in which a user interaction generates a single HTTP request[1], in a microservice application, a user interaction generates multiple HTTP requests across several microservices. This correlation of HTTP requests and database statements is necessary in order to coordinate recovery;
- Some of the existing works require the application to be temporarily unavailable while recovery is carried out [10], [11], [18], [19], [21]–[23]. In a microservice application composed of hundreds of microservices, having to shut down the entire application is complex, time-consuming, and can result in significant monetary costs for the organization.

We present μVerum, a novel framework for allowing microservice applications to *recover from intrusions* without being shut down during recovery. μVerum recovers the affected services by executing *compensating operations* [24] that undo the effects of intrusions while preserving the valid data recorded in the system. μVerum assumes a system architecture with the components that can be found in typical microservice applications, e.g., routers and a discovery service [25]. Developers can setup μVerum progressively in a subset of microservices and gradually extend it to the entire application.

The malicious operations to revert can be identified by the administrator of the application using the μVerum search engine or an intrusion detection system (IDS) [26]–[32]. It is possible to combine μVerum with an IDS to reduce recovery time; however, we do not discuss intrusion *detection* in detail, as it has been widely studied for decades and is mostly orthogonal to intrusion recovery. We also do not consider the general problem of *protecting* microservice applications of intrusions, something that has been studied before [33], [34] and that is also orthogonal to intrusion recovery.

We evaluated μVerum by performing experiments with two open-source microservice applications: SockShop [35] and Piggy Metrics [36]. Our experiments show that μVerum affects the performance of the application by less than 14% with asynchronous logging, which can be further reduced by scaling the agents of the most accessed microservices. μVerum has shown to be capable of recovering faulty requests while maintaining the availability of the application.

This paper provides the following contributions: a novel framework for the development of microservice applications that allows practical intrusion recovery with consistency guarantees; a prototype of the recovery system μVerum [2]; and an experimental evaluation with real-world applications.

The paper is structured as follows: Section II explains the microservices approach, Section III discusses the problem of the dependency graph, Section IV presents the μVerum approach, Section V describes the experiments we performed, Section VI compares μVerum with the state-of-the-art, finally, Section VII concludes the paper.

II. MICROSERVICE APPLICATIONS

The microservice architecture allows software systems to be developed as a set of small, independent, and self-contained services. Each service can be deployed in a specific execution environment on a different physical infrastructure and developed in its own programming language. Microservices communicate using a RESTful (Representational State Transfer) [6] or RPC-based API [1]. In this way, services can be developed by independent teams that only share APIs among them. This gives many benefits to the implementation of complex business applications, as developers can adopt a divide-and-conquer approach and add new features without the need of redeploying the entire application.

A. MICROSERVICE API

A microservice API is provided by a server to clients or consumers. It is assumed that the server and clients are distributed and that the API is invoked through the network. There are different types of interfaces; some rely on a formal definition of the available functions enforcing the consumer of the service to use the same technology as the server (tight coupling), whereas others are more open, allowing different technologies to be used (loose coupling). Some examples of technologies used by microservices are Java RMI [37], .NET Remoting [38], SOAP [39], REST [6] and gRPC [40]. An API of a service can be *synchronous* if the consumer of the service blocks until a response is returned, or *asynchronous* if the consumer proceeds with computation without waiting for a response from the provider.

B. ARCHITECTURE

The architecture of a microservices application can differ. Some applications use routers to guide traffic to the corresponding services [25], [36], others use a central message bus where requests are published by clients and then consumed by the corresponding servers [41], and some use a hybrid approach [35]. To deal with these alternatives, in most of the paper, we consider what we believe is the most commonly adopted architecture – orchestration – and in Section IV-H we discuss the changes necessary for the main alternative – choreography.

The microservices *orchestration* architecture we consider is inspired by Netflix's architecture [25], [42] (Figure 1). We have chosen it given the contribution it has had in the development of systems to support microservice applications, such as Zuul [43], Eureka [44] and Ribbon [25].

Next, we present the main components of the architecture: HTTP server, router, discovery service, and microservices.

The *HTTP server* is the only component of the application that must be publicly available to users. Typically, it is deployed in the DMZ (demilitarized zone, the perimeter network that is located between an organization's internal network and the external network) of the network and serves different types

[1] Although nowadays it is more prevalent to use the HTTPS protocol, in this paper, for simplicity, we refer to the requests of the HTTP and HTTPS protocols as HTTP requests.

[2] Source code available at: https://github.com/davidmatos/uVerum

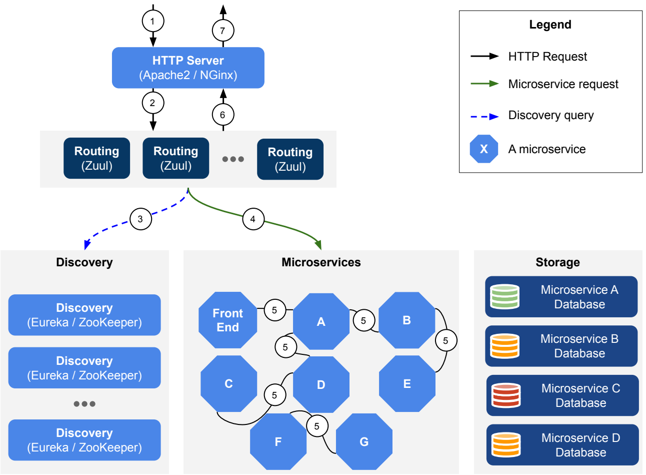


FIGURE 1: Common architecture of a microservices application and flow of processing a request.

of users of the application (web browser, mobile devices, and other applications). This server is stateless, allowing it to be replicated, while having a load balancer coordinating the distribution of requests. It is also possible to deploy cache systems that optimize traffic alongside HTTP servers. For simplicity, we assume that there is only one HTTP server that serves clients and interacts with the APIs provided by the microservices.

The *router* forwards requests to the corresponding services, translating HTTP requests to the server into service requests. The router can be replicated to cope with traffic. Routers are useful for modifying or verifying requests on the fly using filters. This allows developers to incorporate metadata in requests or to perform integrity checks and encrypt data.

The *discovery service* is responsible for keeping a record of the microservices addresses in the network. They translate a service name into an address when the router does not know it. They may also provide fault detection alerts so that the router knows that the service is not available. The use of discovery services is encouraged [1], [45].

The *microservices* themselves are self-contained. They provide APIs for consumers and communicate over the network. The consumers of the microservices can either be the routers that are forwarding requests from the HTTP server or other microservices. We will assume that services are reachable by any service.

C. FLOW OF A REQUEST

A user request passes through the different components of the application before a response is returned to the user. In Figure 1, when a request reaches the application, it is handled by an HTTP server with a public IP. This server will redirect the request to a router, which in turn will generate a service request. It may first consult with the discovery service to translate the address of the service to an IP address. Once a microservice receives a request, it can then generate other requests.

More specifically, the flow goes this way:

(1) a user issues a request using a web or mobile application and that request reaches the HTTP server; (2) if the HTTP request corresponds to a service, then the server will forward it to the router; (3) (optional) if the router does not know the IP address of the corresponding microservice, it will first query the discovery service, otherwise it will contact the microservice directly; (4) the request reaches a front-end microservice responsible for dealing with application level requests; (5) the front-end microservice may contact other back-end microservice or return a response to the router; (6) the router forwards this response to the HTTP server; (7) the HTTP server sends an HTTP response to the client that issued the initial request.

III. THE DEPENDENCY GRAPH

The dependency graph is a tree graph, i.e., an undirected graph in which any two nodes are connected by a single edge, that describes the execution of a user's request in the microservice application. This graph is similar to a stacktrace [46] or stack-traceback [47] which are tools that detail the stack frames that are active during the execution of a program. These tools are commonly used to debug faulty code, since they allow developers to navigate through the execution flow of erroneous code and to reason about what caused the program to crash.

μVerum traces the user's request from the moment it reaches the application (from the HTTP servers in the form of an HTTP request) through every microservice it invokes until it leaves the application (in the HTTP servers in the form of an HTTP response).

The elements of the dependency graph are:

- root: the user's request, i.e., the HTTP request that comes from outside of the application and reaches the HTTP servers;
- nodes: microservice operations that are directly or indirectly executed by the user's request. The same microservice operation can be present in several nodes, e.g., if a user's request aims to delete two files the same *deleteFile* operation can be executed twice, this will result in two nodes with the *deleteOperation*;
- edges: describe the invocation of an operation. For example, the edge (x, y) means that the microservice operations x executed the microservice operation y.

A. BUILDING THE DEPENDENCY GRAPH

To build the dependency graph, it is necessary to collect information about the user's requests and the microservice operations they executed. For each executed operation, μVerum collects the microservice operation that executed it, creating a pair (operation x, operation y). The values x and y are unique IDs that unmistakably identify every executed operation in the application. These pairs are stored as is, in a log (μVerum log), since there is no need to calculate a graph for every user's request since the graph is only used for recovery.

μVerum generates the dependency graph for a request when the administrator selects that request to be reverted, e.g., because the request was malicious. μVerum does this by selecting the malicious HTTP request in the log, the only pair that has the first member empty and the second one has the ID of the malicious HTTP request (null, malicious HTTP request). Then it collects every pair in the log with the same request ID in the first member, in other words, every request that was issued by the malicious HTTP request (every pair (malicious HTTP request, *)). Then, for every collected pair, this process is repeated recursively until there are no more microservice operations that match (operation, *).

B. RECOVERY WITH THE DEPENDENCY GRAPH

The dependency graph guides the recovery process. When the administrator selects an HTTP request to recover from, μVerum calculates the dependency graph of that request and, starting from the root of the graph, it executes the compensating operations in each affected microservice. By following the dependency graph, it is ensured that every affected microservice is recovered from the undesired HTTP request.

The recovery process takes time and is done while the application is available to its users, so users may experience some inconsistencies. Furthermore, while recovery is being performed, it is not desirable to allow users to modify or access the data that is being recovered. For example, if an attacker manages to modify the price of an item in an e-commerce application, we do not want any user to be able to purchase that item at the wrong price. To avoid this, μVerum allows developers to limit access to certain operations while the application is recovered. This is done in the form of invariants, and their usage is explained in detail in Section IV-G

IV. THE μVerum APPROACH

The μVerum approach requires the coordination of several components and their integration with the application code, following a set of guidelines. As long as the application follows the μVerum guidelines then it will be possible to trace the effects of intrusions and later recover from their effects.

A. SYSTEM MODEL

μVerum recovers from intrusions that affect applications composed of a set of microservices (Figure 1). These applications are available to *users* with limited privileges, and maintained by *system administrators* with higher privileges. Users interact with web servers that redirect their requests to a subset of microservices. User requests are encoded as HTTP requests, which in turn generate microservice requests. More rigorously, we make the following assumptions:

- *S1:* the API is RESTful;
- *S2:* user requests reach the application's microservices from web servers;
- *S3:* users cannot access the microservices directly, as they are not publicly available;
- *S4:* only microservice requests intercepted by μVerum agents are recovered;
- *S5:* the HTTP PATCH method [48] is available to use for the recovery process of μVerum (this method is rarely used and its use allows a normalized way to fix a previous request);
- *S6:* the microservices can be written in different programming languages and use different types of data repositories.

B. THREAT MODEL

We define an *intrusion* as a malicious request that leads an application to a faulty state. When this happens, there are two challenges in terms of recovery: first, it is necessary to detect the trail of the faulty operation, that is, the subset of microservices that were affected by the malicious request and need to be corrected; and second, it is necessary to define which compensating actions should be executed to lead the system back to the valid state. At the end of the recovery, the state of the application should be the same as if the malicious request had never occurred.

Intrusions can occur when a malicious user exploits a vulnerability in the application front-end (top of Figure 1), e.g., a SQL injection vulnerability [49] or an authentication flaw [7], [50]. Accidental operations that corrupt the state of the application are also taken under the term "intrusion", as their effects can also be fixed using μVerum. More specifically, an intrusion is the effect of an HTTP request that reaches the application which, in turn, generates several microservice operations causing unwanted changes to the state of the application. We assume that there is no other way for the attacker to cause an intrusion.

The threat model is as follows:

- *T1:* intrusions come from outside the application network and enter the application through the HTTP servers;
- *T2:* intrusions cause state modifications to the microservices' data stores;
- *T3:* neither the application nor μVerum's microservices are compromised or disabled, that is, they are part of the Trusted Computing Base (TCB) [51].

With *T1* we assume that an attacker does not have access to the network in which the microservices are running. This is a reasonable assumption given that most attacks against web applications come through the front-end [7], [50], [52]. In relation to *T2*, we focus only on recovering from intrusions that illegally modified the state of the application. Intrusions that do not modify the state of the application are not covered in this work, as there is nothing to recover from them. *T3* clarifies that our problem is the recovery of the state of the application, not protecting services from attacks that modify their code or configuration.

C. COMPENSATING OPERATIONS

Compensating operations are used to revert the intermediate state of an incomplete/failed transaction. This approach was arguably introduced with the Sagas pattern [53]. A saga is a set of operations that can be interleaved with other operations.

Each operation should be reversible by a compensating action. This ensures that either every operation in the saga is executed or that the compensating actions are performed to revert the incomplete process. This pattern was recently adapted for microservice applications [54]. μVerum implements this pattern to perform recovery. To do so, μVerum requires that every operation that should be reversible has a compensating operation that reverts its state. The compensating operations can be implemented by the developers of the application, since they know what actions need to be performed to recover a microservices operations. This gives the benefit of implementing a more sophisticated compensating operation that, for example, besides reverting the state from the attacker's actions, it also notifies the user that the state of the application was reverted intentionally. It also reduces the overall implementation overhead because compensating operations are also used in the implementation of microservices such as Sagas.

D. SYSTEM ARCHITECTURE

In this section, we describe the architecture of a microservices application extended with the μVerum components (Figure 2). In the figure, some microservices (those in red/darker) are wrapped by a μVerum agent that intercepts the requests so they can be logged. μVerum requires two databases to keep, respectively, operation logs (μVerum Log Database) and configuration values (μVerum Config Data).

1) μVerum admin

Admin is a microservice that runs alongside the other microservices of the application and controls the recovery process ("μ", gray in the figure). It is the only component of μVerum that is accessed by the administrator. When it is necessary to recover microservices, this component fetches the logs and contacts the agents to execute the recovery operations. It is also through μVerum admin that the administrator configures the agents.

2) μVerum routers

The routers (dark blue in the figure) add metadata to every request to allow *correlation* and *ordering* of requests. These metadata consist of a unique serial ID. This *request tainting* technique of adding metadata to trace the data flow has been explored in previous works [55]–[57]. With this serial ID, it is possible to order the HTTP requests that reach the application (from the HTTP servers) and retrieve every microservice operation that was executed as a result of that HTTP request.

3) μVerum log

The log is a distributed database in which user requests and microservice operations are logged. This approach of collecting every log entry of several microservices in the same log database is a recommended practice for microservice applications, as it allows the administrator to view and analyze the application history as a whole rather than as a collection of parts [1], [58]. Another advantage of using a single log for every microservice is that it facilitates the task of correlating every operation that was issued by a user's request.

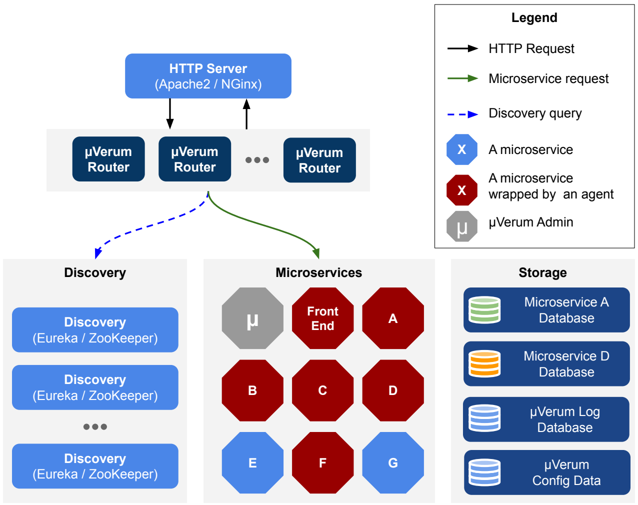


FIGURE 2: System architecture of an application adopting the μVerum approach.

4) μVerum agents

Agents are proxies that intercept requests that reach the microservices. The agents log every HTTP request that reaches a microservice, regardless of whether the service is stateless or not. This information allows μVerum to generate the dependency graph. This process does not interfere with the microservice operation, since the agent only intercepts the requests to log them and does not perform any modifications to them. The agents can be configured to work synchronously, by forwarding requests only after they are logged, or asynchronously, by forwarding the requests immediately and logging them in background.

In addition to requests that reach microservices, μVerum agents also collect the corresponding status codes and response timestamps. Status codes (e.g., `404 Not Found`) are used to discard failed operations from the log. Timestamps will be used to order and correlate the execution of requests. During recovery, μVerum executes concurrent requests in parallel to reduce the overall recovery time. Concurrent requests occur when their execution overlaps. More formally, two operations o_1 and o_2 with timestamps $start\_ts_{o_1}$, $end\_ts_{o_1}$, $start\_ts_{o_2}$ and $end\_ts_{o_2}$ are concurrent if: $start\_ts_{o_2} \leq end\_ts_{o_1} \wedge start\_ts_{o_1} < end\_ts_{o_2}$.

E. SETTING UP A μVerum APPLICATION

To use μVerum in a microservice application, developers have to follow a set of guidelines.

First, microservices have to interact with each other exclusively through HTTP using REST APIs. Operations that use different protocols may exist, but are not traced by μVerum and, as a result, cannot be undone.

Second, the developers are responsible for implementing the PATCH method for each operation that may have to be undone. This is an important aspect of the μVerum approach. Given the heterogeneous architecture of microservice applications, only developers can be fully aware of what needs to be done locally in the microservice to undo an operation and they should leverage existing local recovery mechanisms.

Third, external operations must be handled differently. Examples include cloud storage services, social networks, banking, serverless computing services, and accountability systems. These operations may require a specific recovery approach that uses compensating operations made available by the external service provider.

Fourth, the application shall have an architecture similar to the one presented in Figure 1, that is, there are routing and discovery servers that are used to redirect operations through the μVerum agents and append the required metadata (unique IDs) to the executed operations.

Fifth, if some operations need to be executed atomically or in a specific order then it is the responsibility of the developer to return that information when the PATCH method of an operation is executed. In this way μVerum knows how to process these operations without violating the consistency requirements of the application.

F. INTRUSION RECOVERY WITH μVerum

μVerum logs user requests and microservice operations during normal execution. This information is later used to perform recovery. In this section, we describe how μVerum logs requests and operations and how the recovery process works.

1) Normal execution

During normal execution, μVerum routers and μVerum agents log user requests and microservice operations. Requests are intercepted and logged without interfering with the application. For each logged operation, μVerum records the following information:

- `request_id`: a unique serial ID assigned when the request reaches the application;
- `start_ts`: a timestamp of the moment the operation was logged;
- `end_ts`: a timestamp of the moment the response of the operation was logged;
- `service`: the address (IP or hostname) of the service;
- `sender`: the address of the issuer of the request
- `method`: the HTTP method of the request;
- `operation`: the payload of the HTTP request.

2) Recovery

Recovery is done when the administrator of the application detects faulty requests and wants to undo their effects on the affected microservices. When the administrator selects faulty requests from the log, μVerum presents him with a list of operations that were executed by the microservices and will be reverted. This will allow the administrator to preview what will be recovered. Once the administrator confirms the recovery, μVerum executes compensating operations, one per operation, that will revert the effects of the attack. The compensating operations are invoked by μVerum using the PATCH verb. All microservices should be running when the administrator issues the recovery operation, but that may not be the case, as a set of the microservices that need to be recovered may be offline. In this case, μVerum schedules the recovery actions to be executed as soon as possible.

Algorithm 1 describes the procedure to identify and undo microservice operations given a faulty HTTP request. It takes as input req (line 1), the log entry with the malicious HTTP request that the administrator identified. First, it initializes an empty list to store the undo operations that failed (line 2). This will happen if some microservices happen to be offline during the recovery process and μVerum needs to postpone the undo operation. Then it collects every microservice operation that was caused by req (line 4). With these operations, it is possible to build the *dependency graph* (line 5) that illustrates how a user request relates to the executed operations inside the microservice application. This graph is created using the `service` and `sender` fields of each log entry. Every two operations that have a `service` equal to a `sender` are connected in the graph. Then this graph is presented to the administrator (line 6) for confirmation (line 7) to undo every operation in the graph. If confirmed, the recovery process starts at the root node of the graph (lines 8 and 9).

The undo function is recursive (lines 11 to 31). It takes as input a node from the graph.Then, it gets the service address from the log entry (line 12) and probes for invariants (line 13). Invariants can be of two forms: *ORDER* (lines 14 to 16) or *ATOMIC* (lines 17 to 19). These special cases will be treated by the appropriate functions presented in algorithms 2 and 3. If there are no invariants, the algorithm proceeds to execute the PATCH function of this service (line 21). This PATCH function was previously implemented by the developers of the application, and it will undo the effects of the execution of the operation from the state of the microservice. If it failed to execute the PATCH method then it will be stored in the pending list to be re-executed later. Then, the algorithm fetches the next nodes in the graph, *children*, and for each one of them it recursively invokes the *undo* function (lines 27 to 29). Finally, the list of services that failed to be executed is presented to the administrator (line 30) and the function terminates. This pending list will be used by a background task that runs periodically to complete the recovery operation. In fact, the first time μVerum connects to a service, it will execute all pending operations before forwarding any new requests.

G. RECOVERY CONSISTENCY

To avoid creating an inconsistent state after recovery, the developer may force μVerum to respect the order of operations. This enforcement is defined by the developers in the PATCH methods by invariants, i.e., conditions that μVerum ensure that are not broken during recovery. The two kinds of invariants that will be respected by μVerum during recovery are ordering and atomicity, defined as follows:

Algorithm 1 Recovery without order or atomicity requirements.

```
INPUT $req$ // malicious HTTP request
$pending \leftarrow \perp$
$request\_id \leftarrow req.request\_id$
$log\_entries \leftarrow get\_log\_entries(request\_id)$
$graph \leftarrow trace\_graph(log\_entries)$
$print(graph)$
if $admin\_confirms\_recovery()$ then
    $root \leftarrow graph.get\_root()$
    $UNDO(root)$
end if

function UNDO(node)
    $service \leftarrow node.service$
    $(invariant, nodes) \leftarrow invariant(service.operation, PATCH)$
    if $invariant == ORDER$ then
        $UNDO\_ORDERED(0, nodes, PREPARE)$
    end if
    if $invariant == ATOMIC$ then
        $UNDO\_ATOMIC(nodes)$
    end if
    if $invariant == NULL$ then
        $result \leftarrow execute(service, service.operation, PATCH)$
        if $result.STATUS \neq SUCCESS$ then
            $pending \leftarrow pending \cup node$
        end if
    end if
    $children \leftarrow node.children$
    for $child \in children$ do
        $UNDO(child)$
    end for
    $print(pending)$
end function
```

- *Ordering invariant*: any operation o that takes as a predecessor another operation o' is always executed after o' was completely and successfully executed.
- *Atomicity invariant 1*: if an atomic operation o in a set of operations S_a was completed and executed successfully, then any other operation in S_a was also completely and successfully executed.
- *Atomicity invariant 2*: if an atomic operation o in a set of operations S_a fails to be executed, then every other operation in S_a is not executed or rolled back.

1) Ordering invariants

Ordering invariants are applied to operations that require a specific order to be executed. For example, an operation called *transfer*, which debits a user account balance and credits another user account, should execute these two steps in this exact order: first the debit, then the credit. In this way, if the account being debited does not have enough funds, then the transfer should be canceled. Switching between these two steps can result in money being "created" for a moment. The PATCH method of such an operation should return such ordering requirements. This way, when μVerum is recovering the PATCH of *transfer* it knows that it should invoke *debit* before *credit*.

Ordering invariants are expressed as an ordered list, $ORDER$, of microservice operations. This list will be used by μVerum during a pre-recovery process, during which μVerum probes every microservice operation for any invariants. Once μVerum consulted every operation in the graph, it will order the operations that need to be executed based on the $ORDER$ lists it collected. Following the previous example, the PATCH method of the *transfer* operation should return $ORDER \leftarrow \{debit, credit\}$. With this list, μVerum will investigate the PATCH methods of the *debit* and *credit* operations before starting recovery. If either of these two operations have other ordering dependencies, then μVerum will take such dependencies and join them in the list of operations to be executed in recovery. After examining every operation, μVerum can finally proceed to the execution of compensating operations. The invariant for this example would be expressed as follows:

```
order:{ debit(amount),credit(amount) }
```

Algorithm 2 Recovery with ordering requirements.

```
1:  function UNDO_ORDERED(index, queue, phase)
2:      $service \leftarrow queue[index].service$

3:      if $phase == PREPARE$ then
4:          $result \leftarrow execute(service, PATCH, PREPARE)$
5:          if $result.STATUS \neq SUCCESS$ then
6:              $abort()$
7:          end if
8:          $children \leftarrow queue[index].children$
9:          for $child \in children$ do
10:             $index \leftarrow child.index$
11:             $queue \leftarrow UNDO\_ORDERED(index, queue, PREPARE)$

12:         end for
13:     end if

14:     if $phase == COMMIT$ then
15:         $block\_services(queue)$
16:         for $node \in queue$ do
17:             $service \leftarrow node.service$
18:             $result \leftarrow execute(service.operation, PATCH)$
19:             if $result.STATUS \neq SUCCESS$ then
20:                 $rollback(queue)$
21:                 $abort()$
22:             end if
23:         end for
24:         $resume\_services(queue)$
25:     end if
26: end function
```

Algorithm 2 describes the *undo_ordered* function and how it performs recovery, preserving the execution order of the requests. This function takes as input: an index value pointing to the current node in the list, *index*, an ordered list with the services that need to be recovered by the correct order, *ordered_queue*, and a flag indicating in which phase the algorithm is, *phase*. First, the algorithm extracts from the list the current service that should be executed (line 2). Then, the algorithm follows one of two branches. In the *prepare* branch (lines 3 to 13), μVerum will execute the services in the queue until it reaches the end of the graph. Each service will be executed (line 4) with the parameter *phase* set to *PREPARE*. This is just to fill the queue with every invariant from the services. If any of the services fail to execute in the prepare phase (lines 5 to 7) then the recovery is aborted. This happens because if μVerum cannot determine the order in which the services will be executed, then it cannot guarantee that the invariants are not violated. If the operation preparation is successful, then it will fetch the next nodes from the graph (line 8). For each of the next nodes in the graph, the function *undo_ordered* will be invoked recursively (lines 9 to 12) until it reaches the end of the graph, then it will perform the recursive invocation with the variable *phase* set to *COMMIT*.

In *commit* (lines 14 to 25), the services are executed in the given order. This is done by isolating the services from the user's request (line 15). This does not mean that the requests are discarded, instead they are stored in a list to be executed after the recovery finishes. The algorithm proceeds by iterating through the queue (line 16) and each of the services is executed synchronously (lines 17 and 18), that is, each service starts execution after the previous one has ended. If any execution fails, then a rollback is issued (line 20) and the recovery process is aborted (line 21). At the end of the function (line 24) the blocked services are resumed.

Algorithm 2 follows essentially a 2-phase commit pattern. This algorithm might be trivially modified to follow a 3-phase commit pattern to improve its resilience to certain (unlikely) fault scenarios at the cost of higher time complexity in normal execution.

2) Atomicity invariants

When a developer enforces the atomicity of an operation, it specifies a list of operations that have atomicity requirements, i.e., either all of them are executed or none of them is. During recovery μVerum ensures that every service in the atomicity list executed the recovery operation. If any of the operations fails to run, then recovery must be aborted. For example, an operation that updates the access key of a user requires two operations to be executed: a revocation of the current access key and the generation of the new one. In this example, there is no intermediate state in which the application generated a new key without revoking the previous one. The invariant of this example would be given by the microservice that triggers both operations, and it would be described as follows:

```
atomic:{ revoke_key(user),grant_new_key(user) }
```

Algorithm 3 describes how μVerum recovers services that have an atomic invariant. Function *undo_atomic* (lines 1 to 8) iterates through the bag with the services. After blocking them from user requests (line 2), it will asynchronously execute the PATCH method of each service (line 6). When a service finishes executing the PATCH method (lines 9 to 17), it is removed from the bag (line 10). When the bag is empty (line 11), μVerum resumes the services that were blocked at the beginning of recovery (line 12). When a service fails to execute the PATCH method (line 14), a rollback is executed to revert the partially recovered services (line 15).

H. μVerum FOR THE CHOREOGRAPHY PATTERN

Now we discuss the changes for applications that use the *choreography* pattern, in which microservices interact with each other through a message bus that allows publishing and subscription to operations.

In relation to logging, the μVerum agent provides the means of intercepting and logging operations. For the alternative architecture, this agent can be coupled with the message bus in such a way that any operation that is invoked is logged.

Algorithm 3 Recovery with atomic requirements.

```
1:  function UNDO_ATOMIC(atomic_bag)
2:      $block\_services(atomic\_bag)$
3:      for $node \in atomic\_bag$ do
4:          $service \leftarrow node.service$
5:          $callback \leftarrow UPON\_SERVICE\_RESULT$
6:          $execute\_async(service.operation, PATCH, callback)$
7:      end for
8:  end function

9:  function UPON_SERVICE_RESULT(service, result)
10:     $atomic\_bag \leftarrow atomic\_bag \sim service$
11:     if $atomic\_bag == \perp$ then
12:         $resume\_services()$
13:     end if
14:     if $result.STATUS \neq SUCCESS$ then
15:         $rollback()$
16:     end if
17: end function
```

For recovery, μVerum performs compensating operations that undo the effects of the intrusion. For the alternative architecture, the compensating operations are published in the message bus. The main challenge with this approach is to ensure consistency for operations that have ordering or atomicity requirements. μVerum admin cannot directly coordinate the recovery process, instead it needs the message bus to act as an intermediate in the process. This can be accomplished using message topics that are specific to recovery and are subscribed to by all services. More specifically, the algorithm works as follows:

1) μVerum admin publishes a message with a topic name composed of the name of each of the affected services concatenated with the tag *RECOVERY*;
2) each affected service reads the *RECOVERY* message and publishes another message that has one of the following types: (a) *RECOVERED*, meaning that the patch was successfully executed and recovery is completed; or (b) *ORDERING* or *ATOMICITY*, containing the list of operations that need to be executed;
3) for each of the services in the list, μVerum admin publishes a message with a topic equal to the name of the patch method's name. After a microservice executes the patch method, it publishes *RECOVERED* so that the μVerum admin can proceed to the next microservice.

For this to work, each microservice must be implemented so that it prioritizes *RECOVERY* operations above any other operation. Every microservice is also required to publish *RECOVERED* messages once recovery is complete. In this way, μVerum is able to coordinate the recovery process through the message bus.

V. EXPERIMENTAL EVALUATION

With our experiments, we want to answer these questions: (A) Is the μVerum approach capable of recovering from intrusions in real-world applications? (B) What is the cost, in terms of performance, of using the μVerum agents to log every operation? (C) How long does it take to assess damage and undo unintended actions?

We performed the experiments using Google Compute Engine [59], which allowed us to deploy each microservice on a single and isolated virtual machine. This way we are able to recreate an environment similar to a real-world deployment.

We choose the *n1-standard-2* flavor for every virtual machine, which provides 2 CPU cores with 7.5GB of memory. Table 1 summarises the setup used in the different experiments.

TABLE 1: Experimental Setup

Experiment	**Application**	**Workload (requests)**	**Generated by**	**Network Setup**
Validity	SockShop	10,000	SockShop	15 * (2 vCPU / 7.5GB RAM)
Overhead	SockShop	10,000	SockShop	15 * (2 vCPU / 7.5GB RAM)
Overhead	PiggyMetrics	10,000	JMeter	12 * (2 vCPU / 7.5GB RAM)
MTTR	SockShop	10,000	SockShop	15 * (2 vCPU / 7.5GB RAM)

A. μVerum IMPLEMENTATION

μVerum was implemented in Java, since most of the components were also written in Java or provide a Java API. μVerum admin is a Spring Boot [60] microservice. The μVerum agent is a Java program that uses *Little Shoot Proxy* [61] to intercept requests. The log is a MongoDB [62] database. The router is a Zuul [43] instance with a special filter that appends the metadata to the requests. The discovery service is a non-modified Zookeeper [63] instance. We chose Zuul and Zookeeper because they are widely used in the industry, especially for microservice applications.

We evaluated μVerum experimentally with two microservice applications: SockShop [35] and PiggyMetrics [36]. We chose these applications for our use case for the following reasons: they are open-source, so they can be used by anyone who wants to extend μVerum; they follow a system architecture similar to the one presented in Section IV-D; they differ in terms of architecture, PiggyMetrics uses an orchestration approach, while SockShop has an hybrid architecture with some components working in orchestration and other components working in choreography. These characteristics allow us to evaluate how μVerum works in different types of applications. Both applications have a significant size: SockShop is made up of 9 microservices and 6 datastores, while PiggyMetrics has 6 microservices with 4 databases. SockShop was developed using Java, Go, and NodeJS. PiggyMetrics was developed with SpringBoot.

In our implementation, the compensating operations ranged from 6 to 30 lines of code. We consider this length of code to be short enough for the developers of the application to write it in a single cycle (sprint) of software development.

B. VALIDITY OF μVerum RECOVERY

μVerum successfully recovers an application if it manages to undo malicious operations from the state of the microservice database as if they had never occurred. To evaluate this aspect, we compare a recovered database with one that was never attacked. Specifically, we wrote a script that allows us to execute a *diff*-like operation between two databases: one that has the state of the application after it was recovered by μVerum (database C) and another (database A) that has the state after the application received the exact same operations except the ones that are malicious (thus, that were undone in the case of database C). The script compares all the deterministic values of each database, meaning that non-deterministic values generated by the database itself, such as automatic identifiers and timestamps, are not compared. We are not interested in comparing non-deterministic values since they are outside of the control of the application, which cannot be logged and recovered by μVerum, which only logs application-level requests.

To create both databases, we executed a workload (*workload A*) with 10,000 requests using the SockShop load test in *database A*. Then we created *workload B* by adding to *workload A* an extra percentage of malicious requests. These malicious requests are similar to the ones in *workload A* but they are tagged as malicious to be reverted by μVerum. These malicious requests modify the state of the application by executing *update* operations in the database. The affected microservices have invariants to ensure that data integrity is maintained. More specifically, there are one ordering invariant and one atomic invariant. Finally, we recovered *database B* to produce *database C* and used our script to compare it with *database A*.

We repeated these experiments ten times to recreate different states that allowed us to verify the validity of μVerum recovery. We started by having two identical workloads of 10,000 valid operations. Then, in each experiment, we added 10% malicious requests. In all experiments μVerum was able to achieve a state *C* equivalent to *A*. This was expected given that the PATCH methods that we implemented were tested beforehand, and we validated that they were capable of reverting any executed operation.

C. PERFORMANCE OVERHEAD

To evaluate the performance overhead of logging operations with the μVerum agents, we performed a series of experiments in which we measure the number of requests per second with and without having μVerum logging the operations. To do so, we use the SockShop loadtest scripts configured to issue 10,000 requests simulating 5 concurrent users. First, we tested with SockShop, then we repeated the same workloads with μVerum logging the requests. We performed the tests using both the log methods of μVerum (asynchronous and synchronous).

Figure 3 shows the performance in requests per second of SockShop, with and without μVerum. The overhead of μVerum is around 3.7% for asynchronous logging and 6.4% for synchronous logging. Some endpoints (details.html) reveal lower overheads. This happens because these endpoints are not being logged, and the caching mechanisms of the applications allow the resource to be presented to the user without executing microservice operations.

In addition to SockShop, we also evaluated the performance overhead of using μVerum to log every user request in Piggy Metrics. Since Piggy Metrics does not have a load test tool, we created a test case with Apache JMeter [64]. JMeter

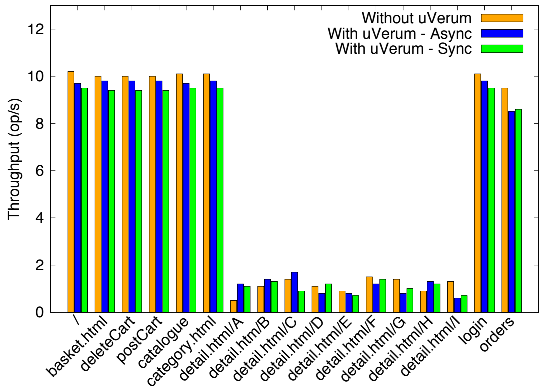


FIGURE 3: Performance overhead of μVerum with in different URLs of SockShop. The results refer to the application without μVerum (orange bar), the application with μVerum intercepting the requests asynchronously (blue bar) and with μVerum intercepting the requests synchronously (green bar).

allows us to create a workload in which a set of URLs are randomly invoked. In our test case, JMeter executed the 11 URLs available by the Piggy Metrics application until it reached 10,000 executions. Then we calculated the average operations per second of each of the URLs. The results, shown in Figure 4, show that there is a performance penalty for using μVerum to log every user operation that varies from 1.8% to 13.3% for asynchronous logging and from 11.2% to 31.1% for synchronous logging. The overhead of using μVerum in PiggyMetrics is higher, compared to using μVerum in SockShop, because we used a different load test tool. In Piggy Metrics, since we did not have an embedded load testing tool, we used JMeter, which allowed us to create a heavier workload than the one used in SockShop. Read operations that do not need to be logged (issued using the GET method) present lower overheads than the ones that modify the state of the microservice (PUT and POST methods). The overall results can be reduced by improving the computing capacities of the virtual machines that host the microservices.

Table 2 presents a comparison of various recovery mechanisms for different recovery systems, including a cloud file system, databases, web applications, and μVerum. The table shows that the overhead of recovery mechanisms can vary widely across the different target systems. In the table the performance overhead values were calculated either by measuring the extra time it takes to execute operations (latency - L) or reduction in the number of operations executed by second (throughput - T). μVerum offers both synchronous and asynchronous recovery mechanisms, with overheads ranging from 1.8% to 31.1%. In comparison, the other recovery mechanisms for web applications have overheads ranging from 3.99% to 30.35%. For file systems and databases, the overheads are relatively lower, ranging from 6% to 26%.

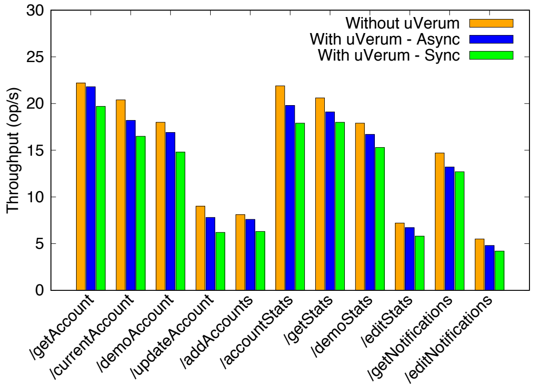


FIGURE 4: Performance overhead of μVerum in Piggy Metrics. The results refer to the application without μVerum (orange bar), the application with μVerum intercepting the requests asynchronously (blue bar) and with μVerum intercepting the requests synchronously (green bar).

TABLE 2: Performance overhead of the different recovery approaches. The overhead is in percentage range and it refers to the decreased performance in terms of Throughput (T) or additional latency (L) for each request.

System	Target system	Overhead
RockFS [65]	Cloud FSs	11% - 26% (L)
Amman et al. [66]	DBs	-
NoSQL Undo [9]	NoSQL DBs	6%-8% / 20%30% (T)
Akkus et al. [18]	Web apps	3.99% / 4.12% (T)
Warp [67]	Web apps	24% - 27% (T)
Aire [12]	Web apps	18,5% - 30,35% (T)
Shuttle [16]	Web apps	13% - 16% (T)
Sanare [68]	Web apps	12% - 17% (T)
Rectify [17]	Web apps	14% - 18% (T)
MIRES [69]	BaaS	15% - 23% (L)
μVerum	Microservices	async: 1.8% - 13.3% sync: 11.2% - 31.1% (T)

D. MEAN TIME TO RECOVER

Mean Time to Recover (MTTR) is the time that it takes since the moment the system administrator starts recovery until every PATCH method is successfully executed. To evaluate the MTTR we executed recovery in SockShop and reverted a set of intrusions that ranged from 10 to 100. We repeated this process 10 times. We performed these experiments with the two recovery methods of μVerum: with atomic invariants and with order invariants.

Figure 5 presents the results for the MTTR with atomic invariants. The time to recover increases linearly, varying from 5 seconds (for 10 intrusions) to around 50 seconds (for 100 intrusions). Undoing a single intrusion would take 0.5 seconds.

Figure 6 presents the MTTR from 10 to 100 intrusions using order invariants. The MTTR varies linearly from 11 seconds (for 10 intrusions) to 107 seconds (for 100 intrusions). We implemented a PATCH method that required 5 operations to be performed in a specific order.

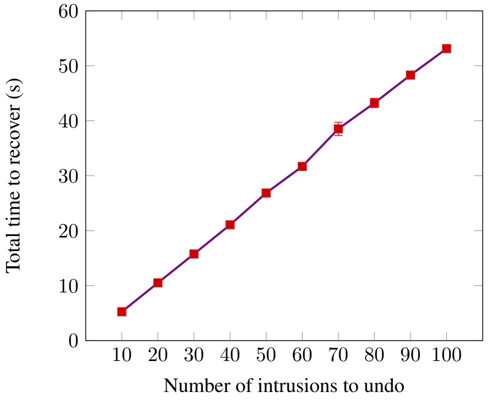


FIGURE 5: Total time to recover with *atomic invariants* changing the number of intrusions to undo.

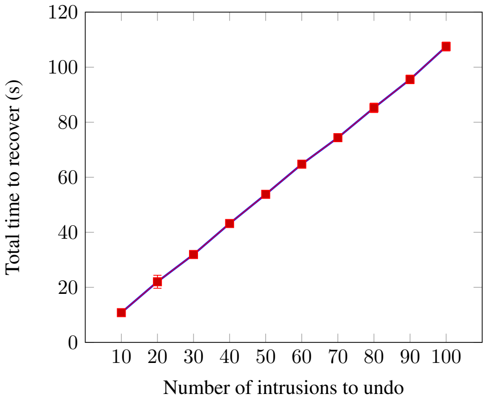


FIGURE 6: Total time to recover with *order invariants* changing the number of intrusions to undo.

The PATCH method that we implemented executes 6 microservices requests that affected 5 services. The MTTR varies depending on two factors: the scale of the application and the complexity of the PATCH method.

In Figure 7 we show the time it takes for a user request to be processed before, during, and after recovery. We repeated these tests by varying the number of intrusions from 1 to 10. During these experiments, we also performed the same load test described in Section V-C to simulate a real-world application with several concurrent users. The peaks in the graph correspond to when the administrator initiates a recovery process. Users experience a delay in the application for a while, but once recovery finishes, the application resumes normal operation. The latency ranged from around 30 ms during normal operation to a couple of seconds (from 1 to 5) during recovery. It is a significant downgrade in performance but we consider that this is an acceptable cost given the benefit of reverting an intrusion of the application without sacrificing the availability. This is the worst case, as the performance impact of recovery can be eased with request execution throttling. However, this makes the recovery process take longer to complete.

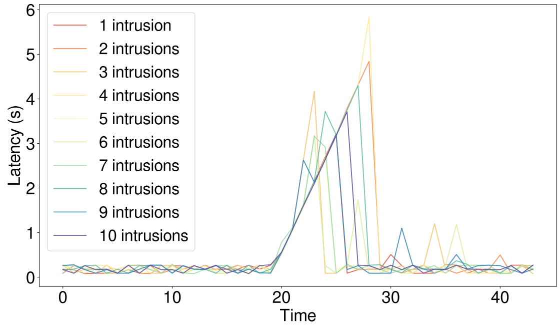


FIGURE 7: Latency of the application before, during and after recovery with *atomic invariants*, changing the number of intrusions.

Note that the throughputs and times presented in this evaluation are necessarily valid only for the microservice applications that we tested. More complex applications, or applications with different characteristics, might provide different results.

Table 3 presents a comparison of the MTTR μVerum with other recovery mechanisms for different recovery systems. The table shows that the recovery time varies in the different recovery systems, which is expected given the heterogeneity of the different target systems. In the table there are two columns for the MTTR: batch and unit. The batch refers to the MTTR a set of operations, while the unit refers to the MTTR of a single operation. The batch size varies because each author uses a different size in their own experiments. For comparison purposes we added the unit column to somehow compare the different systems.The MTTR in the different systems varies widely, from 16ms (Aire [12]) to 700s in NoSQL Undo [9].

TABLE 3: MTTR of the different recovery approaches.

System	**Target system**	**MTTR (batch)**	**Batch size**	**MTTR (unit)**
RockFS [65]	Cloud FSs	40s	100 (files)	2s
Amman et al. [66]	DBs	-	-	-
NoSQL Undo [9]	NoSQL DBs	150s - 200s	10,000 (ops)	1s / 700s
Akkus et al. [18]	Web apps	-	-	-
Warp [67]	Web apps	3,538s	2,093	1.69s
Aire [12]	Web apps	84.06s	5,444	16ms
Shuttle [16]	Web apps	544s - 1,717s	1,000,000	0.5ms - 1.7ms
Sanare [68]	Web apps	90 - 340s	10 - 60	1.8s - 6s
Rectify [17]	Web apps	960s	1,000	12s
MIRES [69]	BaaS	55s	1,000	1s
μVerum	Microservices	atomic: 5s - 50s ordering: 11s - 107s	10 - 100	atomic: 0.5s ordering: 1s

VI. RELATED WORK

The problem of intrusion recovery based on the use of logging operations was explored for databases and is covered in textbooks in the area, e.g., [13]. This work follows a more recent line of work on recovering databases [14], [15], operating systems [10], web applications [16]–[18] and other services [11] by assuming an architecture in which the components of the application are distributed and were implemented in distinct programming environments.

Undo for Operators implements the *"three R's"* model (Rewind, Repair, Replay. It logs every message in a log (*Timeline Log*) with the help of a proxy (*Undo Proxy*) and provides a control panel (*Undo Manager*) that is controlled using an interface (*Control UI*) by the system administrator. The μVerum approach is inspired in some of the components of Undo for Operators, more specifically, the use of logs to record user requests, the implementation of a Control UI that allows the administrator to monitor and perform recovery, and the use of proxies to intercept operations.

Warp [67] is a recovery system for web applications. It works by rolling back a part of the database to a point in time prior to the intrusion and then applying compensation operations to correct the state of the database. Warp uses a browser web browser extension to re-execute HTTP requests. The μVerum approach differs from Warp in the sense that it does not roll back the application to recover, and μVerum agents are used for HTTP requests instead of a browser.

Bezoar [21] is a recovery system that logs file system operations triggered by a virtual machine that supports the application. It requires a virtual machine to host the application to work. In this system model, in which the microservices are deployed in distinct environments, the system administrator may not have control of the virtual machine and therefore is not able to use Bezoar.

The problem of intrusion recovery for web applications deployed in PaaS was explored in Shuttle [16]. Like μVerum, Shuttle requires the developers of the application to implement some functionalities to be able to recover the application. Shuttle was designed for monolithic applications; it cannot be used for microservice applications. Another work that explores intrusion recovery for web applications, thus with the same limitation, is Rectify [17]. Rectify does not require software modifications to the application, since machine learning algorithms are used to find the database effects of HTTP requests.

Aire [12] is an intrusion recovery system for applications composed of interconnected web services. Aire works by propagating repair actions across services to address the unavailability of some services and ensuring consistency when not all repair actions have been propagated yet. Like μVerum, Aire logs operations during normal execution of the application, and once the administrator marks a request as malicious, it undoes its effects in the state of the application. However, Aire employs a selective reexecution approach to recover from the intrusion, while μVerum executes compensating operations. Additionally, Aire assumes a system model in which some web services may be compromised by an adversary who will try to sabotage the recovery process.

Table 4 compares the μVerum approach with other intrusion recovery systems.

VII. CONCLUSION

We propose μVerum, an intrusion recovery approach for microservice applications in both choreography and orchestration architectures. The design, implementation, and evaluation of our proposal show that it is possible and practical to recover from intrusions in microservices, as demonstrated with experiments using two distinct applications. The effort to implement the necessary compensating operations for μVerum is equivalent to implementing the required compensating operations used in sagas transactions. Our results show that it is possible to have the application available to users during recovery, at the expense of only a momentary degradation of performance. To maintain data consistency during and after recovery, we presented two algorithms that allow developers to define invariants for microservice operations that have ordering and atomic requirements. This makes it feasible to have recovery capabilities as soon as possible, and start delivering value to customers through fast and reliable intrusion recovery.

REFERENCES

[1] S. Newman, *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[2] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, 2017, pp. 195–216.

[4] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[5] N. Santos and A. R. Silva, "A complexity metric for microservices architecture migration," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

[6] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, Inc., 2008.

[7] OWASP, "OWASP Top 10 2021," https://owasp.org/Top10/, 2021.

[8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Addison-Wesley Professional, 2021.

[9] D. Matos and M. Correia, "NoSQL Undo: Recovering NoSQL databases by undoing operations," in *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*, Nov. 2016.

[10] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 89–104.

[11] A. B. Brown and D. A. Patterson, "Undo for operators: Building an undoable e-mail store," in *Proceedings of the USENIX Annual Technical Conference*, 2003, pp. 1–14.

[12] R. Chandra, T. Kim, and N. Zeldovich, "Asynchronous intrusion recovery for interconnected web services," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 213–227.

[13] J. Widom, H. Garcia-Molina, and J. D. Ullman, *Database Systems: The Complete Book*, 2nd ed. Pearson, 2009.

[14] P. Liu, J. Jing, P. Luenam, Y. Wang, L. Li, and S. Ingsriswang, "The design and implementation of a self-healing database system," *Journal of Intelligent Information Systems*, vol. 23, no. 3, pp. 247–269, 2004.

[15] T.-C. Chiueh and D. Pilania, "Design, implementation, and evaluation of a repairable database management system," in *Proceedings of the 21st IEEE International Conference on Data Engineering*, 2005, pp. 1024–1035.

[16] D. Nascimento and M. Correia, "Shuttle: Intrusion recovery for PaaS," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, 2015, pp. 653–663.

[17] D. R. Matos, M. L. Pardal, and M. Correia, "Rectify: black-box intrusion recovery in paas clouds," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 209–221.

[18] İ. E. Akkuş and A. Goel, "Data recovery for web applications," in *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010, pp. 81–90.

[19] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara, "The Taser intrusion recovery system," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, vol. 39, no. 5, 2005, pp. 163–176.

TABLE 4: Comparison of the μVerum approach with other intrusion recovery systems.

Recovery System	**Target**	**Selective re-execution**	**Compensating operations**	**Multiversioned**	**External inconsistencies**	**Online recovery**	**Real-Time Recovery**
Operator Undo [11]	Email systems	X			Compensating operations	No	No
EFS [70].	File systems			X	Not mentioned	Yes	No
RFS [71]	File systems		X		Not mentioned	Yes	No
Taser [19]	File systems	X			Compensating operations	No	Yes
Solitude [23]	File systems	X			Compensating operations	No	No
Retro [10]	File systems	X			Compensating operations	No	No
RockFS [65]	Cloud FSs	X	X	X	Compensating operations	Yes	No
Amman et al. [66]	DBs		X		Not mentioned	Yes	No
NoSQL Undo [9]	NoSQL DBs	X	X		Compensating operations	Yes	Yes
Akkuş et al. [18]	Web apps		X		Not mentioned	No	No
Warp [67]	Web apps	X			Compensating operations	Yes	No
AIRE [12]	Web apps		X		Parallel recovery (like Git)	Yes	No
Shuttle [16]	Web apps	X	X		Compensating operations	Yes	No
Sanare [68]	Web apps	X	X	X	Compensating operations	Yes	No
Rectify [17]	Web apps	X	X		Compensating operations	Yes	No
MIRES [69]	BaaS	X	X		Compensating operations	Yes	No
μVerum	**Microservices**		**X**		**Compensating operations**	**Yes**	**Yes**

[20] D. Vaz, D. R. Matos, M. Pardal, and M. Correia, "MIRES: Recovering mobile applications based on backend-as-a-service from cyber attacks," in *Proceedings of the 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2020.

[21] D. Oliveira, J. R. Crandall, G. Wassermann, S. Ye, S. F. Wu, Z. Su, and F. T. Chong, "Bezoar: Automated virtual machine-based full-system recovery from control-flow hijacking attacks," in *Proceedings of the IEEE Network Operations and Management Symposium*, 2008, pp. 121–128.

[22] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 223–236, 2003.

[23] S. Jain, F. Shafique, V. Djeric, and A. Goel, "Application-level isolation and recovery with solitude," in *ACM SIGOPS Operating Systems Review*, vol. 42 (4), 2008, pp. 95–107.

[24] H. F. Korth, E. Levy, and A. Silberschatz, "A formal approach to recovery by compensating transactions," in *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990, pp. 95–106.

[25] A. Wang and S. Tonse, "Announcing ribbon: Tying the Netflix mid-tier services together," 2018, https://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html.

[26] M. Roesch, "Snort: Lightweight intrusion detection for networks." in *Proceedings of LISA'99: 13th Systems Administration Conference*, 1999, pp. 229–238.

[27] K. L. Ingham and H. Inoue, "Comparing anomaly detection techniques for HTTP," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, 2007, pp. 42–62.

[28] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Computer Networks*, vol. 48, no. 5, pp. 717–738, 2005.

[29] G. Nascimento and M. Correia, "Anomaly-based intrusion detection in software as a service," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2011.

[30] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *Proceedings of the 13th Symposium on Network and Distributed System Security*, Feb. 2006.

[31] D. E. Denning and P. G. Neumann, "Requirements and model for IDES - a real-time intrusion detection expert system," Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep., 1985.

[32] H. Debar, M. Dacier, and A. Wespi, "A revised taxonomy of intrusion detection systems," *Annales des Télécommunications*, vol. 55, no. 7, pp. 361–378, 2000.

[33] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering*, 2018, pp. 11–20.

[34] A. Hannousse and S. Yahiouche, "Securing microservices and microservice architectures: A systematic mapping study," *Computer Science Review*, vol. 41, p. 100415, 2021.

[35] WeaveWorks, "Sockshop: A microservices demo application," 2018, https://www.weave.works/blog/sockshop-microservices-demo-application.

[36] Piggy Metrics, "Piggy metrics: A simple way to deal with personal finances," 2018, https://github.com/sqshq/piggymetrics.

[37] B. Downing-Troy, *Java RMI: remote method invocation*. IDG, 1998.

[38] S. McLean, K. Williams, and J. Naftel, *Microsoft .NET Remoting*. Microsoft Press, 2002.

[39] E. Newcomer and G. Lomow, *Understanding SOA with Web services*. Addison-Wesley, 2005.

[40] N. Jackson, *Building Microservices with Go*. Packt Publishing Ltd, 2017.

[41] C. O'Meara, "Reactive Kafka microservice template," https://github.com/omearac/reactive-kafka-microservice-template, 2018.

[42] T. Mauro, "Adopting microservices at Netflix: Lessons for architectural design," 2018, https://www.nginx.com/blog/ microservices-at-netflix-architectural-best-practices/.

[43] Netflix, "Zuul," 2018, https://github.com/Netflix/zuul.

[44] NetFlix, "Eureka," https://github.com/Netflix/eureka, 2018.

[45] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[46] The GNU C Library, "Backtraces," 2014, https://www.gnu.org/ software/libc/manual/html_node/ Backtraces.html.

[47] The Python Software Foundation, "traceback – print or retrieve a stack traceback," 2014, https://docs.python.org/3/ library/traceback.html.

[48] L. Dusseault and J. Snell, "Patch method for http, RFC 5789," Internet Engineering Task Force (IETF), Tech. Rep., 2010.

[49] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006, pp. 13–15.

[50] J. Williams and D. Wichers, "OWASP Top 10 - 2017 rcl - the ten most critical web application security risks," OWASP Foundation, Tech. Rep., 2017.

[51] National Computer Security Center, "Trusted computer systems evaluation criteria," Aug. 1983.

[52] C. Sima, J. Scambray, and V. Liu, *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions*. McGraw Hill, 2011.

[53] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[54] M. Štefanko, O. Chaloupka, B. Rossi, M. van Sinderen, and L. Maciaszek, "The saga pattern in a reactive microservices environment," in *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*. SciTePress Prague, Czech Republic, 2019, pp. 483–490.

[55] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Shirley, J.and Evans, "Automatically hardening web applications using precise tainting," in *IFIP International Information Security Conference*, 2005, pp. 295–307.

[56] I. Papagiannis, M. Migliavacca, and P. Pietzuch, "Php aspis: using partial taint tracking to protect against injection attacks," in *2nd USENIX Conference on Web Application Development*, 2011.

[57] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006, pp. 175–185.

[58] M. Cinque, R. Della Corte, and A. Pecchia, “Microservices monitoring with event logs and black box execution tracing,” *IEEE Transactions on Services Computing*, 2019.

[59] S. P. T. Krishnan and J. L. U. Gonzalez, “Google Compute Engine,” in *Building Your Next Big Thing with Google Cloud Platform*. Springer, 2015, pp. 53–81.

[60] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis, and S. Deleuze, “Spring boot reference guide,” *Part IV. Spring Boot features*, vol. 24, 2013.

[61] Innovate Labs, “Little shoot proxy,” 2018, https://github.com/adamfisk/LittleProxy.

[62] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O’Reilly Media, Inc., 2013.

[63] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems.” in *USENIX annual technical conference*, 2010.

[64] E. H. Halili, *Apache JMeter: A practical beginner’s guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[65] D. R. Matos, M. L. Pardal, and M. Correia, “RockFS: Cloud-backed file system resilience to client-side,” in *Proceedings of the 2018 ACM/IFIP/USENIX International Middleware Conference*, 2018.

[66] P. Ammann, S. Jajodia, and P. Liu, “Recovery from malicious transactions,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1167–1185, 2002.

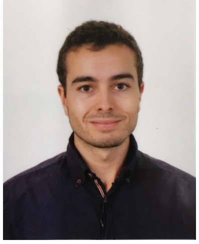
[67] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich, “Intrusion recovery for database-backed web applications,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 101–114.

[68] D. Matos, M. Pardal, and M. Correia, “Sanare: Pluggable intrusion recovery for web applications,” *IEEE Transactions on Dependable and Secure Computing*, 2021.

[69] D. Vaz, D. Matos, M. Pardal, and M. Correia, “MIRES: Intrusion recovery for applications based on backend-as-a-service,” *IEEE Transactions on Cloud Computing*, pp. 1–1, 2022.

[70] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, “Deciding when to forget in the Elephant file system,” in *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, 1999, pp. 110–123.

[71] N. Zhu and T.-c. Chiueh, “Design, implementation, and evaluation of repairable file service,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2003, p. 217.



DAVID R. MATOS has a BSc (2012) and a MSc (2013) in Informatics Engineering from the Faculty of Sciences, University of Lisbon and a PhD (2019) in Computer Sciences and Engineering from Instituto Superior Técnico, University of Lisbon. He is currently an Invited Assistant Professor at Instituto Superior Técnico (IST), University of Lisbon and a researcher at INESC-ID in the Distributed, Parallel and Secure Systems group (DPSS). His research interests are in the area of Distributed Systems and Cybersecurity.

MIGUEL L. PARDAL graduated (2000), mastered (2006), and doctored (2014) in Computer Science and Engineering from Instituto Superior Técnico (IST), University of Lisbon, Portugal. He is an Assistant Professor at IST and a researcher at INESC-ID in the Distributed, Parallel and Secure Systems group (DPSS), where he is leading the SureThing project (FCT) and completed a participation in the Safe Cloud EU Project (H2020). He is also a Guest Scientist at the Chair of Network Architectures and Services at TU Munich. During his PhD, he was a visiting student at the Auto-ID Labs at MIT. His current research interest is in Cybersecurity applied to the digital frontiers of the Internet of Things and Cloud Computing.

ANTÓNIO RITO SILVA is an Associate Professor at Instituto Superior Técnico (IST), University of Lisbon and a researcher of the Distributed Parallel and Secure Systems group (DPSS) at INESC-ID. He received a PhD in software engineering in 1999 from IST. His research interests include software architectures for microservices, digital humanities, and business process management (BPM). António has published more than 90 peer-reviewed articles in journals, conferences and workshops. He is leading research on the migration of monolith applications to a microservices architecture (https: //github.com/socialsoftware/mono2micro).

MIGUEL CORREIA is a Full Professor at Instituto Superior Técnico (IST), Universidade de Lisboa, senior researcher and member of the board of INESC-ID, and member of the Distributed, Parallel and Secure Systems group (DPSS). He has been involved in many international and national research projects related to Cybersecurity (BIG, DE4A, QualiChain, SPARTA, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, MAFTIA) and has more than 200 publications. His research focuses on Cybersecurity and Dependability (a.k.a. Fault Tolerance) in Distributed Systems, in the context of different application areas (Blockchain, Cloud, Mobile).

• • •