Attack tolerance for services-based applications in the cloud

Georges Ouffoué, Fatiha Zaïdi, Ana R Cavalli

Attack tolerance for services-based applications in the Cloud.

Georges Ouffoué[1], Fatiha Zaïdi[1] and Ana R. Cavalli[23]

[1] LRI, Univ. Paris-Sud, UMR 8623 CNRS, Université Paris-Saclay
{ouffoue,zaidi}@lri.fr
[2] IMT/TELECOM SudParis, SAMOVAR, UMR 5157 CNRS, Evry, France
ana.cavalli@it-sudparis.eu
[3] Montimage Paris
ana.cavalli@montimage.com

Abstract. Web services allow the communication of heterogeneous systems and are particularly suitable for building cloud applications. Furthermore, such applications must verify some static properties, but also tolerate attacks at runtime to ensure service continuity. To achieve this, in this paper we propose an attack tolerance framework that includes the risks of attacks. After describing the foundation of this framework, we propose expressing cloud applications as choreographies of services that take into account their distributed nature. Then, we extended the framework to introduce choreography verification by incorporating monitoring (passive tests) and reaction mechanisms. These techniques are validated through relevant experiments. As a result, our framework ensures the required attack tolerance of such cloud applications.

Keywords: Attack tolerance · Runtime verification · Monitoring · Web services and cloud · Passive tests · Software Reflection

1 Introduction

Computer systems are now at the heart of all business functions (accounting, customer relations, production, etc.) and, in general, in everyday life. These systems are based on heterogeneous applications and data. Service Oriented Architectures (SOA) have been proposed for this purpose. These architectures are distributed and facilitate communication between environments of heterogeneous nature. The main components of such architectures are Web services. A Web service is a collection of open protocols and standards for exchanging data between systems. These services can be internal and only concern one organization. In addition, the need to expose services to the outside world is growing due to the technological advances in communication networks, especially the Internet. Besides, security is at the heart of business concerns. Web services, since they are open and inter-operable, are privileged entry points for attacks. Moreover, Web services deployed in the cloud inherit their vulnerabilities. Security must then be

taken into account when implementing Web services and this at all levels : design, specification, development and deployment. It is also appropriate to quantify the risks in order to clearly identify the threats and reduce the vectors of attacks.

In this paper, we adopt a new end-to-end security approach based on risk analysis, formal monitoring, software diversity and software reflection. We propose a new formal monitoring methodology that takes into account the risks that Web and cloud services may face. More precisely, the contributions of the paper are the following:

- A risk-based monitoring methodology is proposed. We claim that the detection and prevention of attacks require a good knowledge of the risks that these systems are facing.
- An instantiation of this methodology for cloud applications based on Web services is described. We propose an attack tolerance framework (offline and online), for such applications. Indeed, it is appropriate to consider tolerance during the modelling of the application and also to monitor that application at runtime for anticipating and detecting attacks *w.r.t* to the risks . For this goal:
  - We first consider any application deployed in the cloud as a choreography of services which must be continuously monitored.
  - For the verification and monitoring of the choreography obtained, we extend a formal framework for choreography verification by incorporating our previous detection and remediation strategies [3].
  - We finally propose a new Domain Specific Language (DSL) called *ChorGen*. If choreographies written in a process algebra are formally verified and projected on the peers, skeletons of the corresponding services are generated by *ChorGen*.

The paper is then organized as follows. We propose a quick presentation of the main attack tolerance techniques in section 2. Section 3 fully describes the risk-based monitoring methodology. Following the above methodology, an attack tolerance framework for cloud applications is presented in section 4. In section 5, we present a concrete case study: an electronic vote system. Experiments on this use-case highlight the attack tolerance capability of the whole framework. Conclusions and future enhancements of this work are given in section 6.

2 Attack tolerance: state of the art

This section presents existing attack tolerance techniques highlighting the main issues that remain unsolved.

2.1 Intrusion and attack tolerance

Several solutions for attack tolerance were proposed. [8] proposed a formalism based on graphs to model an intrusion tolerant system. In this model they introduce system's response to (some of) the attacks. They call this model

that incorporates attacker's actions as well as the system's response an Attack Response Graph (ARG).

Besides [19] classify ITS (Intrusion Tolerant Systems) architectures into four categories :

- **Detection-triggered** [16, 17]: these architectures build multiple levels of defense to increase system survivability. Most of them rely on an intrusion detection that triggers reactions mechanisms.
- **Algorithm-driven** [10, 18] : these systems employ algorithms such as the voting algorithm, threshold cryptography, and fragmentation redundancy scattering (FRS) to harden their resilience.
- **Recovery-based** [1, 14] : these systems assume that when a system goes online, it's compromised. Periodic restoration to a former good state is necessary.
- **Hybrid** [15]: these systems combine different architectures mentioned above.

2.2 Attack tolerance techniques for services-based application

It must be noted that Web services deployed in the cloud or used for building cloud applications inherit the vulnerabilities of the cloud platforms. Few works were conducted in order to transpose the techniques and framework cited in the previous section to web services. [5, 13] presented an attack tolerant Web service architecture based on diversity techniques presented above. These solutions protect essentially against XML DoS Attacks. While these approaches are interesting, they do not address the specificity of services-based application deployed on cloud platforms. The solutions are attack-specific. Moreover, for this kind of application, it is necessary to integrate security in all the process steps *i.e.* from modeling to deployment. We need a more efficient intrusion-tolerant mechanism.

3 Risk-based monitoring methodology

The supervision or monitoring of information systems is of paramount importance for any organization. It essentially consists in deploying probes in various parts of the system based on preset checkpoints. With automatic failure reporting, network agents can respond to key security risks. The disadvantage of such methods is the following. If risks or failures are discovered during operation, the attacks may have already occurred and one or more parts of the system may be non-functional. So, the detection and prevention of attacks require a good knowledge of the risks that these systems face.

As such, it is mandatory to include risk management in the monitoring strategy in order to reduce the probability of failure or uncertainty. Risk management attempts to reduce or eliminate potential vulnerabilities, or at least reduce the impact of potential threats by implementing controls and/or countermeasures. In the case, it is not possible to eliminate the risk, mitigation mechanisms should be

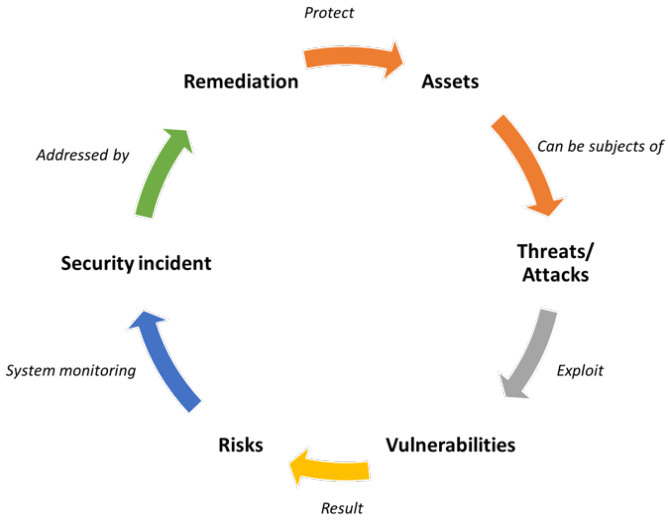


Fig. 1: Risk-based monitoring loop

applied to mitigate their effects. We leveraged the risk management loop to build our risk-based monitoring loop depicted on the Figure 1. Indeed, this risk-based monitoring solution can be summarized in the following objectives:

3.1 Identifying Assets

Assets are defined as proprietary resources of value to the organization and necessary for its proper functioning. We distinguish business-level assets from system assets. In terms of business assets, we mainly find information (for example credit card numbers) and processes (such as transaction management or account administration). The business assets of the organization are often entirely managed through the information system. System assets include technical elements, such as hardware, software and networks, as well as the computer system environment, such as users or buildings. System assets can also represent some attributes or properties of the system such as the data integrity and availability. This is particularly true for cloud services consumers.

3.2 Risk and vulnerability analysis

Risk is the possibility or likelihood that a threat will exploit a vulnerability resulting in a loss, unauthorized access or deterioration of an asset. A threat is a potential occurrence that can be caused by anything or anyone and can result in an undesirable outcome. Natural occurrences, such as floods or earthquakes, accidental acts by an employee, or intentional attacks can all be threats to an

organization. A vulnerability is any type of weakness that can be exploited. The weakness can be due to, for example, a flaw, a limitation, or the absence of a security control.

3.3 Threats identification

The first step to perform to avoid or repel the different threats that can affect an asset is to identify: affected modules/components, actions/behaviour to trigger the threat, and potential objective of the threat. This identification helps to understand the operation of the attacks and allows the creation of security mechanisms to protect, not only the assets, but also the software mechanisms that support them. The threats can be modelled by graphical representations such attack trees.

3.4 System security monitoring

The monitoring mechanism we propose, allows to constantly monitor activities or events occurring in the network, in the applications, and in the systems. This information will be analysed in near real-time to early detect any potential issue that may compromise the security or data privacy. If any anomalous situation is detected, the monitoring module will trigger a series of remediation mechanisms (countermeasures) oriented to notify, repel, or mitigate attacks and its effects.

3.5 Remediation

Once the risks of any system are established and the means of detection identified, it is essential to think about how to set up mechanisms that will allow to complete the risk-based monitoring loop *i.e.*, to tolerate and mitigate the effects of the potential detected attacks.

4 Risk-based monitoring for services-based applications in the cloud

This section presents how we have instantiated the risk-based methodology for services-based applications deployed in the cloud. As the first stages of the risk-based monitoring loop are specific to the type of the application, we will focus on the last two phases of that loop : monitoring and remediation. Two main approaches of remediation can be described:

– Anticipating the attack tolerance capability. This consists in introducing mechanisms allowing the tolerance to the attacks during the modelling of the system. The system is likely said to be tolerant-by-design or offline tolerant to attacks.

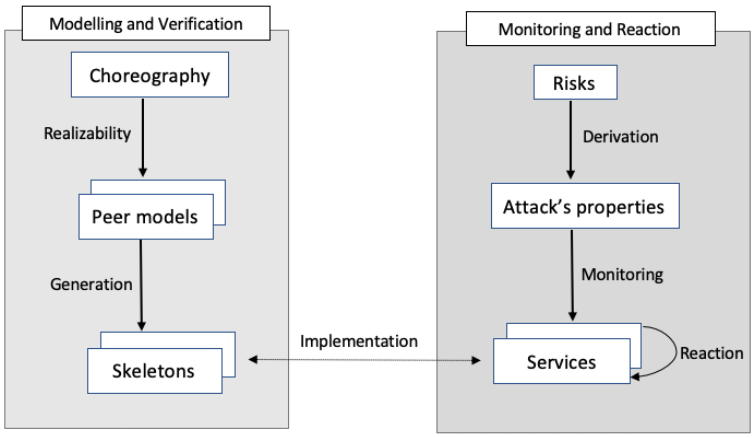


Fig. 2: Architecture and components of the framework

– Considering tolerance by a constant monitoring. In this type of approach, the tolerance capacity is entirely managed by the monitoring tool. The system is actively monitored for detecting malicious behaviours. The system is likely said to be online tolerant to attacks.

We believe that to have an effective attack tolerance (offline and online), it is appropriate to use these two approaches in a complementary way. We therefore propose an attack tolerance both online and offline. The resulting framework consists of two main parts (Figure 2). The first part will present how we model services-based applications deployed in the cloud to make them attack tolerant. The second part will present how we monitor the system to detect the attacks; and how we mitigate these attacks.

4.1 Modelling of cloud applications

Cloud applications are distributed applications. To fully benefit from the advantages of the cloud we will consider our applications as a composition of Web services deployed in the cloud.

Generally service compositions are classified into two styles: orchestrations and choreographies. Orchestration always represents control from one participant's perspective, called the orchestrator. Unlike the orchestration, there is no privileged entities in the choreography. [6] argued that Web services composition, in particular choreography is a suitable solution used to build application and systems in the cloud. They built a middleware solution that is capable of automatically deploying and executing Web services in the cloud. We agree with them that choreography is a good approach for deploying cloud applications

based on web services. The applications in the cloud will be deployed as service choreographies that integrate attack tolerance features. However, before and when deploying such choreography one should ensure that this choreography is realizable. Realizability, a fundamental issue of choreography, is whether a choreography specification can correctly be implemented. In a top-down service choreography approach, the realizability issue results in verifying whether a choreography model can be correctly projected onto role models. For this goal, we will leverage SChorA [9], a verification and testing framework for choreographies.

SChorA. SChorA [4] was proposed by [9]. This framework aims to solve the key issues in choreography-based top-down development: i) Realizability: Whether a choreography is realizable *i.e* ensuring that a choreography can be practically implemented. ii) Projection: Ability to derive local models of a global choreography on peers. In order to easily express the choregraphies, a language, $ChorD$ which is an extension of the $Chor$ language [12] with data, has been proposed. $Chor$ language is expressive and abstract enough to enable one to specify collaborations but lack data support, what $ChorD$ covers.

The basic event in choreography is an interaction. An interaction represents a communication between two roles. There are two kinds of interactions: free interactions and bound interactions. A free interaction represents a communication of value of variable x realized through an operation o from role a to b is denoted by $o^{[a,b]}.x$, while the bound one is denoted $o^{[a,b]}.\langle x\rangle$. In free interaction, the data exchange must be known before the interaction may occur. In bound interaction, the data exchange is bounded at the moment the interaction occurs

$ChorD$ is described as:

$$ChorD ::= 1|\alpha|A; A|A + A|A\|A|A[> A|[\phi] \triangleright A|[\phi] \star A$$

A basic activity is either an inaction (1), or a standard basic event (α) presented above. They are structuring operators, that can be used to specify composite activities such as sequencing ($;$), non-deterministic choice ($+$), parallel activities ($//$), and interruption ($[>$).

One should note that we distinguish the global specification of the choreography called *global model* and the specification of this choreography on the different roles termed *role model*. In *role models* event are modeled as sending (!) or reception (?). For example let's express a simple Online-Shopping choreography between two roles: b (buyer) and v (vendor). The buyer first requests an article by providing an amount to be bought. If the amount is greater than 25 then the vendor aborts this transaction. Otherwise, a confirmation will be issue from the vendor to the buyer. This can be described as follows:

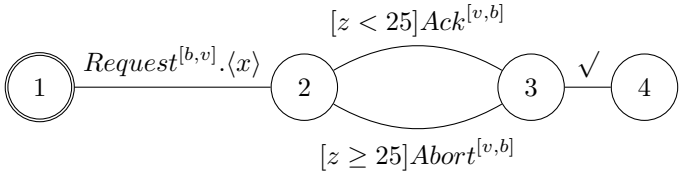
$$C :: Request^{[b,v]}.\langle x\rangle; ([x < 25] \triangleright Ack^{[v,b]} + [x \geq 25] \triangleright Abort^{[v,b]})$$

$$\text{For the Buyer: } Request^{[b,v]}!.\langle y\rangle; (Ack^{[v,b]}? + Abort^{[v,b]}?)$$

[4] http://SChorA.lri.fr

For the Vendor: $Request^{[b,v]}?.\langle z\rangle;([z<25]\triangleright Ack^{[v,b]}!+[z\geq 25]\triangleright Abort^{[v,b]}!)$

In fact *ChorD* is a process algebra and its semantics is given by Symbolic Transition Graphs (STGs) [7]. An STG is a transition system. Each transition of STG is labelled by a guard ϕ and a basic event α. The guard ϕ is a boolean equation which has to hold for the transition to take place. A symbolic transition from state s to state t with a guard ϕ, and an event α is denoted as $s \xrightarrow{[\phi]\alpha} t$. $\surd$ is added to denote activity termination. The representation of the simple shopping choreography using STG is the following:



For the realizability and projection issues, STG are also used. By their formal richness, STGs are perfectly suited for verification of choreographies. STGs can be expanded to describe more operations since they support data, guard and free/bound variables. From the representation of the choreography as STG, if there are non-realizable parts, some additional interactions are incorporated to the graph to allow all the transitions to be realizable. Once the realizability is verified (*i.e* it can be directly implemented or some additional interactions are added), they are projected on the different roles or peers. By doing so, we are sure that our local models projected can actually be implemented concretely. In our case, local models are implemented as a web service. Once this step is over, it is useful to implement these projected models on the peers. The next section presents how in this framework, skeletons of the services are made possible.

Code generation. In top-down software development approaches, an important part of the process is to reduce the costs of development by promoting modularity, reusability, and code generation. This is especially true in modeling and designing choreographies. It is therefore essential to have automatic mechanisms that perform code generation. This is why we propose a code generation strategy in this framework.

For this aim, we can leverage frameworks or tools in the literature ([11]) that take as input STGs and produce source code. Moreover, we propose a new Domain Specific Language (DSL) for our choreography called *ChorGen*.

The *ChorGen* language has the following grammar:

```
Model:
  (choreographies+= Choreography)+;
```

```
Choreography:
  'choreography' name=ID '{'
    (roles+=Roles)*
  '}'
;
Roles:
  'role' name=ID '{'
    operations+=Operation
  '}'
;
Operation:
  'operations' '{'
     (methods+=Function)*
  '}'
;
Function:
  name=ID '('(params+=Param)* ')'
;
Param:
  name=ID type=ID ',' |name =ID type=ID
;
```

We used model-driven engineering technologies Xtext [5] for the semantics and Xtend[6] for code generation to the target languages: WSDL and Python. This means that a choreography contains several roles that expose some operations to interact with the other roles. The advantage of doing such code generation is the reduction of the development costs and efforts. This allows us to be more efficient when implementing the services. This is useful, for example, for choreographies containing a very large number of peers. Another advantage is that since all interactions are taken into account, we are sure that developers will not forget to implement them since their signatures are available. Moreover, it should be noted that a single implementation is not sufficient to have a complete tolerance. It is admitted that more diversification implies more security.

4.2 Deployment, monitoring and reaction

We will leverage diversity as well. For implementing choreography, we had the choice between two methods. The first was to diversify for example at the level of programming languages, *i.e* having the implementations in different languages. This method has the advantage of generating few dependency between the variants of these services. However, this can create significant costs and workload for developers and can increase the time-to-market. The developer may not be able to master other languages. Moreover, since one of the members of the service

[5] https://www.eclipse.org/Xtext/
[6] https://www.eclipse.org/xtend/

choreography can be changed on the fly while the others remain intact, there could be some inconsistencies between the communication between these services. Indeed, although based on the remote procedure call (RPC), the ways to deploy web services are not the same.

The second way is to consider only one target programming language but have diversified implementation. The variants differ for example at the control flow (AST) level and use different data structures. The advantage of such an approach is that it is more flexible. The disadvantage is that we have a low diversity rate. However this rate may be improved by using different OS during the deployment of the services. For such reasons we chose the second method. So, from the global choreography, the local models are projected taking into account the interactions added to the specifications. This is the case for example when in the verification phase, interactions are added to the models to make the choreography realizable. After, there is a generation of the skeletons of the services that will implement the choreography. In particular, we generate the WSDL files (interface file of the Web services) as well as the skeletons of the implementations of these services in Python.

Besides, it is undeniable that to better tolerate attacks, it is necessary to detect them. Our approach of monitoring is based on reflection. Reflection makes it possible to dynamically get the code and even the execution trace of a method, a class and a module. One can also modify the class at run time. Using reflection all the hashes of the source code of any methods of the system are processed ([3]). An example of code using the reflection API in python is depicted in Figure 3. In this piece of code, the source code of the running function in the call stack is retrieved and the hash of this code is computed.

```
def show_stack():
    stack = inspect.stack()
    ''' Inspect the stack '''

    for s in stack:
        a=inspect.getsource(s[0])
        ''' Get the source '''

        m=hashlib.md5()
        ''' hash that source code'''
```

Fig. 3: An example of using reflection in python language.

In fact, hash functions by their robustness are used to ensure the integrity of messages or transactions in distributed systems. This is the case in modern protocols, for example ssh and bitcoin. As such, the detection of attacks leveraging hash functions is legitimate. Any deviation at runtime of that hash value means

the presence of a misbehavior. Such misbehavior could be caused by an insider attack or a virus attack. Information such as Date, Hour, Operation, hash, host are stored in the log file. Any request has then two traces in the logs : outbound(request) and inbound(response). For example, table 1 presents a situation where there is no attack. We observe that the hashes for the Outbound and Inbound requests are the same.

Table 1: Normal entries in the log of the client application

Date	Hour	Methode	Hash	Host
31/05/2019	10:00:00 AM	! update(Outbound)	2224d35250e...	a
31/05/2019	10:15:00 AM	? update(Inbound)	2224d35250e...	b

If there is an attack, the hashes of both Outbound and Inbound could not correspond in the log files. This is the case depicted on table 2. One can also get some other inconsistencies in the logs : hashes not equal, timestamps incoherence, method inconsistencies (answer before request), or combination of inconsistencies.

Table 2: Bad entries in the log of the client application

Date	Hour	Methode	Hash	Host
31/05/2019	10:00:00 AM	! update(Outbound)	2224d35250e...	a
31/05/2019	10:15:00 AM	? update(Inbound)	2504d35222e...	b

For detecting attacks, log are located on the peers. We developed a new plugin for this kind of detection in the monitoring tool MMT. The MMT (Montimage Monitoring Tool) is a solution for monitoring networks and applications. MMT's Security properties are written in XML format. This has the advantage of simple and straightforward structure verification and processing by the tool. Any security property is written in XML. Each property begins with a <property> tag and ends with </property>. A MMT-Security property is an IF-THEN expression that describes constraints on network events captured in a trace T = $\{p_1, ..., p_m\}$. It has the following syntax:

$$e_1 \xrightarrow{W,n,t} e_2$$

W ∈ { BEFORE, AFTER }, n ∈ $\mathbb{N}$, t ∈ $\mathbb{R}_{>0}$ and e_1 and e_2 two events. This property expresses that if the event e_1 is satisfied (by one or several packets p_i , i ∈ $\{1, ..., m\}$, then event e_2 must be satisfied (by another set of packets p_j , j ∈ $\{1, ..., m\}$) before or after (depending on the W value) at most n packets and t units of time. e_1 is called triggering context and e_2 is called clause verdict. When monitoring a system to detect attacks, the non respect of the MMT-Security property indicates the detection of an abnormal behaviour that might imply the occurrence of an attack. For example, if we consider a vote system (our use case deeply presented in the section 5) a rule in the MMT formalism is the following.

```
<beginning>
        <property value="THEN" delay_units="s" property_id="10"
            type_property="ATTACK"
              description="Detection of the insider attack: ">
                 <event value="COMPUTE" event_id="1"
                      boolean_expression="((#strcmp(log.method,
                      'vote(inbound)') != 0)&amp;amp;&amp;amp;(#strcmp(log.hash, '')!=
                      0))"/>

                 <event value="COMPUTE" event_id="2"
                      boolean_expression="((#strcmp(log.hash, '')!= 0)
                      &amp;amp;&amp;amp; (#strcmp(log.method, 'vote(outbound)') ==
                      0) &amp;amp;&amp;amp;(#strcmp(log.hash, log.hash.1) != 0))"/>
        </property>
</beginning>
```

Fig. 4: Security rule of the insider attack of the vote example

Figure 4 describes a property for detecting the insider attack according to the formalism of MMT (section4). This means that in the log file any vote request should have hashes for its operations (outbound and inbound) in the log files and these hashes must correspond; otherwise an attack is triggered. Event 1 (e_1) expressed the reception of the inbound operation in the log file with a hash. If event 2 (e_2), the reception of the corresponding inbound operation in the log appears, there is a comparison between the hash collected of that event and the hash obtained in the previous event e_1 (the built-in C function *strcmp* was used for the comparison). If the hashes correspond, the system is attack free. Otherwise, an alert is triggered.

Then every module is monitored by extracting the program stack using reflection. We have a local database(M-DB) in which all the sources of the program functions are securely stored. If the attack is not known in the M-DB (i.e the hash is not conform), the system checks in the own DB (M-DB). If the attack exists, countermeasures are launched else the hash is stored in the M-DB. For the mitigation of the attack, in case of attacks the current source code is replaced with one of the other variants randomly. The hash is also adapted. However, it should be noted that classic hash functions can provide the same hash for two different strings of characters. In practice, the probability of this happening is small. As a result, a more robust hash will not be foolproof either, but the probability of a collision will be higher and/or the means of generating it will be more complex and inaccessible.

5 Use case: Vote application

In this section, we present the implementation of the risk-based monitoring approach through a concrete case study. To illustrate our approach, we propose an electronic voting choreography for the election of the president in a certain country. This application allows citizens to register on the electoral lists and to vote electronically. The application is described by the *VoteElecService* choreography. It is composed of three basic members: *Inscription*, *Vote* and *Citizen*. The first member of that choreography allows to register a citizen on the electoral lists by providing the personal information (surname, first name, date of birth, address, ...). Now, we describe the main components of the risk-based monitoring associated to this use-case.

5.1 Identifying Assets

In line with our risk-based monitoring approach the main assets remain the votes of the citizens and the availability of the platform. These citizens must be able to vote at any time of the election day.

5.2 Risk and attack scenarios

The main vulnerability in the vote example is an insider malicious developer or a not cautious user of the vote choreography [2, 4]. Then, the following attacks (of course this is not an exhaustive list) can appear:

- Brute force : By analysing the unauthorised user's activities, user impersonation can sometimes be detected using MMT.
- Insider attacks (modification of the votes by a human being or by a virus): A member of the development team can modify the algorithm of the vote method in order to help a candidate or a political party of his choice to win the elections. We will provide some properties for such an attack.
- DoS/DDoS attacks: Making the vote service unavailable.

Brute force and DoS/DDoS attacks are classical and easy to detect and to mitigate. Using strong authentication such as 2-factors authentication and providing some classical MMT security rules can be sufficient. In addition, the reflection methodology is particularly useful for source code attacks. As such, due to space limitation, in the experiment part, we will focus on insider attacks that are more destructive.

5.3 System security monitoring and reaction

Modelling and verification. Let's first describe the choreography. Once registered, the citizen can vote electronically after verification of his registration by the member *Vote*. Subsequently, this service will provide the list of associated candidates and their identification number (1, 2, ...) in addition to the number

zero that is associated with the blank ballot. A registered citizen will vote by selecting one or more voting numbers (including the blank ballot) and submitting his/her choices. The choreography can be expressed in $ChorD$ as follows:

$$inscription^{[c,i]}.\langle info\rangle; voteRequest^{[c,v]}.\langle y\rangle; resultVerifInfo^{[v,i]}.\langle x\rangle;$$
$$([\text{x=0}] \triangleright rejection^{[v,c]} + [x! = 0] \triangleright (confirmation^{[v,c]}; liste^{[v,c]}; vote^{[c,v]})).$$

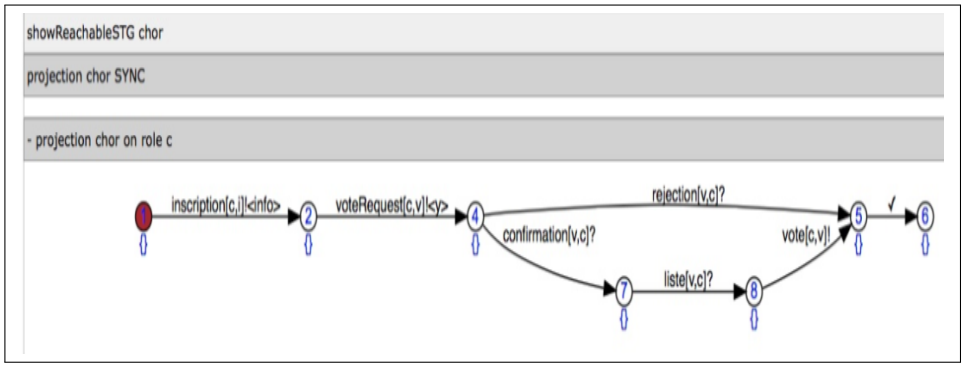


Fig. 5: Projection of the choreography on the peers

Where the Citizen member, the Inscription member, the Vote member and the result member correspond to respectively c, i and v. As one can observe on Figure 5, the choreography is fully realizable without the need of adding new interactions since the projection is effective and doesn't include such added interactions. And from these descriptions, we generate skeletons of the roles that the developer should complete later.

Deployment, experiments and results. The services were deployed on the Amazon Web Services (AWS) cloud platform. We used a virtual machine for each member of the choreography. AWS is among the leaders of the cloud computing market.

For the monitoring, Figure 4 presents a property for detecting the insider attack. This means that in the log file any vote request should have hashes for its operations (outbound and inbound) in the log files and these hashes must correspond; otherwise an attack is triggered.

The same choreography has been deployed on a local test-bed consisting of three Dell machines having two micro processors, 3 Go RAM memory and all using the Ubuntu OS in its latest version. The same choreography has been deployed on an amazon cloud. The three different virtual machines also have 3 Go RAM memory and 2 Vcpu. Some experiments where conducted to test the attack tolerance capability of our approach.

Experiment 1: Since the approach of the framework consists of modelling and deploying cloud-based applications as distributed service choreographies, we evaluated the latency of the service to respond to some amounts of requests on premises and in the cloud.

Table 3: Latency measurement

Number of requests	On premises	In the cloud
100	0.27 seconds	0.34
200	0.69 seconds	0.87
300	1.10 seconds	1.40
400	1.80 seconds	2.56
500	2.48 seconds	3.24
600	3.45 seconds	5.13
700	4.41 seconds	6.35
800	5.65 seconds	7.24
900	6.89 seconds	8.68
1000	8.41 seconds	9.17

As can be seen on Table 3, the response times to requests are substantially equal. The slight difference can be explained by the latency of the network. One way to significantly reduce this latency is if you have the choice to deploy virtual machines in the cloud in regions that are very close to where you are going to go. Then, the flexibility and cost reduction offered by cloud computing make the overhead negligible. The following experiments were conducted for testing the detection capability of the framework. Here, we only focus on attacks consisting in modifying the source code of a method or a function deliberated (by a human being) or made by a virus. As we explained earlier, it has been proven in the literature that this kind of attacks may appear in voting systems. For the sake of demonstration, we design a relatively harmless virus. This virus modifies all the codes of the classes, methods or functions of the python modules of a given directory tree. We generated a signature of the new virus and added to the virus database.

Experiment 2: We evaluated the time elapsed to detect the insider attacks coming from both a modification of the source code and a virus. The accuracy of the detection mechanism was discussed in [3]. In this section, we only evaluate the efficiency of the monitoring $w.r.t$ the two attacks above.

Table 4: Detection mean time

Virus	Modification
0.053 seconds	0.031 seconds

On Table 4 the modification seems to be easily detectable than the virus. This was predictable since the database of virus may contain a larger number of rows in comparison with the database containing the hashes of the methods. To have a fair detection time, one can have a unique database. The drawback of this solution is that we loose readability and flexibility. Along with the detection of attacks, the system reacts as mentioned in section 4. This reaction is transparent for the user. Although we can detect attacks with great granularity, it is also important to consider the impact of the monitoring mechanism. That's why we will measure the overhead of the new monitoring method in the next experiment.

Experiment 3: Evaluation of the impact of the monitoring mechanism.

Table 5: Overhead of the monitoring mechanism

Number of requests	Without Monitoring (s)	With monitoring (s)
100	0.38	0.46
200	0.88	0.91
300	1.39	1.4
400	2.50	2.57
500	3.26	3.27
600	5.22	5.30
700	6.31	6.35
800	7.21	7.45
900	8.89	9.12
1000	9.15	10.02

As one can see on Table 5, the overhead of the monitoring is not too significant. In future works, we will investigate how to reduce this overhead. The detection approach based on software reflection is suitable for the monitoring of cloud applications deployed as choreographies of services. To a certain extent, it can also be useful for detecting attacks such as buffer overflows and SQL injections. One limitation of such approach is the fact that this detection is only appropriate for attacks targeting the source code. But the main limitation is that the approach can not be applied in programs developed in programming languages that do not allow reflection or do not provide a powerful reflection API.

6 Conclusion

In this paper, we have proposed a new approach for attack tolerance based on formal runtime monitoring and software engineering techniques. We show that a good tolerance to attacks requires, on the one hand, to perform attack detection and continuous monitoring; and, on the other hand, reliable reaction mechanisms. In addition, we leverage the traditional risk management loop to build a risk-based approach that integrates risks into monitoring. Finally, we

proposed an offline and online attack tolerance framework for Web services-based application in the cloud. With this aim, we first express any application deployed in the cloud as a choreography of services, which must be continuously monitored and tested. Then, we extend a formal framework for choreography testing by incorporating the methods for detecting and mitigating attacks presented in the previous sections. Adding mechanisms of detection and reaction on the fly to these applications, ensure optimal attack tolerance. In the future work we will evaluate the scalability of the framework for very large choreographies. Besides, we believe that in addition to detection and remediation, it would be necessary to be able to predict and anticipate future attacks. We think that diagnosis and prediction techniques would be interesting to investigate in order to improve the attack detection and tolerance of our approach.

References

1. D. Arsenault, A. Sood, and Y. Huang. Secure, resilient computing clusters: Self-cleansing intrusion tolerance with hardware enforced security (scit/hes). In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 343–350, 2007.
2. P. Beaucamps, D. Reynaud, J.Y. Marion, and E. Filiol. On the impact of malware on internet voting. In *1st Luxembourg Day on security and reliability*, 2009.
3. A. R. Cavalli, A. M. Ortiz, G. Ouffoué, C. A. Sanchez, and F. Zaïdi. Design of a secure shield for internet and web-based services using software reflection. In *Web Services – ICWS 2018*. Springer International Publishing, 2018.
4. S. Estehghari and Y. Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking helios 2.0 as an example. In *Proceedings of the 2010 International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, pages 1–9. USENIX Association, 2010.
5. M. Ficco and M. Rak. Intrusion tolerant approach for denial of service attacks to web services. In *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing*, CCP '11, pages 285–292. IEEE Computer Society, 2011.
6. T. Furtado, E. Francesquini, N. Lago, and F. Kon. A middleware for reflective web service choreographies on the cloud. In *Proceedings of the 13th Workshop on Adaptive and Reflective Middleware*, ARM '14, pages 9:1–9:6. ACM, 2014.
7. M. Hennessy and H. Lin. Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.
8. B. B. Madan and K. S. Trivedi. Security modeling and quantification of intrusion tolerant systems using attack-response graph. *Journal of High Speed Networks*, 13(4):297–308, 2004.
9. Huu Nghia Nguyen. *Une Approche Symbolique pour la Vérification et le Test des Chorégraphies de Services*. PhD thesis, Université Paris-Sud, 2013.
10. D. O'Brien, R. Smith, T. Kappel, and C. Bitzer. Intrusion tolerance via network layer controls. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 90–96 vol.1, 2003.
11. S. Pavel, J. Noyé, P. Poizat, and J-C. Royer. A java implementation of a component model with explicit symbolic protocols. In *Software Composition*. Springer Berlin Heidelberg, 2005.

12. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*, 2007.
13. B. Sadegh and M. A. Azgomi. A new architecture for intrusion-tolerant web services based on design diversity techniques. *Journal of Information Systems and Telecommunication (JIST)*, Autumn 2015.
14. P. Sousa, A. Bessani, N. F. Neves, and R. Obelheiro. The forever service for fault/intrusion removal. In *Proceedings of the 2Nd Workshop on Recent Advances on Intrusiton-tolerant Systems*, WRAITS '08, pages 5:1–5:6. ACM, 2008.
15. P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 373–380, 2007.
16. A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saïdi, V. Stavridou, and T. ás E. Uribe. An architecture for an adaptive intrusion-tolerant server. In *Security Protocols, 10th International Workshop, Cambridge, UK, April 17-19, 2002, Revised Papers*, pages 158–178, 2002.
17. A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saïdi, V. Stavridou, and T. Uribe. An architecture for an adaptive intrusion-tolerant server. In *Security Protocols*, pages 158–178. Springer Berlin Heidelberg, 2004.
18. P. E. Verissimo, N. F. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch. Intrusion-tolerant middleware: the road to automatic security. *IEEE Security Privacy*, 4(4):54–62, 2006.
19. F. Wang, U. Raghavendra, and C. Killian. Analysis of techniques for building intrusion tolerant server systems. In *IEEE Military Communications Conference (MILCOM)*, volume 2, pages 729–734, 2003.