Sécurisation de services via des techniques de healing et d'encapsulations

Jarod Sue, Sébastien Salva

Sécurisation de services via des techniques de healing et d'encapsulations

Jarod Sue[1] et Sébastien Salva[1]
[1]LIMOS - UMR CNRS 6158, Clermont Auvergne University, UCA, France
jarod.sue@uca.fr, sebastien.salva@uca.fr

17 avril 2024

Résumé

Les compositions de services sont largement adoptées dans l'industrie depuis au moins une décennie. Force est de constater que l'intérêt pour leur développement est souvent inversement proportionnel à leur sécurisation, ce qui expose les développeurs et services à devenir les proies d'attaquants. Cet article présente une approche pour aider les entreprises à rendre résilient des services ayant diverses faiblesses (weaknesses) de sécurité grâce une nouvelle technique de guérison (software healing). De ce fait, cette approche semi-supervisée ne modifie pas le code original mais utilise des techniques d'encapsulation d'un service pour proposer un service mieux sécurisé. Celui-ci propose donc la définition d'opérateurs de healing de sécurité composé de 2 mitigations, une construite via un générateur de "prompt" pour IA génératives, et une seconde de repli se basant sur des solutions connues mais plus limitative quand à l'accès au service. Cet article propose également un algorithme de healing utilisant des ensembles de test et les opérateurs de healing. L'approche sera illustrée avec un exemple de services et de weakness. Finalement, sachant que le travail présenté est en cours d'étude, l'article présentera des perspectives de travail inhérentes à la confiance qui peut être apportée aux solutions de healing générées par IA et à l'évaluation de notre algorithme.
Mots clés: Composition de services ; Sécurisation de service ; Encapsulation

1 Introduction

Dans le paysage en constante évolution de l'informatique distribuée, les services web jouent un rôle fondamental en facilitant l'intégration et l'interaction entre les systèmes hétérogènes. Cependant, ceux-ci peuvent se retrouver vulnérables face à des défauts logiciels obtenus par le manque de tests induit par un manque de temps ou d'experts. Il n'est alors pas rare de détecter des bugs exposant les entreprises et les utilisateurs à des risques, obligeant alors à une maintenance longue et fastidieuse. De ce fait, la sécurisation des compositions d'APIs REST contre les exploitations malveillantes représente un défi pour l'industrie. Une solution possible à ce problème prend alors forme dans un concept appelé le software healing. Celui-ci consiste en la capacité d'un système informatique à détecter, diagnostiquer et corriger automatiquement ses défaillances et erreurs. Ce concept fait l'objet de recherche. [Vizcarrondo et al., 2012] présente un middleware entièrement distribué qui corrige les erreurs relié à la qualité du service. Il utilise une ontologie de healing qui va permettre le healing du service. [Shin, 2005] nous présente une architecture en deux couches, la couche service et la couche healing. Les deux sont déployés sur chacun des services. La couche service surveille chacun d'entre eux et détecte les anomalies. Si une anomalie est détectée, elle isole le composant en envoyant des plans de reconfiguration à chacun des autres services le temps du healing. La couche healing complète le service de trois façons différentes : réinitialisation, réinstallation, recherche de virus. Si le healing réussit, la couche service rétablit alors la communication vers le

service de la même façon qu'il l'avait isolé. [Rajagopalan and Jamjoom, 2015] propose de corriger des microservices. Pour cela, Shriram Rajagopalan et Hani Jamjoom proposent de ne déployer les nouvelles versions que sur une petite base d'utilisateurs. Leur proposition détecte alors les périodes prolongées de pertes de performances. Le healing proposée consiste alors en une restauration à une version précédente qui n'avais pas de problème de performance. [Dinkel et al., 2007] propose une réparation en boîte noire de services. Ils fournissent une couche d'abstraction entre le healing et le composant. La couche de healing récupère la connaissance collectée par les services et crée un diagramme d'activité en les utilisant. Ils analysent alors ce diagramme en utilisant de la connaissance métier pour créer des règles qui seront utilisées pour le healing. Le premier healing testé est forcément le redémarrage du service lorsqu'une erreur est détectée. Les healings proposées dans la littérature restent souvent limité aux problèmes de performance et avec des solutions simples : blocage de service, redémarrage, restauration à une version antérieure, etc. Dans ce papier, nous donnons une nouvelle approche pour le healing de services. Notre approche vise à améliorer la sécurité d'une composition de services. Cette approche ne modifie pas le code des services mais utilise des techniques d'encapsulation (introspection, service composite, etc) implémentant des solutions de mitigation aux problèmes de sécurité rencontrés par chacun des services. Notre algorithme prend en entrée un service et une suite de test $T(s)$ et génère une seconde suite de test de sécurité $T_w(s)$ pour chaque weakness w que nous voulons tester. Cette génération s'effectue par mutation des tests présent dans $T(s)$ grâce à l'approche présentée dans le papier [Salva and Sue, 2024]. Ces tests sont alors expérimentés sur notre service s. Si un test est un échec indiquant alors une faiblesse à corriger, nous créons un service corrigé s' qui viendra remplacer s. Nous explorons la possibilité de générer ces solutions d'encapsulation par intelligence artificielle avec la conception d'un générateur de Prompt (vue boite blanche ou vue boite noire). En résumé les contributions de ce papiers sont :

— la définition d'opérateur de healing,
— un algorithme permettant la correction de service grâce à des techniques d'encapsulation
— des mitigations obtenues par prompt

le papier est organisé comme suit, nous donnons les définitions et notations qui nous seront utiles dans la section 2, la génération de nos mitigations est présentée en section 3, notre approche, l'algorithme et un exemple sont présentés section 4,la section 5 résume nos contributions et décrit nos perspectives pour le futur.

2 définitions et notations

2.1 Contexte

Nous considérons que nous avons une composition de APIs RESTful déployée notée S. Cette application possède au moins une weakness que nous allons corriger avec notre algorithme. Elle possède sur chacun de ses services au moins deux ensembles de tests. Ainsi le contexte de notre application comprend :

— S est un ensemble de services,
— $T_r(s)$ est un ensemble de tests de non régression en isolation effectués sur le service s,
— $T_w(s)$ est un ensemble de suites de tests $Tw_1(s), Tw_2(s), ..., T_{w_n}(s)$ telle que $T_{w_i}(s)$ est une suite spécialisée pour détecter la weakness w_i

2.2 Hypothèses

— **Test en boîte noire :** Nous employons une perspective de boîte noire ainsi qu'un environnement isolé afin d'interagir avec le service en utilisant uniquement des requêtes et réponses HTTP. Ceux-ci sont nommés événements. Si le service dépend d'autres services, ceux-ci seront remplacés par des mocks.
— **Suite de test existante :** il existe une suite de tests de non-régression et une suite de test de sécurité (pouvant avoir été créé par notre papier précédent) permettant de tester en isolation

chaque service $s \in S$.

— **Contenu des événements :** des observateurs sont capable d'obtenir tous les événements reliés à s incluant leur contenu (pas de chiffrement). En particulier, les événements incluent des paramètres permettant d'identifier leurs sources et destinations. De plus, un événement peut être identifié en tant que requête ou réponse.

— **Healing en boîte grise :** pour le healing en utilisant une solution en boîte grise, nous considérons avoir accès aux informations de l'interface du service : méthode, URL, package, projet de développement de l'APIrest

2.3 Cas de test

Un cas de test est ici modélisé par un Input Output Transition System(IOTS) ayant une forme d'arbre et dont les états finaux expriment le verdict du test, pass, fail ou inconclusive. Chaque étape du test correspond à une transition entre deux états de l'IOTS $q \xrightarrow{e(\alpha),l} q'$ avec $e(\alpha)$ un événement et l un ensemble de labels, qui peut être vide. De plus, nous utilisons la notation θ sur les transitions afin de représenter l'absence de réaction d'un service en cours de test [**?**]. L'ensemble de labels permet d'exprimer facilement de la connaissance à propos d'un événement. Par exemple "crash" est utilisé quand un status HTTP 500 est reçu. Le label spécial "mock" identifie les événements effectués par d'autres services dépendants. Comme nous assumons que nous testons le service en isolation, les services dépendants sont remplacés par des composants simulés, aka mock. Un cas de test de type IOTS doit remplir un nombre restrictions afin d'éviter tout comportement non-déterministique. Ainsi, un cas de test doit autoriser un seul événement d'entrée à chaque étape. En référence à [**?**], si le cas de test respecte cette restriction, nous disons que celui-ci est *input restricted*. De plus, pour garder le contrôle lors du test et dans le contexte du test en isolation, un composant mock doit-être déterministe et retourner au maximum une réponse après avoir été invoqué avec le même événement. Pour la définition précise, vous pouvez voir [Salva and Sue, 2024].

Definition 1 (Verdict de test (Tretmans 2008)) *Un cas de test t est dit pass si celui-ci est exécuté de manière réussie sur un service s. Il est alors dénoté $s\ pass\ t$. Si l'exécution a raté, alors ce test est dénoté $s\ fail\ t$. Ces notations peuvent être étendues à une suite de tests T :*

$$s\ pass\ T \iff \forall t \in T : s\ pass\ t$$

2.4 Localisation d'erreur

La localisation d'une erreur nous permet d'identifier l'opérateur ou la ressource qui a déclenchée celle-ci. Elle est inférée à partir de l'erreur survenue dans un cas de test $t \in Tw_i(s)$. Une localisation est composée de l'URL, la méthode, le package de la partie du code d'où vient l'erreur. Ces variables peuvent ne pas être affectées. Cette localisation peut être inférée en analysant l'événement du cas de test qui a échoué avec des outils de débogage.

Definition 2 (Localisation d'erreur) *Une Localisation d'erreur $L(s,t)$ pour le service s suite au test de t est un tuple $(URL, methode, package, verbe)$ de variables qui peuvent être ne pas être affectées provenant de l'API du service s et qui permet de déterminer l'opération ou ressource qui a produit l'erreur. $T_{w_i}(s, L(s,t))$ représente alors tous les tests de sécurité pour la localisation L(s,t).*

2.5 Opérateur de healing

Afin de pouvoir sécuriser un service s possédant une weakness w, nous utilisons un opérateur de healing $H_w(L(s,t))$. Il est composé de 3 éléments, une localisation d'erreur ainsi que de 2

solutions de mitigations. La première des 2 solutions est constitué d'un prompt pré généré pour une IA génératrice qui est instancié grâce aux informations de la localisation.
Si cette solution s'avère ne pas permettre de corriger la faiblesse du service, la deuxième mitigation alors utilisée correspond à une solution pouvant aller de la restauration du service à une version antérieure à l'arrêt complet de celui-ci.

Definition 3 (Opérateur de Healing) *Un opérateur de Healing est un tuple $H_w(L(s,t)$ tel que :*
— *$L(s,t)$ est la localisation d'une erreur pour le service s suite au test avec t,*
— *$m_1 : s \rightarrow s$ est une fonction appliquant une mitigation constitué d'un prompt pour une IA génératrice permettant de créer un nouveau service s_1 tel que $s_1 = H_w.m_1(s)$,*
— *$m_2 : s \rightarrow s$ est une fonction appliquant une mitigation connue permettant de créer un nouveau service s_2 tel que $s_2 = H_w.m_2(s)$.*

3 Mitigations des opérateurs de healing

3.1 Mitigation par IA génératrice

La génération de la solution de mitigation par IA m_1 est effectuée à l'aide de la localisation et de la weakness détectée. Celle ci est constituée d'un prompt composé de variables à instancier venant d'une base de prompts dépendant du niveau d'accès au service (boîte grise ou boîte noire). Les variables utilisées sont :
— la localisation,
— la faiblesse à corriger,
— le type de mitigation voulu (filtre obligatoire pour la black box, choix pour la grey),
— les options voulues pour le code généré ainsi que les versions de ces options.
Voici un exemple de prompt de mitigation en boîte noire pour la weakness "token removal" :

```
You are a developper and an expert in software security
I have a TYPE web service
its path PATH is vulnerable
The verb used is VERB
the vulnerability is VULNERABILITY
can you create a new project which uses FIX
with [OPTIONS VERSIONS]
Which communicates via PROTOCOL to the vulnerable service
to correct the vulnerability before forwarding the request
```

Voici quelques exemples de valeurs pour les variables ci-dessus :

```
TYPE: spring-boot 3.2.3, jersey 3.0.12
PATH: http :// localhost :8080/ produit
VERB: POST
VULNERABILITY: "Authentification Failure not detected when the authorization token is deleted"
FIX: a filter
$[$OPTIONS VERSIONS$]$: jersey 3.0.12, spring-boot 3.2.3, jakarta ee 9.1, java 17
PROTOCOL: HTTP
```

```
Give me all the files required to make my project work in java VERSION with maven MVNVERSION

VERSION: 17
MVNVERSION: 3.8.3
```

La génération de code s'effectue dans un environnement semi-supervisée. Pour chaque weakness w, nous avons étudié la mitigation par prompt considérant 2 niveau d'accessibilité au service s, la boîte noire et la boîte grise. 2 prompts sont donc proposés pour chacune des weakness en fonction de ce

degré d'accessibilité. Pour ce papier, le générateur utilisé fut chatGPT. Ce type de génération permet alors d'obtenir une solution hautement personnalisable et permettant de couvrir un maximum d'architectures possible sans avoir à multiplier les prompts de départ. Une fois la solution générée par un prompt, nous obtenons un projet qui dépend du type de prompt souhaité (un nouveau projet avec un nouveau service dans le cas d'une boîte noire, pour la boîte grise, il est aussi possible de corriger le projet existant). Un nouveau service s_1 est alors créé.

3.2 Mitigation brute

Une mitigation m_2 est nécessaire si m_1 ne permet pas de corriger la faiblesse constatée. m_2 dépend de la weakness constatée ainsi que de la localisation de l'erreur. Plusieurs type de mitigations sont possibles et viennent de travaux différents sur le healing de service :

— l'arrêt total de s et la déviation des routes qui passaient par celui-ci,
— la restauration de s à une version antérieure si cette version pass les deux suites de tests,
— l'application d'un pattern de sécurité afin de créer une encapsulation différente de celles proposées par l'IA générative,
— la création d'une nouvelle instance de s ainsi que le routage vers celui-ci.

4 Healing de Service

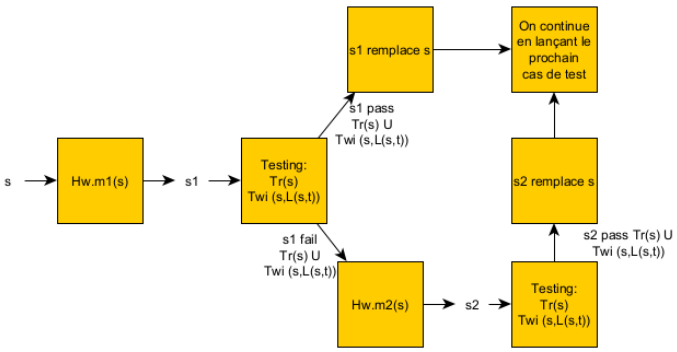

FIGURE 1 – Schéma du healing

Comme montré dans la Figure 1, La sécurisation de service que nous proposons s'effectue en plusieurs étapes :

— La première étape prend en entrée un service s qui possède une weakness w et dont un cas de test $t \in T_{w_i}(s)$ produit une erreur. Nous utilisons la première mitigation m_1 proposée par notre opérateur de healing $H_w(L(s,t))$ qui a été créé en inférant la localisation de l'erreur $L(s,t)$. Cela nous permet donc de créer notre nouveau service s_1
— La deuxième étape prend ce nouveau service s_1 en entrée. s_1 est expérimenté par $T_r(s)$ et $T_{w_i}(s, L(s,t))$. Si s_1 $fail$ $T_r(s) \bigcup T_{w_i}(s, L(s,t))$, alors nous passons à la troisième étape.
— La troisième étape produit un nouveau service s_2 en prenant en entrée notre service initial s et en appliquant sur celui-ci la mitigation générique m_2 incluse dans notre opérateur de healing $H_w(L(s,t))$.
— La quatrième étape prend le service qui permet la mitigation de l'erreur (s_1 si la mitigation par IA a été un succès, s_2 sinon) et corrige notre service s en l'encapsulant avec la mitigation.

Algorithme 1 : Algorithme de healing

input : Service s, Suite de test de non régression $T_r(s)$, Suite de test de sécurité $T_w(s)$, opérateur de Healing H_w
output : Service DC

foreach $T_{w_i}(s) \in T_w(s)$ **do**
  foreach $t \in T_{w_i}(s)$ **do**
    if $s\ fail\ t$ **then**
      $inférer\ L(s,t)\ depuis\ s\ et\ t$;
      $s_1 = Hw.m_1(s)$;
      $expérimenter\ T_r(s)\ et\ T_{w_i}(s, L(s,t))\ sur\ s_1$;
      if $s_1\ pass\ T_r(s) \bigcup T_{w_i}(s, L(s,t))$ **then**
        $s = s_1$;
      else
        $s_2 = Hw.m_2(s)$;
        $expérimenter\ T_r(s)\ et\ T_{w_i}(s, L(s,t))\ sur\ s_2$;
        $s = s_2$;
  $déployer\ s$;

4.1 Algorithme

Notre algorithme prend en entrée un service $s \in S$, un opérateur de healing H_w ainsi que les deux suites de tests : la suite de tests de non régression $T_r(s)$ et la suite de tests de sécurité $T_w(s)$. Il couvre tous les tests $t \in T_{w_i}(s)$ avec $T_{w_i}(s) \in T_w(s)$ et produit un nouveau service pour chaque weakness détectée. Si $s\ fail$ un des cas de test $t \in T_{w_i}(s)$, alors nous inférons la localisation de l'erreur depuis s et t. Une fois cette localisation obtenue, nous utilisons l'opérateur de healing H_w en lui donnant la localisation $L(s,t)$ afin de créer m_1 et m_2. m_1 est alors utilisé sur le service s permettant alors de créer le service s_1. $T_r(s)$ et $T_{w_i}(s, L(s,t))$ sont ensuite expérimentés sur s_1. Si $s_1\ pass\ T_r(s) \bigcup T_{w_i}(s, L(s,t))$, nous remplaçons s par s_1 et nous déployons s_1. Sinon, s_1 n'est pas retenu et nous passons alors à la deuxième solution de mitigation m_2. Cette solution crée un nouveau service s_2. Utilisant une mitigation connue, nous assumons que $s_2\ pass\ T_r(s) \bigcup T_{w_i}(s, L(s,t))$. s_2 est déployé en remplacement de s.

4.2 exemple

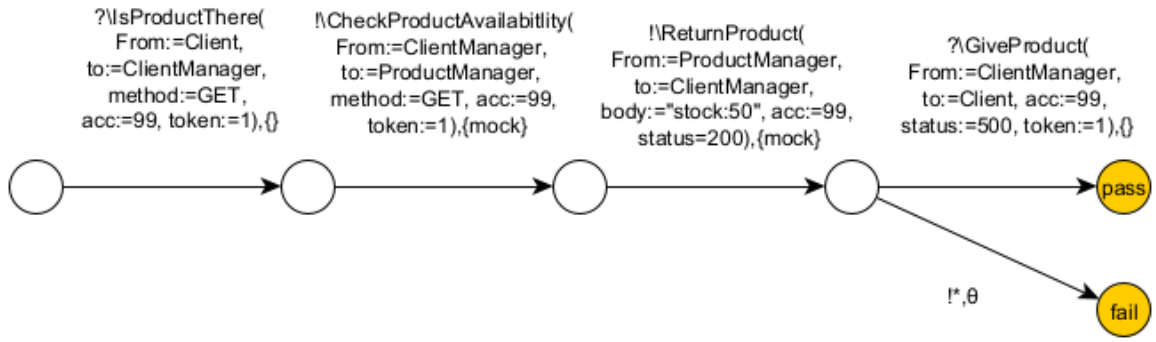

FIGURE 2 – exemple d'un cas de test de sécurité pour l'erreur "token manquant" avec uniquement le chemin pass

Afin de montrer le principe de l'algorithme, nous allons utiliser un exemple. L'exemple utilisé est illustré ici par la figure 2 et est effectué sur le service "ClientManager" dénoté s et la mitiga-

tion est effectuée en boîte grise. Comme nous pouvons le voir, le test consiste en une requête http effectuée sur s en utilisant le verbe GET. Cette requête déclenche une requête de s vers le service dépendant ProductManager qui va s'occuper de lui répondre. Afin de tester la résilience de notre service à la faiblesse du "token manquant", ProductManager lui renvoie une réponse où celui-ci retire le token d'authentification. Pour obtenir le verdict pass, notre cas de test doit alors détecter le manque du token et ainsi renvoyer une réponse à notre client qui consiste en un code status HTTP 500.Cependant, lors du test, ClientManager a répondu au client avec la réponse suivante :

```
!\GiveProduct(From:=ClientManager, to:= Client , body:="stock :50", acc:=99, status :=200, token:=1),{}
```

s n'a pas détecté le manque de token et le verdict de t est donc fail. Une requête a donc était créée et envoyée à une IA génératrice (ici ChatGPT 3.5) afin de lui demander une correction pour cette erreur en utilisant un filtre.

```
You are a developper and an expert in software security
I have a web service in spring-boot
its URL is http :// localhost :8080/ProductManager
its path /getStock is vulnerable
The verb used is GET
the vulnerability is token removal, the service need to detect the absence of the bearer token in the
    response from a downstream service
can you create a new project which uses a filter
with Spring-boot 2.5
Which communicates via HTTP to the vulnerable service
to correct the vulnerability before forwarding the request
```

Celui-ci a alors créé un nouveau projet contenant un filtre effectuant l'opération suivante :

```
package com.example.tokenfix;

import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component
public class TokenFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain
        filterChain) throws ServletException, IOException {
    // Check if the request is going to the vulnerable endpoint
    if (request.getRequestURI().equals("/getStock") && request.getMethod().equalsIgnoreCase("GET")) {
        // Check if token is present in the request
        String requestToken = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (requestToken == null || !requestToken.startsWith("Bearer")) {
            // Add token to the request
            String token = retrieveTokenFromSomewhere(); // Retrieve token from a secure place
            request.addHeader(HttpHeaders.AUTHORIZATION, "Bearer " + token);
            requestToken = "Bearer " + token; // Update the request token for comparison
        }

        // Continue with the request
        filterChain.doFilter(request, response);

        // Intercept the response
        if (response.getStatus() == HttpServletResponse.SC_OK) { // Only check if response is successful
            // Check if the response contains the token
```

```
                String responseToken = response.getHeader(HttpHeaders.AUTHORIZATION);
                if (responseToken == null || !responseToken.equals(requestToken)) {
                    // Handle the mismatched token in the response
                    handleMismatchedTokenInResponse(response);
                }
            }
        } else {
            filterChain.doFilter(request, response);
        }
    }

    private String retrieveTokenFromSomewhere() {
        // Implement logic to retrieve token from a secure place
        return "your_token_here";
    }

    private void handleMissingTokenInResponse(HttpServletResponse response) throws IOException {
        // Return a status 500 response indicating the missing token
        response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        response.getWriter().write("Token is missing in the response from the downstream service!");
        response.getWriter().flush();
    }
}
```

Ce nouveau projet envoie la requête à s après avoir appliqué le filtre et reçoit sa réponse. Le reste de l'algorithme expérimente alors $T_r(s)$ et $T_{w_i}(s)$ sur le nouveau projet créé qui encapsule le service s.

5 Conclusion

Dans ce papier, nous avons présenté une solution originale afin de sécuriser des API Restful via une technique de healing. Nous avons défini le principe d'un opérateur de healing basé sur 2 mitigations : une effectuée par une IA génératrice et une seconde de repli au cas où la première ne fonctionne pas. Nous avons ensuite présenté notre algorithme nous permettant de sécuriser un service. Utilisant une IA générative, notre travail futur cherchera avant tout à établir un niveau de confiance dans les solutions que celle-ci donne. Pour l'instant, ce niveau de confiance est déterminé par le succès des suites de test $T_r(s)$ et $T_{w_i}(s)$. De plus, notre travail futur explorera la possibilité de rassembler les mitigations des weakness afin d'éviter l'encapsulation du service à chaque erreur qui le compose produisant alors cascades d'encapsulations.

6 Bibliographie

Références

[Dinkel et al., 2007] Dinkel, M., Stesny, S., and Baumgarten, U. (2007). Interactive self-healing for black-box components in distributed embedded environments. In *Communication in Distributed Systems - 15. ITG/GI Symposium*, pages 1–12.

[Rajagopalan and Jamjoom, 2015] Rajagopalan, S. and Jamjoom, H. (2015). App–Bisect : Autonomous healing for Microservice-Based apps. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA. USENIX Association.

[Salva and Sue, 2024] Salva, S. and Sue, J. (2024). Security testing of restful apis with test case mutation. *arXiv preprint arXiv :2403.03701*.

[Shin, 2005] Shin, M. E. (2005). Self-healing components in robust software architecture for concurrent and distributed systems. *Science of Computer Programming*, 57(1) :27–44. System and Software Architectures.

[Vizcarrondo et al., 2012] Vizcarrondo, J., Aguilar, J., Exposito, E., and Subias, A. (2012). Armiscom : Autonomic reflective middleware for management service composition. In *2012 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–8.