FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MQTT Chaos Engineering for Self-Healing IoT Systems

Miguel Pereira Duarte

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Monteiro de Oliveira Restivo

Second Supervisor: João Pedro Dias

July 14, 2021

MQTT Chaos Engineering for Self-Healing IoT Systems

Miguel Pereira Duarte

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. João Reis
External Examiner: Prof. Ângelo Martins
Supervisor: Prof. André Restivo

July 14, 2021

Abstract

The Internet-of-Things (IoT) has shown substantial growth throughout the years, especially due to advancements in connectivity, computing power, and device affordability. Nowadays, any device might have a small computer inside it, enabling "smart" features. These devices are used to create a useful connection between the digital and physical world (through sensors and actuators), but failures or attacks might render them useless or compromise their users' safety and well-being.

Thus, it is increasingly important that they work correctly, since failures in their functioning may result in consequences ranging from slight discomfort to life-threatening situations. Fault-tolerant systems have been a subject of research that aims to tackle these issues. These systems have increased reliability due to being able to handle faults that may occur. More recently, approaches based on autonomic computing and self-healing have been introduced, providing valuable tools to handle and recover from failures in complex IoT systems.

To the best of our knowledge, what has been defined as best practices of security and dependability for both software and hardware development has not yet been fully applied to the IoT-scope. In particular, Chaos Engineering is currently being applied mainly to cloud infrastructure, without a lot of focus on IoT systems. Despite Fault Injection already having been applied in IoT, its focus is mostly on injecting faults in its hardware. As such, there is room for further development: the usage of Chaos Engineering in IoT systems can be useful to identify operational issues, as well as to find limitations in existing fault-tolerance mechanisms. Chaos Engineering may be paired with other techniques (*e.g.*, Fuzzing, Fault Injection) to further assess the dependability of existing systems.

This work focuses on the core problem of being able to exercise fault-tolerance mechanisms — more concretely, self-healing strategies — using Chaos Engineering. We hypothesise that the application of Chaos Engineering leads to systems with higher reliability since we can (a) exercise self-healing mechanisms, and (b) know when these mechanisms are not working correctly.

To find supporting evidence to our hypothesis, we proceed to implement a proof-of-concept of a Chaos injection solution. This solution consists of an extension to an MQTT Broker which enables the injection of faults via user-configured Chaos Engineering pipelines.

Using our proof-of-concept, we draft two experimental scenarios with a total of six experiments, mimicking real-world IoT systems and instrumenting them. In these experiments we focus on injecting faults into systems with and without Self-Healing capabilities, as well as comparing their behaviour to when faults are not injected to achieve a full comprehension of the systems' behaviour in each situation.

The conducted experiments supported our hypothesis by showing that it is possible to exercise and verify the correct behaviour of the Self-Healing mechanisms and strategies present in the systems under test.

Keywords: IoT, Internet of Things, Dependability, Fault Injection, Chaos Engineering, Self-Healing

Resumo

A *Internet-of-Things* (IoT) tem mostrado um crescimento substancial ao longo dos anos, especialmente devido a avanços na conetividade, no poder computacional, e menor custo por dispositivo. Hoje em dia, qualquer dispositivo pode conter um pequeno computador, que possibilite funcionalidades "inteligentes". Estes dispositivos são usados para criar uma útil ligação entre o mundo digital e físico (através de sensores e atuadores), mas falhas ou ataques poderão inutilizá-los ou até comprometer a segurança e bem-estar dos seus utilizadores.

Então, é cada vez mais importante que estes funcionem corretamente, pois falhas no seu funcionamento poderão resultar em consequências variadas, desde ligeiro desconforto a situações que apresentam risco de vida. Sistemas tolerantes a falhas (*fault-tolerant systems*) têm sido um tópico de investigação que almeja enfrentar estes desafios. Estes sistemas possuem uma maior *reliability* devido à sua capacidade de lidar com falhas que possam ocorrer. Mais recentemente, têm sido apresentadas abordagens baseadas em *autonomic computing* e *self-healing*, que providenciam valiosas ferramentas para a capidade de gestão e recuperação de falhas em sistemas IoT complexos.

Tanto quanto sabemos, o que tem sido definido como as melhores práticas de segurança e *dependability* para desenvolvimento de *software* e *hardware* ainda não foram totalmente aplicadas ao contexto IoT. De forma particular, *Chaos Engineering* tem neste momento um maior destaque na sua aplicação a infraestrutura na *cloud*, sem haver um grande foco em sistemas IoT. Apesar de *Fault Injection* já ter tido alguma aplicação a IoT, o seu foco é principalmente na injeção de falhas em *hardware*. Como tal, existe espaço para crescimento: a utilização de *Chaos Engineering* em sistemas IoT pode ser útil na identificação de questões operacionais, bem como para encontrar limitações em mecanismos de *fault-tolerance* existentes. *Chaos Engineering* poderá ser utilizado em conjunto com outras técnicas (*e.g.*, *Fuzzing*, *Fault Injection*) de modo a avaliar a *dependability* de sistemas existentes.

Esta dissertação foca-se no problema central de ser capaz de exercitar mecanismos de *fault-tolerance* — mais concretamente, estratégias de *self-healing* — com recurso a *Chaos Engineering*. A nossa hipótese é que a aplicação de *Chaos Engineering* leva a sistemas com maior *reliability*, dado que podemos (a) exercitar mecanismos de *self-healing*, e (b) saber quando estes mecanismos não estão a funcionar corretamente.

De modo a encontrar evidências que apoiem a nossa hipótese, procedemos a implementar uma solução prova-de-conceito que efetue injeção de *Chaos*. Esta solução consiste numa extensão a um *Broker MQTT* que possibilita a injeção de falhas através de configuração pelo utilizador, por meio de *Chaos Engineering pipelines*.

Utilizando a nossa prova-de-conceito, criamos dois cenários experimentais com um total de seis experiências, imitando sistemas IoT de mundo-real, e instrumentando-os. Nestas experiências focamo-nos na injeção de falhas em sistemas com e sem capacidades de *self-healing*, bem como na comparação do seu comportamento quando falhas não eram injetadas, de modo a atingir uma total compreensão do comportamento de cada sistema em cada situação.

As experiências realizadas apoiaram a nossa hipótese ao mostrar que é possível exercitar e verificar o comportamento correto de mecanismos e estratégias de *self-healing* presentes nos sistemas sob teste.

Keywords: IoT, Internet of Things, Dependability, Fault Injection, Chaos Engineering, Self-Healing

Acknowledgements

Firstly, I would like to thank my supervisors, André Restivo and João Pedro Dias, for the guidance and insight given through endless threads of back-and-forth discussion, which undoubtedly resulted in the best work I could produce. A special thanks to JP for all the off-topic conversations about all things geeky, and for helping me keep my motivation up throughout this journey.

To my family, especially my mother, father, and sister, a warm thank-you for all the kind words in dire moments. They were the ones that made sure I always had what I needed, be it a nice space to work in, or a much needed break after stressing out with some small issue over god-knows how long. I am certain that you will be glad to know that this journey has reached its end, even if only to hear me complain about something different for once.

To Catarina, my girlfriend, thank you for the "push to deliver" that I many times lacked, along with the affectionate smiles, long conversations and shared moments. I am ever grateful for the incentive to work, especially when it was the last thing I wanted to do.

To all the "help groups" that I am part of, spread along Messenger, Telegram and Discord: thank you for the moments we shared, of both virtue and vice, fun and frustration.

To Teresa, for the random calls, book recommendations and conversations, even if just to confirm that we were both on the same boat.

To Rui, my MIEIC "partner-in-crime", for the jokes, projects, and growth we shared these last few years.

To all those not mentioned, friends and otherwise, that also contributed to this journey. Thank you, for I would not be who I am or where I am today without you.

Miguel Duarte

"Websites that collect quotes are full of mistakes and never check original sources."

Randall Munroe, in XKCD 1942

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

CoAP	Constrained Application Protocol
DoS	Denial of Service
IoT	Internet of Things
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
SCE	Security Chaos Engineering

Chapter 1

Introduction

1.1 Context

The Internet-of-Things (IoT) has shown substantial growth throughout the years, especially due to advancements in connectivity, computing power, and device affordability [49]. Nowadays, any device might have a small computer inside it, enabling "smart" features. These devices are used to create a useful connection between the digital and physical world (through sensors and actuators), but failures or attacks might render them useless or even compromise their users' safety and well-being [47, 18].

With the wide-spreading of IoT devices across domains of application at an unprecedented scale, it becomes of the utmost importance to ensure that IoT systems are dependable. However, due to budget and time constraints, reliability of IoT systems is frequently disregarded. Some of the dependability risks for IoT systems are often a result of not receiving support via patches or updates, low or no use of encryption or an inability to deal with unexpected circumstances such as network failures [47, 50].

1.2 Motivation

As IoT devices and systems become more common in everyday life, it becomes increasingly important that they work correctly, specially given the criticality of some the devices that users wish to automate [48]. Failures in their functioning may result in consequences ranging from slight discomfort to life-threatening situations [20, 19]. For example, one can think of a simple system consisting of a thermometer feeding data into an IoT system, which outputs a signal to a room heater, configuring it with the desired intensity. In this case, certain failures in the thermometer (*i.e.*, sensor readings stuck at a low value) may result in the IoT system unknowingly causing a fire due to continuously turning on the room heater at the highest intensity. Furthermore, as another

motivational example one may consider the impact of malfunctioning IoT devices in critical systems such as healthcare. Wrong sensor readings may result in wrong patient diagnosis, incorrect treatment administration or even pose serious risk to the device users in the case of critical devices such as an automated insulin pump [9].

In order to tackle these issues, fault-tolerant systems have been a subject of research for some time [5, 59, 40]. These systems have increased reliability due to their ability to handle faults in their components using techniques such as redundancy (having several components that perform the same task such that if one fails, others can be used as backups).

More recently, approaches based on autonomic computing such as self-healing have been introduced by IBM [5], which have become increasingly valuable tools to handle failures and the complexity of IoT systems since applying these techniques increases the systems' dependability [19, 20, 17]. These concepts are further explored in Subsection 2.2.1 (p. 7).

Given the complexity and heterogeneity of Internet-of-Things systems [47], paired with the lack of technical knowledge of most of their users (which may be driven by the aforementioned affordability of IoT devices), it is becoming increasingly important that faults and issues of these devices are fixed or mitigated without the need for specific technical intervention on behalf of the users. Thus, not only is the application of autonomic computing concepts such as self-healing progressively relevant, but also the need to ensure that these capabilities are working correctly. This would further increase the confidence in the adequate functioning of complex IoT systems.

1.3 Problem

The problem that this work will focus upon can be decomposed into several parts:

- Analysis of the reliability of an IoT system when facing misoperation (*e.g.*, stuck-at values, values out-of-range);
- Analysis of the extent to which self-healing mechanisms may help in avoiding or mitigating the impact of faults the system faces;
- Usage of Chaos Engineering techniques to confirm that the self-healing mechanisms are working as expected, correctly recovering from injected faults.

To the best of our knowledge, what has been defined as best practices of security and dependability for both software and hardware development has not yet been fully applied to the IoT-scope. In particular, the usage of fault injection techniques and chaos engineering to assert the system's compliance with its expected behaviour may be further applied [15, 16].

Furthermore, Chaos Engineering has been applied to availability and reliability issues, having little exploration in its application to security aspects [41, 52].

Chapter 3 (p. 21) further expands on this problem, explaining it in greater detail, and defining its hypothesis and scope.

1.4 Goal

The core goal of this dissertation is to study the applicability of Chaos Engineering to Internet-of-Things systems, with a particular focus in those that pro-actively attempt to address issues as they occur, namely the ones that employ self-healing mechanisms and strategies.

To achieve this goal, we aim to be able to inject Chaos in IoT systems so that it is possible to analyse how these behave. Furthermore, it is also relevant to assess how recovery mechanisms such as self-healing strategies function in these conditions.

1.5 Document Structure

This document is composed by six chapters, structured as follows:

- Chapter 1 (p. 1), **Introduction**, which presents the problem under study, as well as its goal and motivation;
- Chapter 2 (p. 5), **Literature Review**, which overviews relevant literature to this work, describing the current state of the art of the areas relevant to it;
- Chapter 3 (p. 21), **Problem Statement**, which presents the problem under study and the solution proposed to solve it;
- Chapter 4 (p. 25), **Validation**, which focuses on the evaluation process to validate the hypothesis of this work, including description of the implemented solution, as well as the presentation and discussion of the results of the evaluation;
- Chapter 5 (p. 51), **Conclusions**, which concludes this dissertation by reflecting on this work's core aspects, presenting a summary of the developed work and highlighting some perspectives for future work.

Chapter 2

Literature Review

This chapter will focus on analyzing research that was previously done on topics of interest to this work. As such, this analysis will focus especially on technologies that are widely used in IoT, such as 802.11 and MQTT and failures that may be injected into devices, networks and systems using them, as well as contributions from related fields.

In Section 2.1 we overview the methodology for the literature review detailed in this chapter. Section 2.2 (p. 6) highlights some core concepts necessary for full comprehension of this work. Sections 2.3 (p. 9) to 2.6 (p. 16) focus on reviewing the results of the literature review for the concepts of Chaos Engineering, Fault Injection, Fuzzing and Security Attacks in IoT, respectively. Finally, Section 2.7 (p. 18) summarizes this chapter's findings.

2.1 Methodology

The searched content is based on conference papers and journal articles published in authentic electronic databases that are technically and scientifically reviewed by peers. These include IEEE *Xplore*, ACM Digital Library, Science Direct, Elsevier Journals and Springer Link. Grey Literature consisting of articles, technical reports, among others, also supported the search process.

The amount of relevant peer-reviewed literature was very low (a search in Google Scholar for `("internet of things" OR iot) AND "chaos engineering"` returned only 98 results[1]). However, this was not surprising as the concerns about IoT dependability have only recently come into focus. Since this topic is rapidly evolving, peer-reviewed articles are lagging behind (as is usually the case with new topics). As such, this has led to the decision to also include some grey literature into the state of the art. This literature resulted from using Google

[1] As of 2021-01-19

Scholar search engine, along with following some renowned researchers in the area. As such, it was possible to obtain magazines, reports, blog posts, and other web-items.

The bottom-up methodology to reach the found papers and articles can be summarized as follows:

- Initially, several queries were made for specific keywords;
- Then, these query terms were refined, according to related keywords found in initial results, with the aim of reaching broader topics;
- Finally, the references of the most relevant results were collected and searched for relevant publications, resulting in a snowballing of terms, topics, and papers that helped the expansion to related topics as well as the deepening of found literature for the topic under consideration;

This methodology is iterative by nature. As such, it can be repeated in order to further improve its results until they are satisfactory.

The used keywords were: `MQTT`, `802.11`, `WiFi`, `deauthentication`, `attack`, `fuzz`, `fuzzing`, `IoT`, `Internet of Things`, `Security`, `Chaos Engineering`, `self-healing`, `dependability`, `fault injection`, `failure injection`, `denial of service`, `DoS`, and `network`. This list is non-exhaustive, including all the used keywords which were then combined in several ways in querying for papers (such as including more keywords via `OR` or `AND` filters).

In order to achieve a complete overview of the cornerstones of this work, as well as the currently developed work in these areas, the found papers were analyzed with a *funnelling* scope: (1) Initially, the keywords were considered in application to broad areas or ones unrelated to this work; (2) Then, they were considered for the Internet of Things in specific; (3) Finally, whenever possible, they were considered for specific protocols or vulnerabilities.

Furthermore, for an overview of broad areas (*e.g.*, Chaos Engineering in general, not specifically applied to IoT or security) a different criterion was used whenever possible: review or survey publications were considered, as well as papers with over 100 citations or belonging to recent editions of relevant conferences or symposiums. Thus, it is possible to reach some insight of the area in general before diving into a specific subtopic, while not having to consider thousands of publications to understand the current *state-of-the-art* for it.

2.2 Background

This section highlights concepts that are necessary to define, describe or expand upon for the complete understanding of this work.

The first section focuses on Dependability and Autonomic Systems, defining several concepts of relevance in this area, such as *reliability* and *fault tolerance*. The second section focuses on

Reliability Measurement in IoT, highlighting some metrics that are currently being used for measuring reliability in IoT devices, as well as clarifying that there currently is not a solution capable of assessing the reliability of IoT infrastructure in its whole.

2.2.1 Dependability and Autonomic Systems

Ganek *et al*. [22] define several useful concepts for this work, namely *self-protection*, described as the ability for systems to *"anticipate, detect, identify and protect themselves from attacks from anywhere"*. These systems should be able to manage user access to computing resources in order to protect against unauthorized resource access, detect intrusions and report and prevent these types of activities as they occur. Additionally, this can be used to provide backup and recovery capabilities that are as secure as the original resource management systems. These systems may be built on top of several existing core security technologies (*e.g*., LDAP, SSL) and must provide capabilities for easier comprehension and handling of user identities in various contexts, removing this burden from administrators. Even though most of the definition provided by Ganek *et al*. may not apply specifically to the developed work due to the systems having different requirements and capabilities (*e.g*., a network of micro-controllers might not be powerful enough to run LDAP), the core concept of the system's ability to anticipate, detect, identify and protect itself from attacks is still relevant and fundamental for this work.

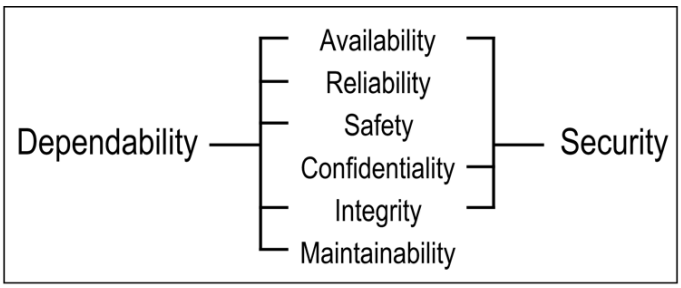

Figure 2.1: Relationship between dependability and security attributes, from [5].

Furthermore, Avizienis *et al*. [5] define *dependability*, *security* and other related concepts that are of relevance to this work:

Dependability Integrating concept that encompasses availability, reliability, safety, integrity and maintainability.

Availability Readiness for correct service.

Reliability Continuity of correct service.

Safety Absence of catastrophic consequences on the user(s) and the environment.

Integrity Absence of improper system alterations.

Maintainability Ability to undergo modifications and repairs.

Security A composite of the attributes of confidentiality, integrity and availability. Requires the concurrent existence of 1) availability for authorized actions only, 2) confidentiality, and 3) integrity with "improper" meaning "unauthorized".

Confidentiality The absence of unauthorized disclosure of information.

The relationship between dependability and security is summarized in Figure 2.1 (p. 7) in regards to their principal attributes. Furthermore, failures (deviation from correct behaviour), faults (cause for error, *e.g.*, bugs) and errors (deviation of external system state from the correct service state, caused by faults) are threats to the dependability and security of systems. There are several means to achieve the attributes of security and dependability, which can be grouped into four major categories [5]:

Fault prevention Prevent the occurrence or introduction of faults.

Fault tolerance Avoid service failures in the presence of faults. In other words: even when facing faults, the system should not fail.

Fault removal Reduce the number and severity of faults.

Fault forecasting Estimate the present number, future incidence, and likely consequences of faults.

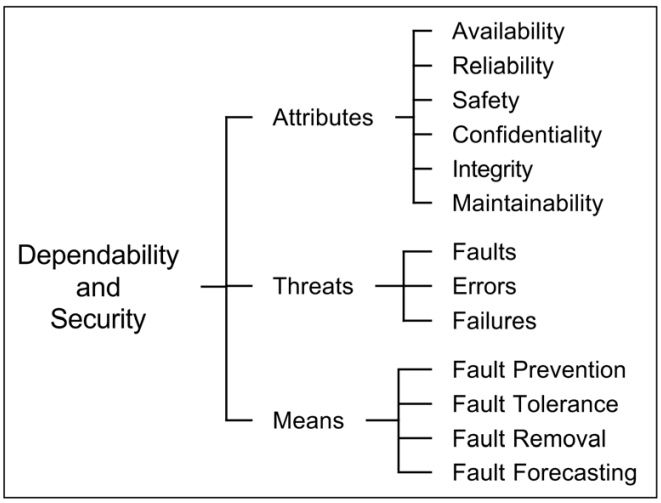

Figure 2.2: The dependability and security tree, from [5].

While fault prevention and fault tolerance provide the ability to deliver a system that can be trusted, fault removal and fault forecasting offer confidence in this ability by justifying the adequacy of functional and dependability and security specifications, as well as the likelihood of the system to meet these requirements [5].

Moreover, the attributes, threats to, and means to achieve dependability and security are summarized in Figure 2.2 (p. 8). Avizienis *et al.* also describe other basic system properties and concepts that are currently not relevant for the scope of this work.

2.2.2 Reliability Measurement in IoT

Moore *et al.* [31] provide an in-depth review of the state-of-the-art of reliability measurement in IoT. While the detailed evolution of the study and quantification of reliability in IoT is outside the scope of this work, the authors' work elicits several useful metrics for measuring reliability in IoT which may be used by this dissertation. Some are mentioned as standard reliability metrics, being used by several works to quantify the state of reliability in IoT devices. These include MTTF (Mean Time To Failure), MTTR (Mean Time To Repair), MTBF (Mean Time Between Failures), and Availability, Maintainability and Failure Rates. Despite these metrics being able to provide a view of how reliable an IoT device or set of devices is, the authors mention that further research should be done, for example, in regards to: extending these metrics to include network infrastructure and communication protocols, improving the metrics to allow the systems to predict and preempt failure, and further analysing the interplay between IoT device reliability and anomaly detection (researching the impact of anomalies on IoT device reliability).

Moreover, the research presented by Moore *et al.* [31] presents *"a clear gap in the knowledge and understanding of IoT: there is currently not a solution available capable of, in an end-to-end sense, assessing the reliability of IoT infrastructure"*, which supports the objective of this dissertation.

2.3 Chaos Engineering

In 2012, Martin Fowler [21] first introduced the concept of Phoenix Servers, which is closely related to the idea behind Chaos Engineering. The Principles of Chaos Engineering state that *"Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production"* [11]. Furthermore, modern software-based services are frequently implemented as distributed systems with complex behaviour and several failure states. According to Basiri *et al.*, Chaos Engineering is the practice of using experimentation to verify the reliability of these kinds of systems [6].

As such, Chaos Engineering presents itself as an appealing approach to test the dependability of systems.

2.3.1 Chaos Engineering for Fault Detection

Netflix's Chaos Monkey is one of the main driving forces in the popularization of Chaos Engineering. *"Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures"*. This solution randomly terminates virtual machines and containers that run in a production environment during working hours (so that engineers can respond quickly if a service fails).

The idea behind this is that frequently exposing engineers to failures motivates them to develop resilient services [33].

This technique has proved successful since Netflix engineers now design their services considering potential failures (and recovery from these) from the get-go. Basiri *et al*. [6] point out that despite the concepts of Chaos Engineering not being fundamentally new, the practical application of these concepts to software systems is still in its early stages. They mention that the increasing complexity of software systems will be a driving force for the increasing adoption of empirical approaches to achieve system availability.

In fact, this area has since grown, as expected, as Rosenthal *et al*. [42] highlight by mentioning that Chaos Engineering practitioners from over 20 organizations (such as Microsoft, Google, Amazon, Uber, or the University of California) were present at a recent event about Chaos Engineering. In their own words, *"It's Not Just for Netflix"*. Chaos Engineering is now even extensively applied to companies that are not natively digital, like healthcare or large financial institutions.

Jernberg *et al*. [26] introduced Chaos Engineering in a group of companies in order to improve their systems' resilience. As a result, the authors contribute not only with a comprehensive synthesis of Chaos Engineering literature and tools but also with guidelines for the successful introduction of Chaos Engineering in a company. This research concludes that applying Chaos Engineering principles and tooling to a company is feasible and successfully improved the considered systems' resilience, due to finding a set of improvement opportunities. The presented overview of Chaos Engineering tools highlights that the vast majority of currently existing tooling for Chaos Engineering is aimed at testing infrastructure managed by cloud providers, with only some tools being able to test specific applications of specific languages and frameworks (*e.g*., Java, Spring Boot). Furthermore, the authors categorize the attacks that the tools perform into five categories, mentioning that only one tool mentions the ability to perform security attacks (ChaoSlingr [35]). The authors present an implementation framework to apply Chaos Engineering to a company which, despite interesting, is outside of the scope of this work.

2.3.2 Chaos Engineering applied to Security

Security Chaos Engineering (SCE) was a term introduced by Aaron Rinehart and Charles Nwatu [34] to describe the application of Chaos Engineering to tackle not only reliability issues but also security ones. They mention that *"(Security) Chaos Experiments are foundationally rooted in the scientific method, in that they seek not to validate what is already known to be true or already known to be false, rather they are focused on deriving new insights about the current state"*, and that *"Security Chaos Engineering is the discipline of instrumentation, identification, and remediation of failure within security controls through proactive experimentation to build confidence in the system's ability to defend against malicious conditions in production"*. Security Chaos Engineering provides a methodology for the experimentation of distributed systems' security with the aim of building confidence in the system's ability to withstand malicious conditions.

According to Rinehart [41], the motivation for this application of Chaos Engineering to the field of cybersecurity arises from the need to think differently about information security due to a

generalized movement toward complex distributed systems with which security struggles to keep pace, as these systems are becoming impossible for the human mind to mentally model. Furthermore, Security Chaos Engineering allows teams to safely discover system weaknesses before they disrupt the systems themselves.

Furthermore, Rinehart [41] points out that ChaoSlingr is proof that Chaos Engineering can be applied to cybersecurity, and their experience of utilizing ChaoSlingr at UnitedHealth Group demonstrates that there is value in this approach since it has the potential reveal valuable, objective information about the operation of security controls, allowing organizations to optimize their security budgets further.

ChaoSlingr is a Security Chaos Engineering Tool initially published in September 2017 that has since stopped being actively maintained. It was a Proof of Concept (PoC) focused on experimentation on AWS Infrastructure to identify security weaknesses in it [35]. ChaoSlingr proactively introduces known security failures through a set of experiments in order to determine the effectiveness of the implemented security measures [41].

In the contextualization of their developed tool, *CloudStrike*, Torkura *et al*. [52] provide an overview of current chaos engineering techniques, with particular emphasis in security. They mention that *state-of-the-art* chaos engineering techniques inject faults into software systems to detect availability issues which may be resolved to improve system resilience via resiliency patterns (*e.g*., timeouts, retries). While these resiliency strategies provide useful and clear feedback about system behaviour for chaos engineering experiments, they are not designed to improve security, instead tackling availability attributes. Additionally, in their summary of notable chaos engineering tools, the authors highlight that despite there being a diversity of concrete applications of chaos engineering concepts, most tools focus on issues related to non-security availability attributes, mentioning only CloudStrike (the tool proposed and developed by the authors) and ChaoSlingr [35] as being focused on security resiliency attributes.

CloudStrike is a tool that is designed for multi-cloud security experimentation. It leverages chaos engineering principles, focusing on security via fault injection, which may impact confidentiality, integrity and availability of the tested cloud resources. This tool changes cloud infrastructure configurations to insecure settings to assess if these changes are detected by configurable alerts (thus detecting potential security misconfiguration that may happen in a real scenario). While similar to ChaoSlingr, it implements more fault injection rules (over 20 compared to ChaoSlingr's 3), covering a more expansive fault space. [52, 51].

Despite the findings mentioned above, Security Chaos Engineering is still a novel term. Searching for `"security chaos engineering"` in Google Scholar and Web of Science (all databases) resulted in only 7 and 2 results, respectively[2].

[2] As of 2020-01-23

2.4 Fault Injection

Fault Injection is the act of exercising failures to measure their impact, with the aim of ensuring resilience [8]. Fault Injection will be a core topic to focus on for this work, due to how helpful this methodology may be in finding faults in systems. Software Fault Injection (SFI) is a method for anticipating worst-case scenarios caused by faulty software via the deliberate injection of software faults [32].

There is no clear distinction between the terms "Fault Injection" and "Failure Injection". As such, "Fault Injection" will be used henceforth to refer to this concept.

Natella *et al.* [32] point out that software faults are a severe problem for current and future systems, namely due to their continuous growth in complexity. Their survey analyses several techniques and tools for software fault injection, while pointing out that several challenging issues are still without an answer despite the evolution of the approaches. The authors list three core research directions and open issues for Software Fault Injection:

- **Usability and Dependability Benchmarking:** Fault Injection techniques are difficult to apply to software, *e.g.*, when compared to hardware bit-flip faults. Improvements in the usability of fault injection are necessary for the widespread adoption of this technique to be feasible, as otherwise the industry will hesitate to adopt dependability benchmarking. This usability improvement is also fundamental in making dependability benchmarking viable for real software projects since the current costs and know-how required for its application are deterrents for its widespread adoption.

- **Challenge of Software Fault Tolerance:** Despite the recognition of software faults as a significant threat for modern systems, tolerating these faults is a challenging issue. Software faults affect a program's logic, requiring logic redundancy to be tolerated, which is non-trivial in many cases. Even knowing that software is defective, one never knows exactly where the defects are, when they will appear or which consequences they may have. These challenges are an excellent opportunity for applying Software Fault Injection.

- **Evolution of Fault Models:** Fault models will have to keep up with the evolution of software technologies and new development paradigms. For example, the growth of distributed and concurrent systems requires new types of fault injection to be considered, such as the injection of multiple faults, which is still at an early stage. There is also a need of further research into the interaction between Software Fault Injection and *software security* since SFI can be used as a simulation of security vulnerabilities, given that the injected faults may, as an example, cause unauthorized access to confidential data. Given that software is frequently exposed to security attacks, being able to emulate security vulnerabilities would enable the creation of better countermeasures.

This last topic (the interaction between Fault Injection and software security) is of particular relevance to this work, as Fault Injection will be applied to IoT systems in order to make them more secure.

Yoneyama *et al*. [57] performed model-based fault injection on MQTT using Modbat [4]. Their approach mimics unstable network environments by using fault injection to simulate network errors and delays. The introduction of packet forwarders in the connections between the clients and the server allows the software-based simulation of unstable TCP network environments during test execution, without any modification to the system under test. Additionally, this technique enables the observation of the effect of lost packets on MQTT, being the authors able to confirm that under higher QoS (Quality of Service) settings the data is retransmitted until delivered.

Kazemi *et al*. [27] highlight the importance of carefully considering security and privacy in the development of IoT devices. These must be protected against software and network-based attacks, as well as hardware-based attacks. Their work goes into more detail on types of physical fault attacks, which is outside of the scope of this work. Nevertheless, their research applies fault injection in the form of physical hardware-based fault attacks with the aim of helping improve the security and robustness of embedded systems, which aligns with this work's aim to also improve the research on the security of IoT devices.

Despite the applicability of fault injection to IoT, the found work focuses mostly on hardware fault injection via physical interaction with the devices [27, 58, 25], rather than interfering with their communications and network, as this work aims to do.

2.5 Fuzzing

At a high level, fuzzing refers to repeatedly executing a program with inputs that are generated and may be syntactically or semantically malformed [30]. Fuzzing is a simple yet effective approach to discover software bugs utilizing randomly generated inputs [36].

Moreover, Fuzzing is a prevalent software testing technique, being its popularity a result of its simplicity, ease of deployment and proven effectiveness in discovering real-world software vulnerabilities [30].

Li *et al*. [28] provide a thorough overview of the current state of fuzzing, as well as new trends and challenges this area faces. They mention that as services are being deployed on a network and client applications communicate with these servers via network protocols, security testing of network protocols becomes a significant concern. Security problems that occur in these protocols may result in much more severe damage than local applications, as denial of service and leakage of private information, among others, are considerable threats. Some research is being done in this area, resulting in several tools and proposals that the authors highlight. Finally, the authors mention that fuzzing *"is currently the most effective and efficient vulnerability discovery solution"*, which supports this work's objective of using it to investigate potential security issues.

In their overview of the state of the art of fuzzing, Manès *et al*. [30] also mention several network protocol fuzzers which take protocol specifications from the user. These fuzzers are mentioned as using a predefined model for fuzzing, being referred to as model-based fuzzers.

The research for related work uncovered several tools for which not much publication was found. Nevertheless, these tools have much potential to be applied to this work due to being aimed at fuzzing network protocols, some having already been used for real-world scenarios (*e.g.*, Sulley being used to fuzz SCADA Protocols, shown in Defcon [14] or to find previously known vulnerabilities, validating fuzzing as an approach to find security issues [54]). These are compared among themselves and with some popular general-purpose fuzzers mentioned in some of the literature in Table 2.1 (p. 15) and also briefly described below:

- **AFL** (American Fuzzy Lop) [23] - A coverage-based grey-box fuzzer that employs an instrumentation-guided genetic algorithm to fuzz its targets efficiently. AFL is aimed at fuzzing binary targets, also being able to instrument programs when the source-code is available. It has already helped detect several vulnerabilities and is actively maintained, as well as being the base for several forks (*e.g.*, FIRM-AFL [60]);
- **BED** [1] - Stands for Bruteforce Exploit Detector. A tool designed to fuzz several protocols (*e.g.*, FTP, SMTP, HTTP, POP, IMAP) in order to find buffer overflow or format string vulnerabilities;
- **Doona** [56, 55] - A fork of BED with the same objective, created to add more features to BED since its development was halted. Renamed due to the addition of *"a significant enough number of features/changes"*;
- **Sulley** [38, 14, 54] - A fuzzing framework that (unlike other fuzzers) not only focuses on data generation but also includes other aspects *"a modern fuzzer should provide"*. Sulley watches the network and maintains records, instrumenting and monitoring the health of its target, being able to detect and categorize detected faults, run in parallel, among other features. Sulley is aimed at fuzzing network protocols;
- **boofuzz** [39] - A fork of Sulley that provides several bug fixes and aims for extensibility. Also targets network protocols, describing itself as *"Network Protocol Fuzzing for Humans"*. It was created with the objective of being able to fuzz anything, as well as due to the lack of maintenance of Sulley.

Some of the found tools are quite old (*e.g.*, **BED**) and some are unmaintained (*e.g.*, **Sulley**), showing how fuzzing has recently focused mostly on honing some specific tools (*e.g.*, **AFL**), as well as custom-made fuzzers for specific purposes (*e.g.*, **boofuzz**, **Doona**).

2.5.1 Fuzzing Network Protocols

Gorbunov *et al.* [24] introduce AutoFuzz, which learns protocol implementations by observing communications between clients and servers and building a Finite State Automaton to represent the protocol. Then, it can fuzz client or server protocol implementations using the constructed automaton. The authors apply AutoFuzz to several FTP implementations, which resulted in the

Table 2.1: Fuzzing tools comparison.

Name	Language	Scope	Actively maintained?	Creation date
BED [1]	Perl	Network protocols	no	Before 2011
Doona [56, 55] (BED fork)	Perl	Network protocols	no	2012
Sulley [38, 14]	Python	Network protocols, network and process watching	no	2006
boofuzz [39] (Sulley fork)	Python	Network protocols, network and process watching	yes	2016
Peach [44]	Python	General Purpose	no	2016
AFL (American Fuzzy Lop) [23]	C	General Purpose Guided Brute-force	yes	2013

discovery and confirmation of vulnerabilities and unexpected behaviours, including arbitrary remote code execution prior to authentication without the need to write shellcode. This research shows the potential of the application of fuzzing to network protocols.

2.5.2 Fuzzing Practices for IoT

Chen *et al*. [10] present an automatic fuzzing framework, IoTFuzzer. It aims to find memory corruption vulnerabilities in IoT devices without access to their firmware, unlike other fuzzers which require access to these images. The authors' approach was to focus on the mobile applications through which IoT devices are frequently controlled. IoTFuzzer was thus built as a protocol-guided fuzzer, using information that the app contains about the protocol without reverse-engineering the protocol or explicitly recovering specific knowledge on the protocol from the app. This tool instead focuses on performing a dynamic analysis of the app, identifying how it generates the messages when communicating with the target device, mutating the message's content in run-time, and producing test cases for probing the firmware under study. IoTFuzzer was evaluated on 17 real-world IoT devices - 15 security-critical vulnerabilities found, 8 of these not having been previously reported. It was also compared to other network fuzzers such as BED and Sulley. Finally, the authors highlight that while impressive progress is being made in vulnerability research for embedded systems, the approaches employed are often specific to certain models, vendors or products, instead of being automatic, systematic or generic, like IoTFuzzer.

Zheng *et al*. [60] present FIRM-AFL, *"the first high-throughput greybox fuzzer for IoT firmware"*. Coverage-based greybox fuzzing has been shown to be an effective and efficient way to find vulnerabilities in real-world programs. However, the application of greybox fuzzing to IoT firmware faces two main challenges: state-of-the-art greybox fuzzers such as AFL are unable to run many IoT programs due to hardware-specific restrictions or dependencies; and solutions for this first issue have very low fuzzing throughput due to using full-system emulation. As such, FIRM-AFL tackles both these problems by presenting a novel technique which the authors refer to as *"augmented process emulation"*. This technique combines system and user-mode emulation, taking

advantage of the high compatibility of system-mode emulation, as well as the higher throughput of user-mode emulation. FIRM-AFL is built on top of AFL [23], modifying it to adapt it to solve the mentioned problems while taking advantage of its tried-and-tested workflow (*e.g.*, replacing its user-mode QEMU with augmented process emulation). The authors perform a series of tests on several aspects, showing that this novel technique does incur a large performance loss in performed benchmarks, and its throughput outperforms system-mode emulation based fuzzing by 8.2 times. Overall, FIRM-AFL is 3.6 to over ten times faster than full-system emulation in finding crashes for the tested device firmwares, being able to find 1-day vulnerabilities much faster than full-system emulation and finding two new vulnerabilities (0-days).

Liu *et al.* [29] apply fuzzing to Narrowband Internet of Things (NB-IoT) protocols to test them. These protocols have complex frame structures, which makes it challenging to test them manually. As such, they are great candidates for fuzz testing. The authors present HFuzz, a developed automatic hierarchy-aware fuzzing framework that takes advantage of format information on NB-IoT network protocols to outperform traditional fuzzing tools (AFL and Peach, in this case).

2.6 Security Attacks in IoT

This section focuses on specific attacks performed on IoT devices or which would be applicable to IoT devices. These attacks include WiFi deauthentication and MQTT Denial of Service attacks, among others.

Bellardo *et al.* [7] provide an experimental analysis of 802.11-specific denial-of-service (DoS) attacks by targeting its MAC protocol. By taking advantage of MAC-level spoofing, the authors explore several techniques such as deauthentication (*cf.* Figure 2.3, p. 17) and disassociation attacks, providing a theoretical context of how they work and empirical validation of elicited vulnerabilities and their impact in real networks. Despite their widespread adoption, 802.11-based networks are shown as having a poor availability aspect, since there are several easily achieved methods to launch DoS attacks on a network. Although several countermeasures are highlighted, these fall out of scope for this work.

Sun *et al.* [50] analyze a set of common smart home appliances from a potential attacker's perspective, using side-channel attacks to fingerprint these IoT devices. In this work, they claim that *"modern day smart homes are not much different than a castle of glass: an intimidating, towering paragon of defense, but indefensible with its inner workings exposed and fragile walls easily cracked"*, as the data that the IoT devices leak in their operation compromise their location's security. Furthermore, the authors use WiFi deauthentication attacks to assess the impact of a loss of connectivity on the devices' functionality, which frequently renders them useless due to the lack of fallback physical controls on the devices.

Andy *et al.* [3] explore MQTT's lack of security mechanisms with two types of attacks: using Shodan to find public and unsecured MQTT servers, to then exfiltrate potentially private data by subscribing to all topics, perform DoS attacks by flooding the broker with spam data or inject incorrect data; and attacking MQTT brokers in a local network by sniffing and modifying packets

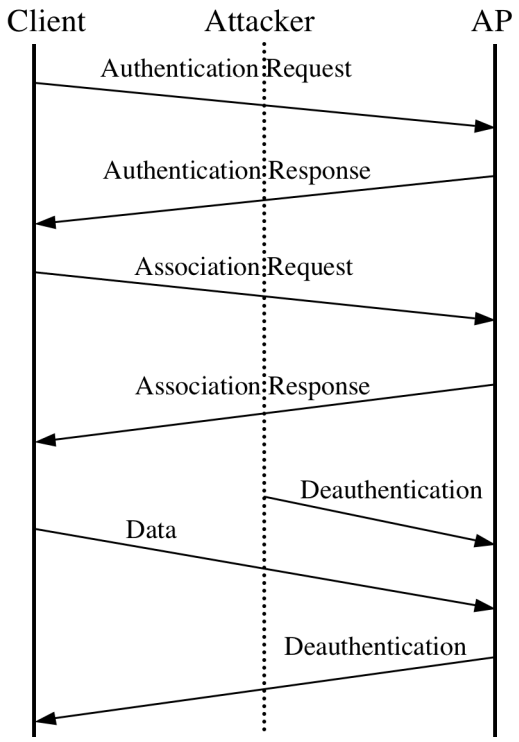

Figure 2.3: Graphical depiction of WiFi deauthentication attack, from [7].

to compromise privacy, data integrity and the MQTT authentication mechanism, which is deemed to be lackluster, especially in its default settings. The authors conclude that MQTT is vulnerable to the aforementioned attacks and mention several mitigations with regards to improving security by implementing cryptographic countermeasures. However, this cryptographic research is out of scope for this work.

Vaccari *et al*. [53] present SlowITe, a Slow DoS Attack against IoT systems, specifically targeting MQTT services. This attack exploits a flaw in the MQTT protocol in which the client can set the Keep-Alive parameter (which denotes the time that a connection should be kept alive for) of their connection to the server to an arbitrary value. It is possible to set this parameter to its maximum value, which results in the connection being kept alive for over 27 hours without sending any data to the listening daemon other than the initial 30 bytes of the CONNECT packet. As such, it is possible to launch a low-rate DoS attack that seizes all of the broker's available connections by repeating this process for several hundred or thousands of connections, with a very low resource cost for the attacker. Tests were performed on real networks against several different MQTT servers (Eclipse Mosquitto MQTT, ActiveMQ, HiveMQ, and VerneMQ), using both unencrypted and encrypted communication. The connection closure was tested for the maximum value of the Keep-Alive parameter (65535), which resulted in the connection remaining open for the expected period of about 27 hours, without data transmission from the client. Given the simplicity of replicating this attack for several connections (up to the MQTT broker's maximum number

of connections), as well as the ability to reach a DoS state in the different services (as shown in further tests), the effectiveness of this simple attack in disabling MQTT servers is proven.

Since MQTT and AMQP are both asynchronous publish/subscribe protocols used in IoT, research upon these protocols was focused more on MQTT since it yielded more results (29100 for `"MQTT"` and 8600 for `"AMQP"`[3]).

2.7 Summary

This chapter performed a literature review focused on the three main areas that this work aims to build upon: Chaos Engineering, Fault Injection and Fuzzing and their applications to IoT and security. Additionally, we researched about attacks and security vulnerabilities relevant to IoT due to affecting IoT devices and protocols.

Chaos Engineering presents itself as an appealing approach to test the dependability of systems. In 2016, despite Chaos Engineering concepts not being new, their practical application was still in its early stages [6]. Since then, the area has grown and even seen extensive application to areas that are not natively digital [42]. Despite this, the performed literature review highlighted that current investigation in Chaos Engineering focuses mostly on cloud systems, being the found tooling primarily aimed at cloud providers such as AWS. Furthermore, most of the found work on Chaos Engineering focused primarily on improving the system's availability, rarely focusing on security aspects [52, 26]. Only two tools were found which claimed to perform Security Chaos Engineering: ChaoSlingr [35] and CloudStrike [52, 51]. Additionally, Security Chaos Engineering may be applicable to IoT due to fitting the description of systems that are becoming so complex that they are impossible for the human mind to mentally model [41].

The review of Fault Injection showed that it is primarily considered for testing a system for flaws and is frequently mentioned alongside Chaos Engineering. According to Natella *et al.* [32], there are three core research directions and open issues for Software Fault Injection: Usability and Dependability benchmarking; the Challenge of Software Fault Tolerance; and the Evolution of Fault Models. These are explored in further detail in Section 2.4 (p. 12). Furthermore, the literature recognizes the difficulty of applying Fault Injection to software (*e.g.*, when compared to hardware) and highlights the importance of improving the usability of fault injection methods. Moreover, it recognizes the potential in the interaction between Software Fault Injection and software security, which should be further researched [32]. Some work has been done in applying fault injection to IoT to simulate network errors and delays [57]. However, found work focuses mostly on hardware fault injection [27, 25, 58].

The literature review of fuzzing highlighted that it is considered a simple technique for testing systems and has proven effectiveness in discovering real-world vulnerabilities [30]. Some authors mention fuzzing as an effective and efficient method for vulnerability discovery [28], as well as highlighting the security testing of network protocols as a significant concern. Model-based fuzzing may be an interesting technique to apply for fuzzing network protocols [30], in a grey or

[3] As of 2021-01-26, in Google Scholar search engine

white-box approach. This research also found several tools that may be useful for this work in grey literature [1, 56, 55, 38, 14, 54, 39]. Furthermore, we found some work done in the application of fuzzing to IoT [10, 29, 60] which showed that fuzzers made for or adapted to fuzz specific IoT targets (*e.g.*, protocols, devices or improved emulation) outperformed popular general-purpose fuzzers when applied to the same targets.

Finally, in this literature review, we researched about vulnerabilities found in IoT systems or that would apply to them, which will help determine relevant attacks to perform and methodologies with which to find vulnerabilities. This research resulted in finding some issues which may still be exploitable and thus might be useful for this work (*e.g.*, WiFi deauthentication attacks [7, 50] or MQTT Denial of Service [53]).

In sum, after the literature review performed in this chapter, we found that Chaos Engineering's applications are currently focused mainly on cloud infrastructure and improving a system's availability, without much focus in IoT systems or security aspects of dependability. Moreover, although Fault Injection has seen some application in security research, the found work is more focused on hardware and physical fault injection instead of on network fault injection. Furthermore, Fuzzing has been proven to be effective in finding security vulnerabilities, already having some applications to IoT, which shows it may be applicable to the problem under study. Finally, we already found some security attacks performed on and applicable to IoT networks and devices which may be a good starting point for this research.

As such, several areas and shortcomings were found. These will be analysed in further detail in Chapter 3 (p. 21).

Chapter 3

Problem Statement

3.1 Problem under Study

The Internet-of-Things (IoT) has shown substantial growth throughout the years, primarily due to advancements in connectivity, computing power, and device affordability. Nowadays, any device might have a small computer inside it, enabling "smart" features. These devices may create a useful connection between the digital and physical world (through sensors and actuators), but failures or attacks might render them useless or compromise their users' safety and well-being [47]. Furthermore, since IoT employs network architectures very similar to traditional ones for communication between devices, it also inherits the flaws of such traditional networks [13].

The literature review presented in Chapter 2 (p. 5) focused on three areas that are the basis of this work: Chaos Engineering, Fault Injection and Fuzzing. Likewise, we researched some attacks and security vulnerabilities relevant to IoT (focusing on the exploration of IoT devices and networks or protocols commonly used in IoT).

This review showed that current investigation in Chaos Engineering focuses mostly on cloud systems, being the found tooling even aimed at cloud providers such as AWS. Furthermore, most of the found work on Chaos Engineering focused primarily on improving the system's availability, rarely focusing on security aspects [52, 26]. Additionally, Security Chaos Engineering may also be applicable to IoT since it fits the description of systems that are becoming so complex that they are impossible for the human mind to mentally model [41].

The review of Fault Injection showed that it is primarily considered for testing a system for flaws and is frequently mentioned alongside Chaos Engineering. The literature recognizes the difficulty of applying Fault Injection to software (*e.g.*, when compared to hardware) and highlights the importance of improving the usability of fault injection methods. Moreover, it recognizes the potential in the interaction between Software Fault Injection and software security, which should be further researched [32]. Some work has been done in applying fault injection to IoT to simulate network errors and delays [57]. However, found work focuses mostly on hardware fault injection [27, 25, 58].

The literature review of fuzzing highlighted that it is considered a simple technique for testing systems and has proven effectiveness in discovering real-world vulnerabilities [30]. Some authors mention fuzzing as an effective and efficient method for vulnerability discovery [28], as well as highlighting the security testing of network protocols as a significant concern. This research also found several tools that may be useful for this work, despite not having appearing in scientific literature [1, 56, 55, 38, 14, 39]. Furthermore, we found some work done in the application of fuzzing to IoT [10, 29, 60] which showed that fuzzers made for or adapted to fuzz specific IoT targets (*e.g.*, protocols, devices or improved emulation) outperformed popular general-purpose fuzzers.

Complimentary, we also researched about vulnerabilities found in IoT systems or that would apply to them, which will help determine relevant attacks to perform and methodologies with which to find vulnerabilities. This research resulted in finding some issues which may still be exploitable and thus might be useful for this work (*e.g.*, WiFi deauthentication attacks [7, 50] or MQTT Denial of Service [53]).

In sum, after the literature review in Chapter 2 (p. 5), we found that Chaos Engineering's applications are currently focused mainly on cloud infrastructure and improving a system's availability, without much focus in IoT systems or security aspects of dependability. Moreover, although Fault Injection has seen some application in security research, the found work is more focused on hardware and physical fault injection instead of on network fault injection. Furthermore, Fuzzing has been proven to be effective in finding security vulnerabilities, already having some applications to IoT, which shows it may be applicable to the problem under study. Finally, we already found some security attacks performed on and applicable to IoT networks and devices which may be a good starting point for this research.

3.2 Hypothesis and Validation

We define four IoT systems (*cf.* Table 3.1, p. 23), which are: NC_{BL}, without Self-Healing mechanisms nor Chaos Engineering being applied to it; NC_{SH}, also without Chaos but with Self-Healing capabilities; WC_{BL}, which does not have Self-Healing mechanisms but to which Chaos Engineering is applied; and WC_{SH}, which has both Self-Healing capabilities and also has Chaos applied to it. The work developed in this dissertation aims to validate the following hypothesis:

> ***H:*** *The application of Chaos Engineering leads to a system with higher reliability since we can (a) exercise self-healing mechanisms, and (b) know when these mechanisms are not working correctly.*

We can measure the exercise of Self-Healing in order to validate (a) by verifying that WC_{SH} behaves similarly to NC_{SH}, since when Chaos is applied, the Self-Healing mechanisms in the system are able to recover from the injected faults (which can be confirmed of being injected by observing the measurably different input data). This assertion has as a pre-condition that NC_{BL}

Table 3.1: Definition of four systems with and without Self-Healing mechanisms and Chaos Engineering application, for use in elaborating the hypothesis.

	No Chaos (NC)	With Chaos (WC)
Without Self-Healing (BL)	NC_{BL}	WC_{BL}
With Self-Healing (SH)	NC_{SH}	WC_{SH}

and WC_{BL} must differ significantly in their behaviour, *i.e.*, that the base system without Self-Healing components is unable to recover from the injected faults, ensuring that the performed Chaos Engineering is *"chaotic"* enough.

Furthermore, we can validate (b) by measuring that the functioning of the self-healing mechanisms. If the self-healing mechanisms are working correctly then: (b.1) The similarity of the output between NC_{SH} and WC_{SH} will increase, and (b.2) the similarity between NC_{BL} and WC_{BL} will decrease (*i.e.*, applying Chaos to a system which does not have self-healing capabilities will result in its behaviour differing from the baseline behaviour, without Chaos).

The aforementioned relationships are illustrated in Figure 3.1. NC_{BL} and NC_{SH} should have a similar behaviour in order to confirm that the added self-healing mechanisms are not having a meaningful impact in the system's base behaviour. NC_{BL} and WC_{BL} should have a different behaviour, in order to confirm that the applied chaos is sufficiently severe, impacting the base system's functioning. NC_{SH} and WC_{SH} should be similar, so that we verify that the self-healing mechanisms are able to mitigate the injected faults.

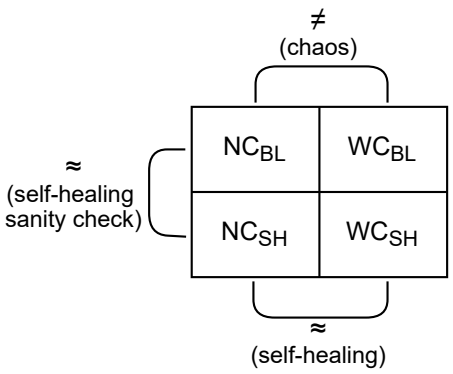

Figure 3.1: Relationships between systems defined in hypothesis.

3.3 Scope

This dissertation focuses on the application of Chaos Engineering to Self-Healing IoT systems. Despite this, since the scope could be too broad if a large variety of systems were taken into account, it was narrowed down to applying Chaos Engineering to an MQTT Broker, as well as focusing on Self-Healing systems developed using Node-RED for validation.

Furthermore, this work will not focus on furthering the self-healing area. Instead, the aim will be of using previously existing self-healing solutions as part of this work's validation.

In terms of target audience, this work may be useful for those considering the hardening of self-healing capabilities in IoT systems. Additionally, the concepts applied by this work can be helpful for fault injection in other technologies, since the process with which this work was developed is also applicable to other technologies.

3.4 Summary

The problem of applying Chaos Engineering to Self-Healing Internet-of-Things systems in order to validate their increased reliability poses several challenges. It is introduced in Section 3.1 (p. 21), along with a knowledge gap analysis found in the literature review done in Chapter 2 (p. 5). Section 3.2 (p. 22) presents the hypothesis that drives this work, as well as defining the driving concept behind its validation.

Chapter 4

Validation

4.1 Methodology

In order to validate this work's hypothesis (*cf.* Section 3.2, p. 22), we have acted upon a thorough process which consists of the following high-level phases:

1. **Proof-of-Concept Implementation:** We implemented a proof-of-concept extension to an MQTT Broker to perform Chaos Engineering in MQTT communications;
2. **Sanity Checks:** The selected Self-Healing tooling (`SHEN`) was tested to ensure they performed as expected in reduced test scenarios;
3. **Scenario Definition:** Several test scenarios and experiments for each of them were defined, after careful consideration of ways to inject faults via Chaos Engineering in relevant systems;
4. **Running the Experiments:** The experiments were conducted in a 4-axis fashion (with and without chaos, and with and without self-healing) with the collection of useful metrics in order to analyse and compare the systems' behaviours;
5. **Experiment Discussion:** We analysed the results of running the experiments in order to conclude if the hypothesis's claims still held after practical experimentation.

These phases will be further explored in the following sections of this chapter.

In order to develop and test systems with self-healing capabilities, we will make use of the `node-red-contrib-self-healing` package[1] by Dias *et al.*[17, 19], also represented as `SHEN`. `SHEN` targets Node-RED, since the author's work was motivated by it being open-source, having widespread popularity, and having widespread adoption by the research community as a target and base for other experiments. Node-RED[2] is a flow-based visual programming tool which provides a browser-based editor for flow editing. It is an open-source tool with a large focus on its community due to its package-driven architecture which allows for easy extensibility by importing external packages developed by other users. Furthermore, it allows deployment to several types of platforms, namely IoT devices.

4.2 Proof-of-Concept Implementation

In order to support the validation of the hypothesis (*cf.* Section 3.2, p. 22), we developed additional code for an existing MQTT Broker. These additions allowed us to use the MQTT Broker as a proxy which was able to intercept and modify messages before they were broadcasted to subscribers of a topic.

The architecture of an example system which makes use of the developed solution is showcased in Figure 4.1. A message emitted by a sensor is processed in the MQTT Broker, which then transmits it to an IoT Rules Engine, that will be able to interact with an actuator through the MQTT Broker once again. Furthermore, Figure 4.2 (p. 27) highlights some of the logic in the developed Chaos Engineering additions for the MQTT Broker.

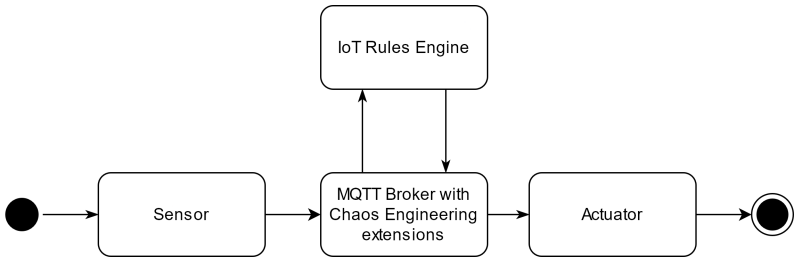

Figure 4.1: Example IoT system which uses the implemented solution.

This section will further elaborate on certain implementation details which may be relevant for those attempting to replicate or expand upon the developed proof-of-concept.

4.2.1 MQTT Broker Middleware

We used the AEDES MQTT broker[3] as a base project from which to expand. In order to be able to convert the MQTT broker into an MQTT proxy, we introduced code which would intercept subscribes, unsubscribes and publish requests and call our own code instead of the broker's original

[1] https://github.com/jpdias/node-red-contrib-self-healing/
[2] https://nodered.org/
[3] https://github.com/moscajs/aedes

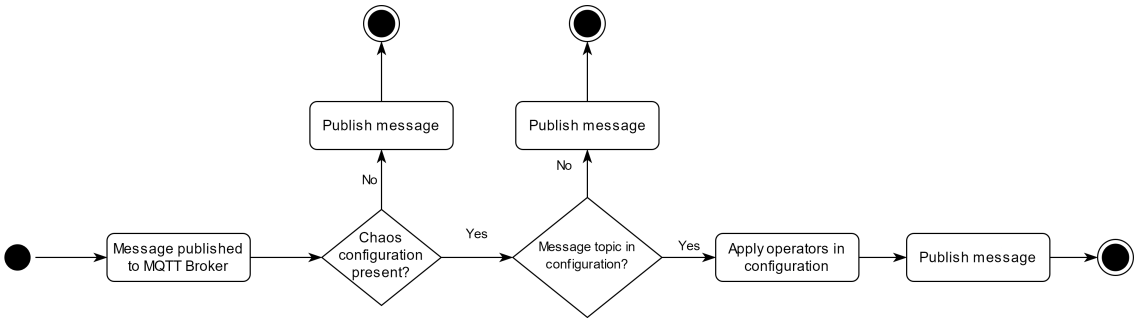

Figure 4.2: Chaos Engineering extension high-level logic.

functions.

After this, we developed a Chaos Engineering scaffolding which is able to modify messages according to user-defined rules. These rules are able to use several operators based upon and implemented with RxJS[4]. This allows the creation of powerful, extensible and modular pipelines of operations that can be applied to MQTT packets that are published in a certain topic.

4.2.2 MQTT Broker Pipeline Operators

In order to ensure that the configuration was sufficiently powerful, we implemented several operators:

Map This operator mimics the `Map` function that many programming languages provide. It runs the provided function, calling it with the current packet as a parameter. The value returned by this function will be used as an argument in following pipeline operators;

Random Delay This operator adds a random delay in the specified interval to the messages in the pipeline;

Buffer This operator buffers up to a maximum time interval and/or number of messages. After the first of these two conditions is achieve, the captured messages are released all at once. The condition can be either time or size based, and both can be provided simultaneously;

Random Drop This operator randomly drops some of the messages according to the specified chance. Messages are dropped before exiting the broker, thus not being broadcasted for any of the subscribers of the target topic;

These operators are based upon ReactiveX[5]. We believe that following an already existing standard may be helpful for adoption of this software, as well as to simplify future extension.

[4] https://rxjs.dev/
[5] http://reactivex.io/

4.2.3 JSON Configuration File

A JSON file must be used in order to configure the Chaos Engineering pipeline. This JSON should have as its root element an array of objects, each defining a pipeline for a specific MQTT topic. Each of these objects must contain a `topic` attribute with a string representing the MQTT topic that this operator should be applied to, as well as an `operators` attribute.

The `operators` attribute is an array containing objects that define a set of operators (*cf.* Subsection 4.2.2, p. 27) to apply in a pipeline fashion.

Besides specific operator arguments, there are two optional arguments of each pipeline definition: `startAfter` and `stopAfter`. These can be used to define the indexes between which the Chaos Engineering pipeline will function. When the message index is in the specified interval ($[startAfter, stopAfter]$ — the interval is inclusive), the Chaos Engineering pipeline is applied normally. On the other hand, when the message index is outside this interval, the pipeline will not be run and the topic will behave as a regular MQTT topic. Messages are zero-indexed, and their indexes are topic-specific.

Several examples of Chaos Engineering configuration files that were used for the validation experiments (*cf.* Section 4.6, p. 35) are included in Appendix B (p. 57).

4.2.4 Summary

The implemented proof-of-concept solution enables fault injection in IoT systems via the manipulation of MQTT messages which are specified by user-defined configuration, resulting in Chaos Engineering. Its extensible configuration is helpful in the creation of varied test scenarios for validating this work's hypothesis, as well as easily applying Chaos Engineering to any MQTT-connected device.

4.3 Sanity Checks

In order to validate that the used self-healing extensions were working as expected, we implemented several Sanity Check Experiments (`SCE`) which aimed to confirm that the chaos engineering injected via the MQTT broker's chaos pipelines correctly triggered `SHEN`'s probes and recovery mechanisms.

Additionally, we performed other ad-hoc experiments in order to validate the correct functioning of all the used `SHEN` nodes.

4.3.1 Testing threshold-check (SCE1)

This sanity check experiment aimed to test the functionality of the `threshold-check` node. Additionally, it was also helpful for some initial validation on the applicability of the `Map` operator (*cf.* Subsection 4.2.2, p. 27) for changing message values.

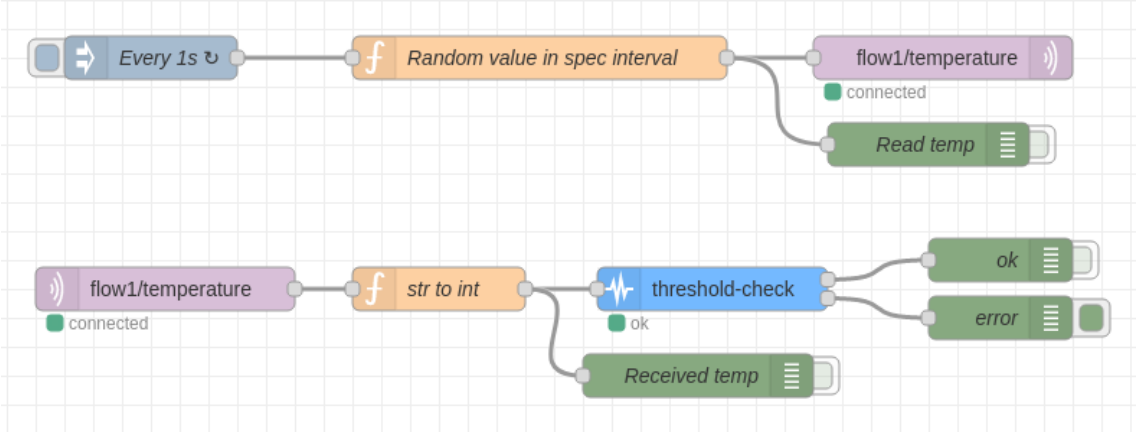

Figure 4.3: **SCE1** Node-RED Flow. This system emits temperatures within a range to a topic, which are then read and checked to be inside the specification range.

As per Figure 4.3 (p. 29), this Node-RED flow represents a system which feeds itself with temperature data from a fictional sensor that emits values inside its operating range. The "Random value in spec interval" `JavaScript` node sets the message payload to a random integer in the range $[0, 40]$, which is considered as the operating range for this sensor. Likewise, the `threshold-check` node is configured so that `msg.payload` must be in this operating interval for the message to be considered valid.

```
[{
    "topic": "flow1/temperature",
    "operators": [
        {
            "type": "map",
            "func": "function (packet) {
                const min = -15;
                const max = 25;
                const shift = Math.floor(Math.random() * (max - min + 1)) + min;
                packet.payload = Buffer.from(
                    (parseInt(packet.payload.toString()) + shift)
                .toString());
                return packet;
            }"
        }
    ]
}]
```

Listing 1: Chaos Engineering Broker configuration for **SCE1**. Presented with added line-breaks for ease of viewing.

Listing 1 presents the configuration used with the Chaos Engineering Broker (*cf.* Section 4.2, p. 26). This configuration resulted in a random shift to the message payload value in a range of $[-15, 25]$.

As such, it is expected that several of the messages originally emitted by the sensor will result in messages with payloads outside of the sensor's specification.

In practice, we verified that the results matched our expectations. When messages were emitted, they were inside the operating range. However, due to the Chaos Engineering pipeline that was applied, some messages would be shifted to a value outside the operating range. These messages were correctly filtered by the `threshold-check` node, thus validating that it is behaving as expected.

4.3.2 Testing timing-check (SCE2)

This sanity check experiment aimed to test the functionality of the `timing-check` node. This is especially useful due to the node's planned usage for **SCE3**.

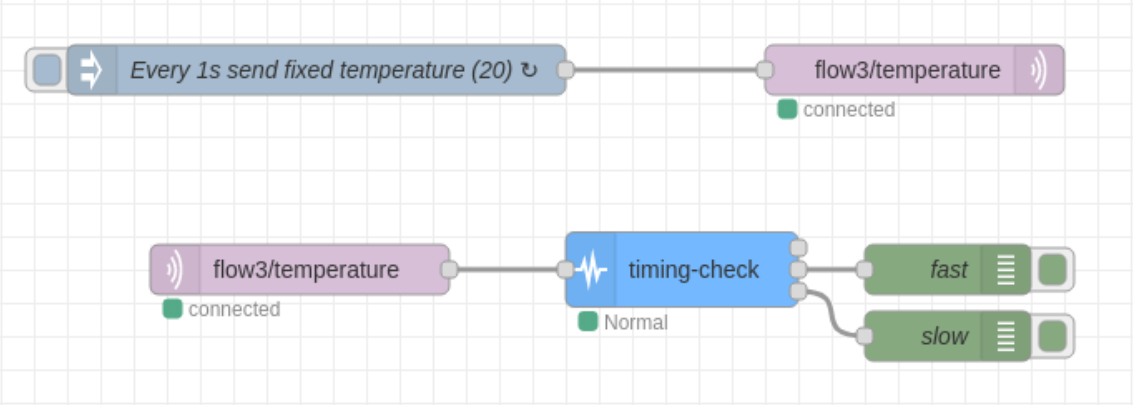

Figure 4.4: **SCE2** Node-RED Flow. This system emits temperatures in a steady timing which are then fed to a `timing-check` node to verify their periodicity.

As per Figure 4.4, this Node-RED flow represents a basic system which feeds itself with temperature data from a fictional sensor that emits values with a standard periodicity of one second. Likewise, the `timing-check` node is configured with an expected period of one second.

```
[
    {
        "topic": "flow4/temperature",
        "operators": [
            {
                "type": "randomDelay",
                "min": 0,
                "max": 500
            }
        ]
    }
]
```

Listing 2: Chaos Engineering Broker configuration for **SCE2**.

In order to validate this node's behaviour, we created another flow mirroring the one above, which used a different topic. To one of the topics, we applied no Chaos Engineering operators. To the other, we applied a **Random Delay** operator (*cf.* Listing 2, p. 30). In theory, the result of this should be: the original topic has messages in the expected periodicity, and the mirrored topic has its messages delayed by a random factor in the range of zero to half a second. We expect that the `timing-check` node for the original system remains in the "Normal" state, while its counterpart in the second system is expected to alternate between "Normal", "Too Fast" and "Too Slow" states.

In practice, our expectations were met. By subscribing the used MQTT topics, we were able to confirm the time elapsed between messages and thus further validate `timing-check`'s correct functioning for both systems.

4.3.3 Testing compensate (SCE3)

This sanity check experiment has the goal of testing the functionality of the `compensate` node. It will be tested by using two systems: one with a reliable topic, and a second system similar to the first one, with an unreliable topic which has a chance of losing messages that are published in it.

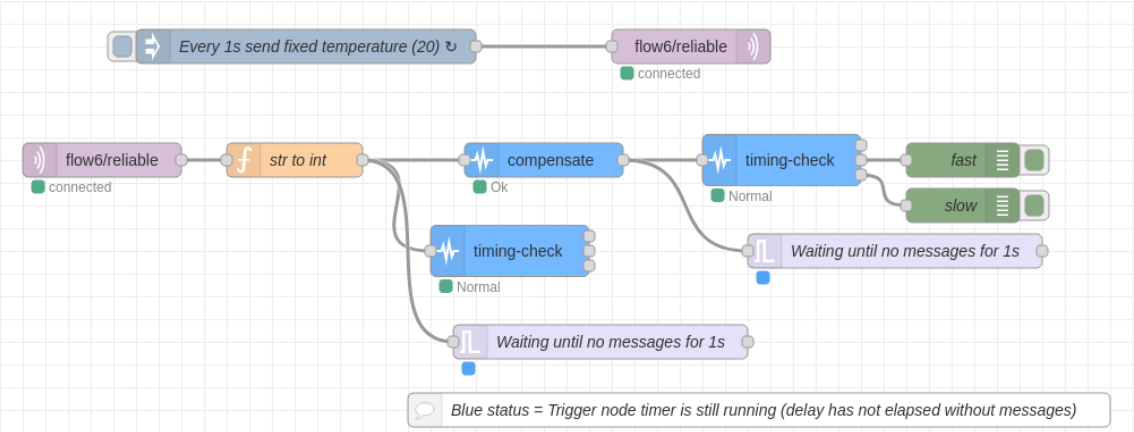

Figure 4.5: **SCE3** Node-RED Flow. This system emits a fixed temperature to a topic, whose periodicity is checked.

As per Figure 4.5, this Node-RED flow represents a basic system that feeds a temperature reading in a fixed interval to a topic. Then, it listens to this topic and using both a `timing-check` and a `trigger` node, confirm the periodicity of the output messages both before and after the `compensate` node. The `compensate` node has been configured with a timeout of 2 seconds and a compensate strategy of "last". The `timing-check` and `trigger` nodes are configured with an expected periodicity of one second.

If the node under test is behaving correctly, it is expected that even when the timing of the received messages is slower than expected (*e.g.*, due to losing some messages), the message periodicity of the node's output will remain stable. It is possible to compare the outputs and states of

the `timing-check` and `trigger` nodes before and after the `compensate` node to verify that this is happening correctly.

```
[{
    "topic": "flow6/reliable",
    "operators": []
},
{
    "topic": "flow6/unreliable",
    "operators": [
        {
            "type": "randomDrop",
            "chance": 0.70
        }
    ]
}]
```

Listing 3: Chaos Engineering Broker configuration for **SCE3**. Messages published in the unreliable topic have a 70% chance of being dropped, while messages in the reliable topic do not have any operators applied, operating normally.

Listing 3 presents the configuration used with the Chaos Engineering Broker (*cf.* Section 4.2, p. 26). This configuration results in the reliable topic operating normally, and the unreliable topic having a 70% chance to drop messages, simulating an unstable network.

In practice, we verified that the results matched our expectations. When messages were dropped the `compensate` node correctly injected messages in the flow according to its compensate strategy, resulting in the timing for the nodes after it being kept steady. This validates that the `compensate` node is working as expected.

4.4 Experimental Dataset

The dataset[6] for the experiments was taken from DeVito *et al.* [12]. We used this dataset since it was collected using a real system, thus it makes sense to consider it for mimicking the real operation of an IoT system. Despite this dataset consisting of hourly averages, we consider the measurements as peak measurements for the purposes of the validation experiments, since the objective is to mimic a real-time IoT system that takes action as soon as a value goes beyond a certain threshold.

The data was filtered and converted using a *Python* script developed with recourse to the `pandas` library. We used the first 120 readings of `NOx(GT)`. Since these readings are average readings and three different readings for three different sensors are necessary, the dataset was modified: we added two extra readings for each sensor value by adding random integer values in a range. Thus, the second and third sensor readings resulted from the aforementioned random shift

[6] DeVito *et al.* [12] dataset: https://archive.ics.uci.edu/ml/datasets/Air+Quality# and http://archive.ics.uci.edu/ml/machine-learning-databases/00360/

in the intervals of $[-10, 10]$ and $[-15, 15]$, respectively. The final dataset containing the readings for the three sensors, as well as an index representing the reading number was saved to a separate `csv` file. The original dataset was carefully analysed by the authors in the scope of their work in [12]. Additionally, dataset metrics are available in Appendix A (p. 55).

The filtered dataset includes 114 instead of 120 values since some are missing in the original dataset. This will produce some inherent chaos, which we believe to be acceptable and even helpful for the experiment.

Since the specification of the sensor's ranges is not clear, it is possible to consider a reference sensor [7] which has a hardware range of 0-1 ppm (0-1000 ppb) and a minimum detection limit of 0.005 ppm (5 ppb). Thus, only values ranging from 5 to 1000 ppb should be considered valid.

As for the dataset's context, "the device was located on the field in a significantly polluted area, at road level, within an Italian city." [12].

Despite there being no clear standards for NO2 concentration levels, the USA-based entity EPA mentions that 0.053ppm should be the average 24-hour limit for NO2 in outdoor air[8]. As such, given that 1 ppm is 1000 ppb, a threshold of 53 ppb may be considered for a warning of potentially harmful levels of NOx. Additionally, another threshold can be naïvely established by multiplying the aforementioned limit by 4, resulting in 212 ppb, which can be considered an average 6-hour limit for NO2 in outdoor air. We considered this a threshold for a higher level of danger and thus have attributed to it an "alarm-level" alarm state.

4.5 Scenarios

In order to validate the previously presented hypothesis (*cf.* Section 3.2, p. 22), we evaluated the impact of the developed *proof-of-concept* Chaos Engineering toolkit (*cf.* Section 4.2, p. 26) using Node-RED flows, communicating over MQTT.

The experiments were done with Node-RED (ver. 1.3.2), and the modified AEDES MQTT broker was ran with NodeJS (ver. 14.15.5). The NodeJS scripts created to run the experiments were ran with the same version as the broker. The duration of each experiment was around 10 minutes.

Two experimental scenarios are the foundation for our experiments:

S1 A set of three NOx sensors feed into an alarm, which emits alerts for certain thresholds of NOx concentration values. Chaos Engineering is applied as faults in the messages' values, simulating sensor malfunctions.

S2 Similar to **S1**, instead applying Chaos Engineering to the messages' timings, simulating faults in the sensors and the network.

These scenarios mirror problems that occur in real systems. Sailhan *et al.*[43] categorize faults in a fashion similar to the one done in this work (*e.g.*, `stuck-at` faults and `spike` faults).

[7] Reference NO2 sensor: https://www.aeroqual.com/product/nitrogen-dioxide-sensor-0-1ppm

[8] "EPA National Ambient Air Quality Standards list 0.053ppm as the avg. 24-hour limit for NO2 in outdoor air." [2]

Furthermore, research has been developed in the area of Self-Healing and autonomic computing related to these types of faults [20, 17, 19].

Three Node-RED flows were implemented, each representing a different system: the baseline system, without any self-healing additions, focusing solely on the system's core functionality, represented as **BL**; the first self-healing system, represented as **SH1**, which extends upon **BL**'s capabilities with self-healing components that may help it recover from faults; and the second self-healing system, represented as **SH2**, which further extends upon **SH1** with more self-healing components which may help the system recover from faults related to the messages' periodicity. The self-healing flows use the `node-red-contrib-self-healing` package[9] by Dias *et al.*[19] first presented in Section 4.1 (p. 25).

For **S1**, we used **BL** and **SH1**, while for **S2** we used **BL** and **SH2**. **BL** was configured with the respective topic that was being used for the specific scenario.

For each scenario, three sensors with NOx readings will be the input, and the expected output is a value in an alarm MQTT topic. This alarm can have the following values, in increasing urgency: 0 (no risk), 1 (warning) and 2 (alert). After analysing the concentrations in the dataset (*cf.* Section 4.4, p. 32), we concluded that the threshold for the warning level should be of 53 ppb and the threshold for the alarm level should be 212 ppb. As such, the alarm's expected behaviour is that:

- For values under 53, the alarm should not be triggered (0);
- For values above 53 but below 212, the alarm should be triggered in a warning state (1);
- For values above 212 the alarm should be triggered in the alarm state (2).

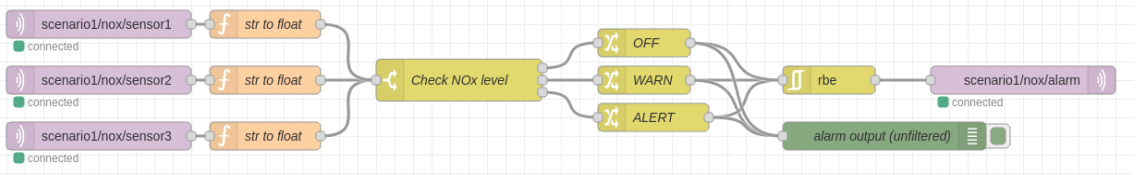

Figure 4.6: Baseline System Node-RED Flow (**BL**). This system parses the reading received from the sensors and sends the respective alarm level as output, filtered by a report-by-exception node to ensure that values are only emitted when the current alarm level changes. The topics in this figure are set to those of Scenario 1 but are altered depending on the Scenario being tested.

In regards to the Node-RED flow implementation of the **SH1** system (*cf.* Figure 4.7, p. 35): the `join` and `compensate` nodes are configured with a timeout of 6 seconds since the messages are expected to have a periodicity of 5 seconds.

In regards to the Node-RED flow implementation of the **SH2** system (*cf.* Figure 4.8, p. 35): the `join` and `compensate` nodes are configured with a timeout of 6 seconds and the `debounce` nodes are configured with a timeout of 4.5 seconds since the messages are expected to have a periodicity of 5 seconds.

[9] https://github.com/jpdias/node-red-contrib-self-healing/

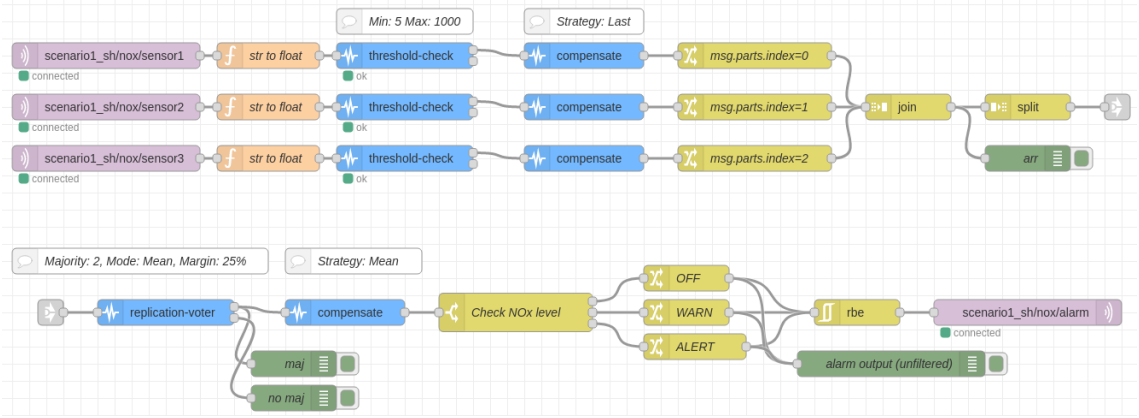

Figure 4.7: First Self-Healing System Node-RED Flow (**SH1**). This system expands upon **BL** by introducing self-healing capabilities via `SHEN` nodes. It filters extraneous messages that are outside the expected operating range, compensates for missing values after a certain timeout, joins messages so that they are considered in groups of 3, considers majority of values with a minimum of 2 "agreeing" values in a 25% value difference margin, and compensates for readings for which there is no majority with a mean of the previous readings, besides the basic functionality already implemented by **BL**.

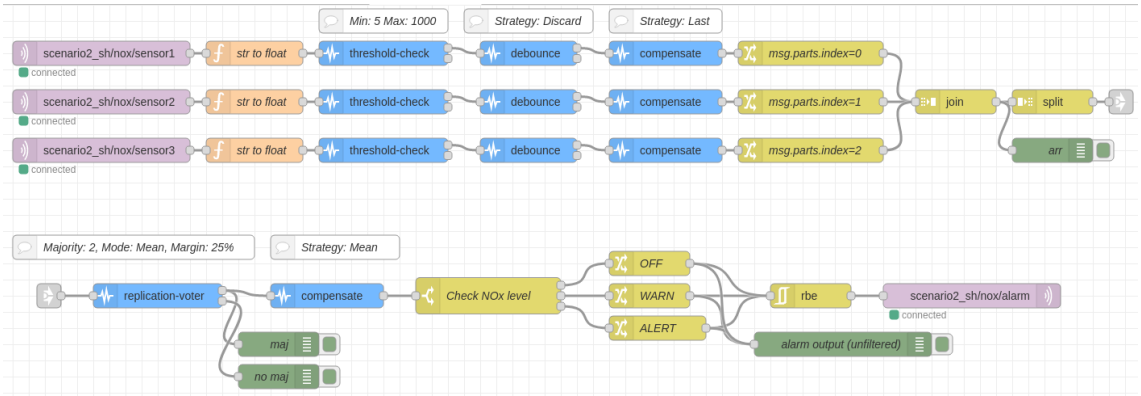

Figure 4.8: Second Self-Healing System Node-RED Flow (**SH2**). This system expands upon **SH1** by introducing `debounce` nodes which can filter out extraneous messages based on the expected timing of the system's regular messages, besides the functionality already implemented by **SH1**.

4.6 Experiments

This section will detail the experiments that were conducted for each experimental scenario. We briefly elaborate upon the several experiments and their respective configuration.

4.6.1 Value Changing Experiments (S1)

Four experiments were done for **S1**. Each of these used a separate chaos configuration so that different operations were applied to the same dataset of messages. For each experiment, messages

were relayed to both **BL** and **SH1** simultaneously to ensure that both systems received the same input and that their outputs (alarm level) could be compared meaningfully with recourse to a Node-RED flow that mirrored the messages that entered a specific input topic. This also ensures that operators which rely on the broker's random number generation do not produce different results for each flow (as the operator is not re-ran). Chaos is applied to messages 10 thru 110 so that the system has time to gain stability before entering a chaotic state and can also resume normality after chaos is applied.

The experiments are as follows:

S1E1 No chaos is applied, as this is the base case. The chaos configuration is empty. This experiment serves as a way to compare the behaviour of **BL** with **SH1** in a situation in which the system functions normally, as well as to provide a baseline with which to compare the functioning of the two systems when chaos is applied with when it isn't, comparing the system's regular behaviour with its behaviour in the presence of faults.

S1E2 A constant fault is injected in sensor 3's readings. As a result, this sensor's readings are altered to be stuck at the upper operating bound (1000 ppb).

S1E3 Readings done by sensor 3 have a 40% chance to be multiplied by a random factor in the range $[0.2, 2.2]$, resulting in "spikes" for sensor values. The factor is randomized for each spike that occurs.

S1E4 Faults are injected in sensor 3 so that it has a 20% chance to lose messages. The system does not received lost messages, as these are suppressed before leaving the MQTT broker.

The used Chaos Engineering configuration files (*cf.* Subsection 4.2.3, p. 28) are included in Appendix B (p. 57).

In order to collect insights on the behaviour of the systems under test, their input and output MQTT topics are monitored and logged during the course of the experiment.

4.6.2 Timing Experiments (S2)

Two experiments were conducted for **S2**. Both experiments were run in different circumstances, created with different script files used for running the experiments with different setups. Similarly to **S1**, messages were delivered simultaneously to **BL** and **SH2** via an auxiliary Node-RED flow, which mirrored the received messages to the input topics of **BL** and **SH2**. Furthermore, Chaos is applied to messages 10 thru 110 so that the system can gain stability before faults are injected and can resume normality after Chaos is applied.

The experiments are defined as follows:

S2E1 No faults are injected for this experiment. Its purpose is to provide a method for confirming that the system's base functionality is correctly implemented for both **BL** and **SH2**, as well as to provide a base experimental output with which to compare the behaviour of the systems in following experiments.

S2E2 In order to introduce additional noise into the system, each message for sensor 3 is repeated after 6 seconds. Since the periodicity of the system's readings in regular circumstances is of 5 seconds, the repeated message will be output in close proximity to the next reading.

Since it is more difficult to visualize the effect of the injected faults in **S2E2**, Figure **??** (p. **??**) depicts the message flow for this experiment as a marble diagram. The top diagram shows a regular message flow, with no faults being injected, and the bottom diagram is the result of applying the operators that are applied in **S2E2** to the original messages.

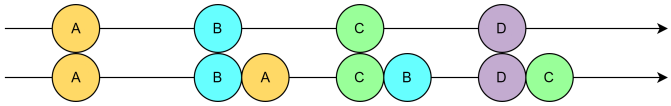

Figure 4.9: Marble diagram of messages for S2E2. The top diagram depicts the regular flow of messages, while the bottom diagram shows the messages after the fault injection done for S2E2.

In order to collect insights on the behaviour of the systems under test, their input and output MQTT topics are monitored and logged during the course of the experiment.

4.7 Discussion

The scenarios and respective experiments were conducted, being the input and output messages of every used topic monitored. The following sections present and discuss the experimental results we obtained.

To analyse these experiments' results, the overlap between two outputs ("output" is considered the result of running an experiment for a specific system, *e.g.*, running **S1E1** for **BL** or **SH1)** has been calculated using a helper script. This metric, mentioned as "overlap percentage", represents the overlap between the ranges of alarm states for two different experimental executions. Any two outputs of experiment executions may be compared by using the logs generated after performing the experiments. While this metric may not be the most accurate (especially if timing faults are injected instead of only value-changing faults), it is an uncomplicated and surefire way to compare two systems' behaviours since we are comparing the final behaviour by comparing the outputs of the systems directly (*i.e.*, if the alarm state is supposed to be in the warning state at a given timestamp for S1E1 and it is not, it may be indicative of a fault that was not correctly handled by the system).

Furthermore, in order to complement the analysis done using the overlap percentage, we calculated the number of alarm level state transitions for each experiment output. These are calculated by counting the number of times that the system transitions to each alarm level. This aims to enable the comparison of the stability of two different systems, *i.e.*, even if two systems for S1E2 have a high overlap percentage with their counterparts in S1E1, one may be more stable than the other, since there may be frequent extraneous alarm level state transitions. These extraneous transitions are not easy to notice using the overlap percentage since they are frequently accompanied

straight away by a transition to the correct alarm state for that given timestamp and thus do not significantly impact the overlap percentage. Thus, this metric may provide further insight for cases in which the previously detailed metric would otherwise result in inconclusive experiments.

4.7.1 Value Changing Experiments (S1)

These experiments focus on evaluating the system's ability to detect and recover from faults injected in the sensor messages, with a particular focus on their values.

4.7.1.1 S1E1

This experiment serves as a baseline case to establish the systems' behaviour in the presence of just the original dataset (no additional faults are injected). It is also insightful to be able to compare the behaviour of **BL** with that of **SH1** since we can then better understand the self-healing mechanisms' recovery profile when there is no added entropy.

We expect that the systems under observation remain stable during this experiment since there are no injected faults, correctly outputting the respective alarm levels for the sensor readings' thresholds. Despite both systems likely having similar alarm levels over time, it is also expected that **SH1**'s alarm level output will be more stable than that of **BL**. This is due to the latter not implementing any type of consensus or majority voting and instead simply using the received values directly as a stream.

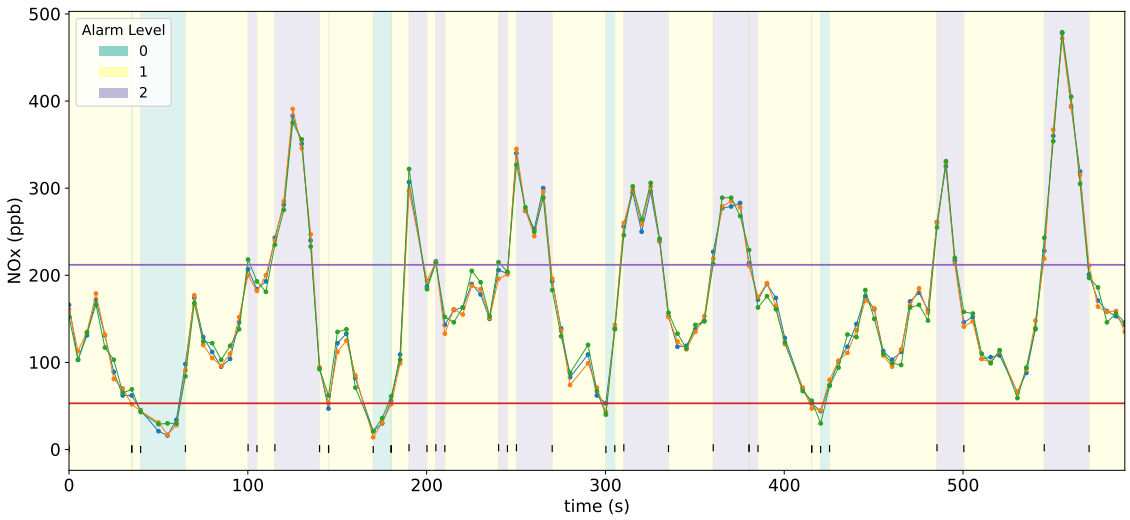

Figure 4.10: NOx concentration and alarm status without self-healing ($S1E1_{BL}$)

Figure 4.10 shows the experiment results for **BL**, while Figure 4.11 (p. 39) shows the experiment results for **SH1**. Despite the alarm output being very similar for both experiment outputs, its stability is higher for **SH1**. This can be observed by the lack of quick alarm state changes for borderline values for **SH1**, which occur several times for **BL** (*e.g.*, around 35 to 40 seconds into the experiment or around 415 to 420 seconds). The cause of this is likely to be **BL**'s lack of

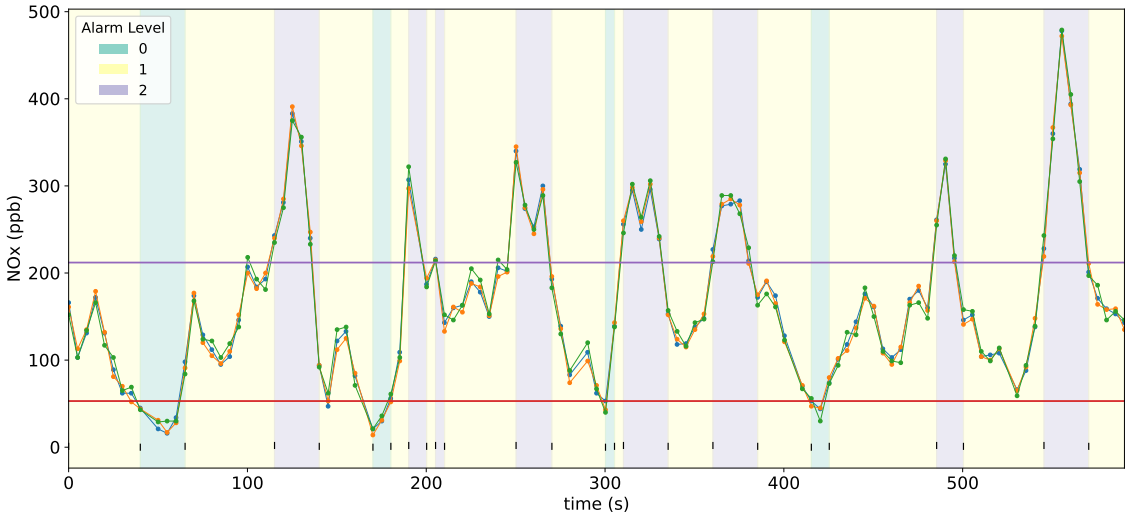

Figure 4.11: NOx concentration and alarm status with self-healing ($S1E1_{SH1}$)

consensus mechanism, given that this system instead simply considers the most recent reading in order to determine the alarm state. When the three sensors report in quick succession, if the values of their readings are near the alarm level thresholds, it is understandable to observe this fluctuation (which is what occurred in the examples above).

$$S1E1_{BL} \cap S1E1_{SH1} = 97.3\% \tag{4.1}$$

The output similarity is confirmed by the calculated alarm level overlap percentage (*cf.* Section 4.7, p. 37) between these two outputs, which is 97.3% (*cf.* Equation 4.1). This is also a good sanity check to confirm that the addition of self-healing capabilities to the base system does not significantly change the alarm status output, which means that comparisons for **SH1** between **S1E1** and further experiments will be meaningful in validating correct self-healing recovery of injected faults.

Table 4.1: Alarm level state transitions for S1E1.

	$S1E1_{BL}$	$S1E1_{SH1}$
OFF (0)	8	4
WARN (1)	20	13
ALERT (2)	11	8
Total	39	25

Table 4.1 clearly supports the previous claim that **SH1** provides an improvement in system stability in comparison to **BL**, due to the lower number of alarm state transitions.

This experiment also confirms that both **BL** and **SH1** correctly implement the expected core functionality (triggering the different alarm levels for different sensor reading thresholds), given that the alarm level at a given point in time (*cf.* Section 4.4, p. 32) corresponds to the sensor

readings' distribution along the thresholds (represented in the mentioned figures by the horizontal lines).

4.7.1.2 S1E2

This experiment simulates a fault in which a sensor malfunctions by continuously emitting readings in its top operating bound.

We expect that **BL** will have an inconsistent output, especially if the third sensor is frequently the last to emit its reading, even if only by a slight delay. Due to relying on a majority of at least 2 values to decide on the alarm level to emit, we expect that **SH1** will be able to withstand the applied Chaos Engineering.

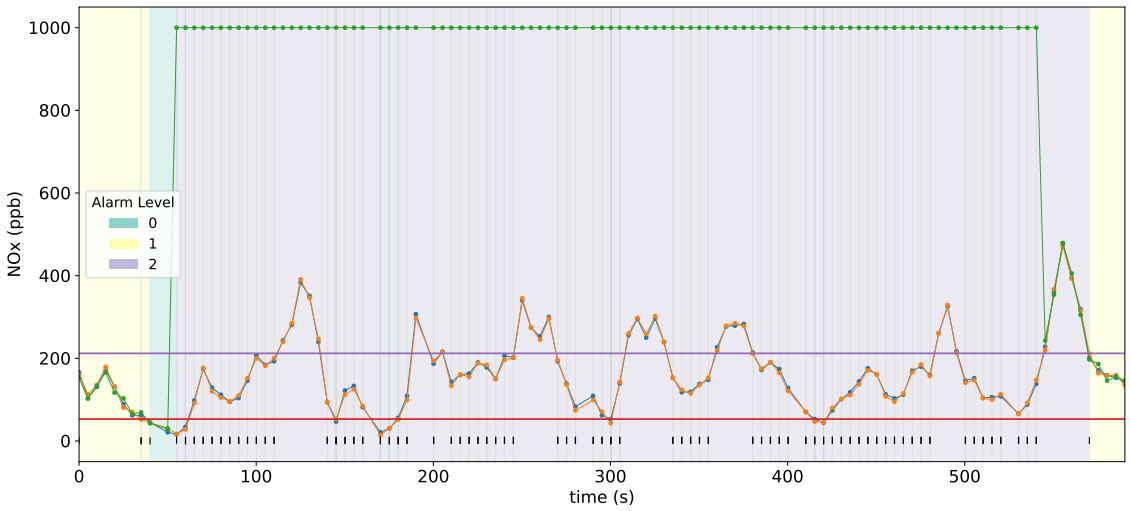

Figure 4.12: NOx concentration and alarm status without self-healing ($S1E2_{BL}$)

Figure 4.12 shows the experiment results for **BL**, while Figure 4.13 (p. 41) shows the experiment results for **SH1**. As expected, the faults that the applied Chaos Engineering injects are too severe for the original system. **BL** malfunctions, quickly alternating between alarm states and especially spending most of the experiment's time in the highest alarm level. Meanwhile, **SH1** seems to have successfully recovered from the injected faults, having a near-perfect performance in comparison to this system's output for **S1E1**.

Table 4.2: **S1E2** Overlap Percentages in comparison with the base experiment (**S1E1**).

	$S1E1_{BL}$	$S1E1_{SH1}$
$S1E2_{BL}$	40.0%	-
$S1E2_{SH1}$	-	98.1%

These statements are supported by Table 4.2, which shows that **SH1** has a similar behaviour when chaos is and is not applied, with an overlap of 98.1%. Furthermore, **BL** has a significantly lower overlap percentage of 40.0%.

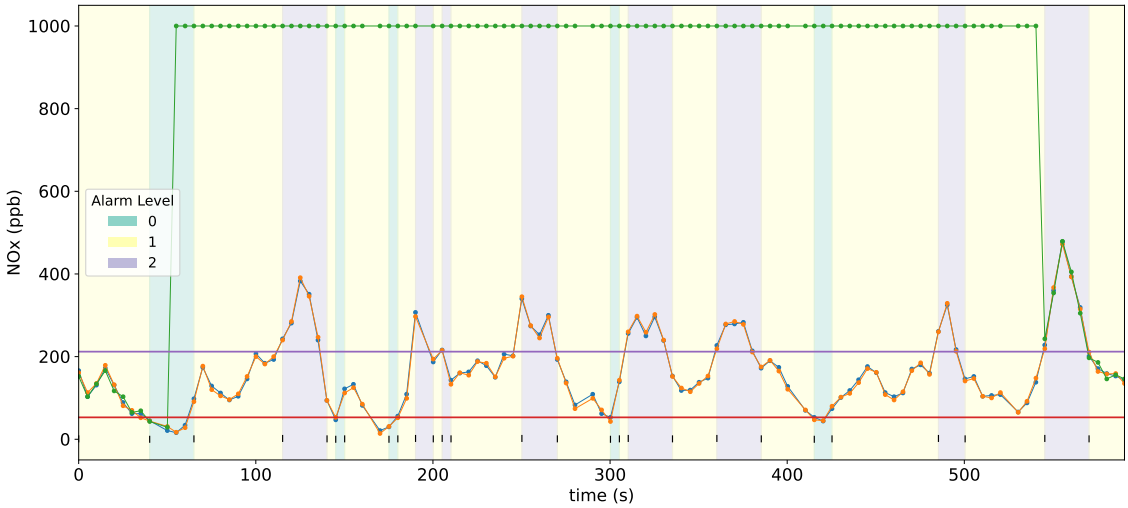

Figure 4.13: NOx concentration and alarm status with self-healing ($S1E2_{SH1}$)

The previous statements are further supported by Table 4.3 (p. 42), in which $S1E2_{BL}$ is much more unstable in comparison to $S1E1_{BL}$, being the total number of state transitions for this experiment 148, while there were only 39 state transitions for the base experiment. Furthermore, the number of state transitions for **SH1** has increased only marginally, going from 25 in the base experiment to 27 in this experiment.

Therefore, **S1E2** demonstrates that the original system cannot handle "sensor-stuck-at"-type errors since the behaviour of **BL** is considerably affected, which validates that the performed Chaos Engineering was meaningful enough to disturb the system's regular operation. On the other hand, **SH1** can recover from the injected faults, having a remarkably similar behaviour to that in **S1E1** (behaving similarly to when the system did not have any chaos injected).

4.7.1.3 S1E3

This experiment simulates a fault in which a sensor malfunctions by occasionally *"spiking"* its readings. This is simulated by multiplying a reading by a random factor (in the range $[0.2, 2.2]$). This fault may naturally occur when a sensor is running out of battery, for example.

We expect that **BL** may output incorrect alarm values (in comparison to its behaviour in **S1E1**), especially when the altered values switch between alarm level thresholds. On the other hand, **SH1** should be able to handle the changes in values since even if one of the three sensors outputs a value considerably different from the others, it will be discarded and the other two sensors' values will be considered instead, due to its usage of the `replication-voter` node from the `SHEN` library.

Figure 4.14 (p. 42) shows the experiment results for **BL**, while Figure 4.15 (p. 42) shows the experiment results for **SH1**. **BL** has had a notably good performance in the presence of the sensor reading spikes (both when increasing and decreasing the read value). However, there were still several situations in which the sensor reading spike caused the output alarm level to differ from

Table 4.3: Alarm level state transitions for S1E2.

	$S1E2_{BL}$	$S1E2_{SH1}$
OFF (0)	10	5
WARN (1)	68	14
ALERT (2)	70	8
Total	148	27

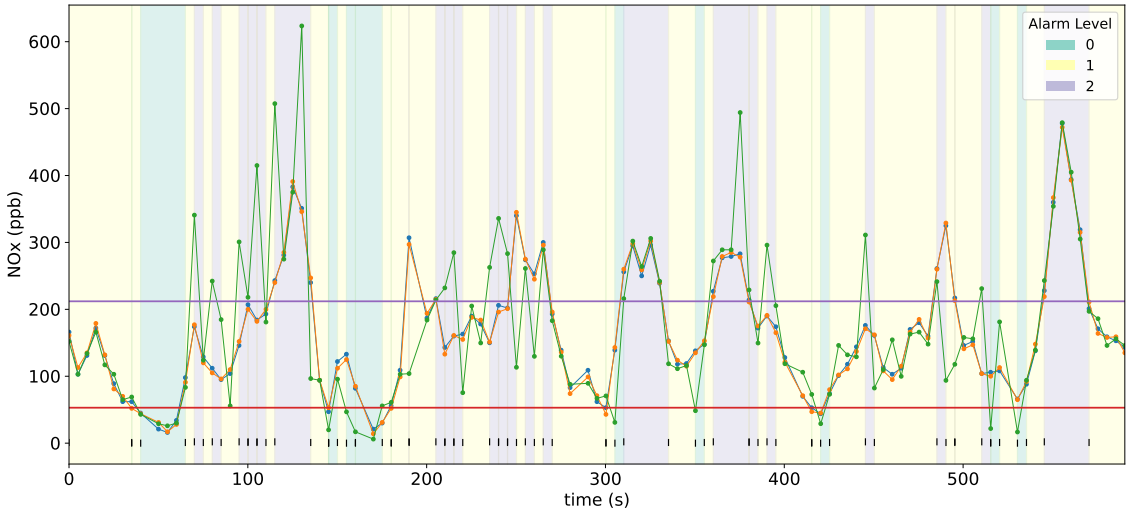

Figure 4.14: NOx concentration and alarm status without self-healing ($S1E3_{BL}$)

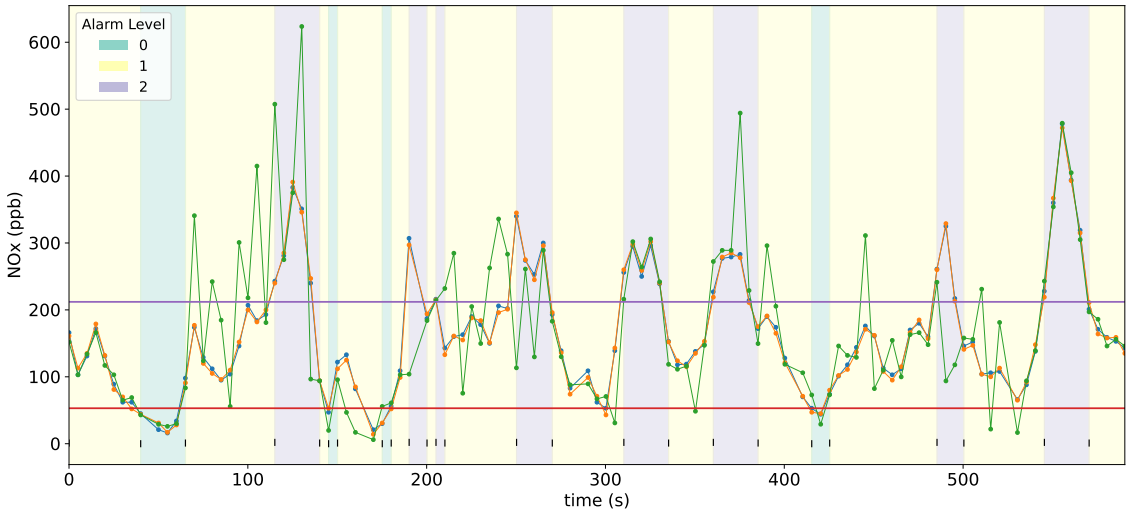

Figure 4.15: NOx concentration and alarm status with self-healing ($S1E3_{SH1}$)

the expected value in the base case (without Chaos). On the other hand, **SH1** has held up to our expectations, handling the injected faults quite well and almost perfectly matching the output of the base case.

Table 4.4: **S1E3** Overlap Percentages in comparison with the base experiment (**S1E1**).

	$S1E1_{BL}$	$S1E1_{SH1}$
$S1E3_{BL}$	76.3%	-
$S1E3_{SH1}$	-	97.4%

These statements are supported by Table 4.4, which shows that **SH1** has a similar behaviour when Chaos is and is not applied, with a near-perfect overlap of 97.4%, while **BL** has a somewhat lower overlap percentage of 76.3%.

Table 4.5: Alarm level state transitions for S1E3.

	$S1E3_{BL}$	$S1E3_{SH1}$
OFF (0)	8	4
WARN (1)	26	13
ALERT (2)	17	8
Total	51	25

Table 4.5 further supports the previous statements. Despite the difference not being as remarkable as that of the overlap percentages, it is of note to mention that **SH1** has the exact same number of alarm level state transitions as it did in **S1E1** while the number of state transitions has slightly increased for **BL**.

Despite this experiment not causing as significant a variation of the behaviour of **BL** as **S1E2**, we were still able to observe a mismatch between the behaviour of this system between the base case and this experiment, even if to a lesser extent. This shows that for the system under study, these faults are not as severe as those injected in **S1E2**. Nevertheless, due to the decline in the overlap percentage for **BL** in comparison with **S1E1**, we can conclude that the faults injected using Chaos Engineering were significant enough to affect the system's correct functioning.

As such, since **SH1** displayed a performance with much greater similarity in comparison to the base case when Chaos was not applied, we are able to confirm that for this experiment the presence of self-healing capabilities are beneficial for the system's correct operation.

4.7.1.4 S1E4

This experiment simulates a fault in which the sensor loses some readings. This fault may naturally occur when a sensor is disconnected, has an intermittent power supply, or the network is unstable.

We expect that **BL** may report erroneous alarm values (compared to the base experiment, **S1E1**), especially when the missing values are in proximity to the alarm thresholds. **SH1** should be able to handle the injected faults by compensating the missing values with previous ones.

Figure 4.16 (p. 44) shows the experiment results for **BL**, while Figure 4.17 (p. 44) shows the experiment results for **SH1**. **BL** has shown a great ability to handle the loss of some readings, being that the alarm output is quite similar to the one in **S1E1**. **SH1** is also able to handle the loss of readings, similarly having almost the same behaviour as in the base case. After further analysis,

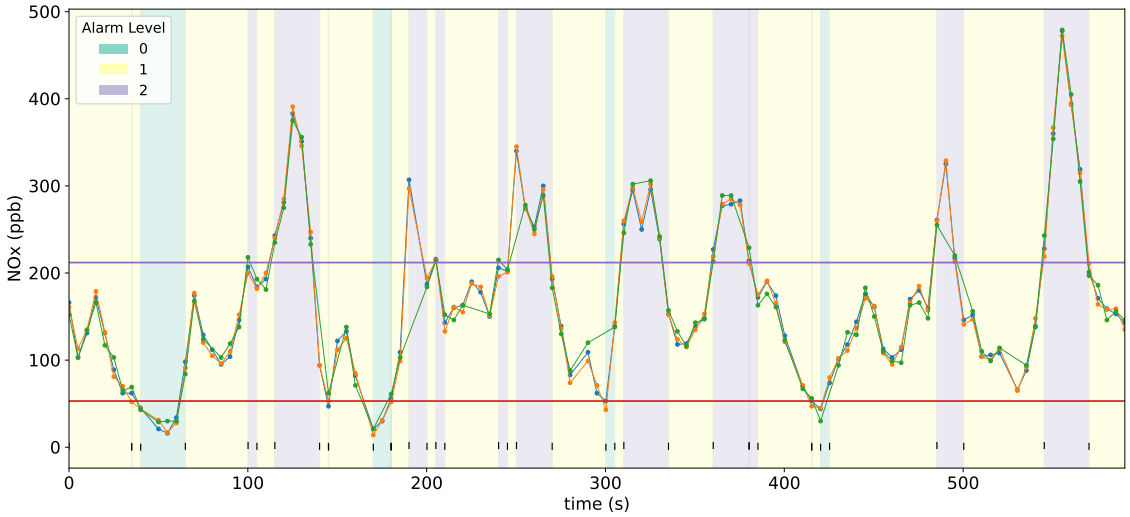

Figure 4.16: NOx concentration and alarm status without self-healing ($S1E4_{BL}$)

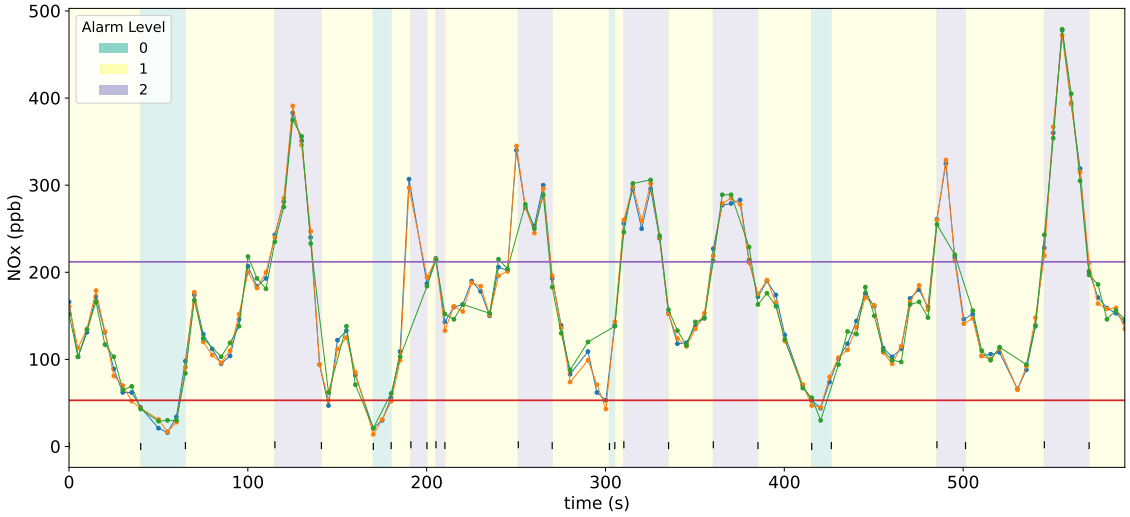

Figure 4.17: NOx concentration and alarm status with self-healing ($S1E4_{SH1}$)

this may be attributed to the fact that since the chance of failure was low, not many readings were suppressed. Furthermore, since the values for different sensors in the original dataset are in great proximity to one another, even heavy suppression of values from one of the sensors would likely result in even the naïve **BL** system being able to output the expected alarm values for most of the experiment's duration.

These statements are supported by Table 4.6 (p. 45), which shows that **SH1** has a similar behaviour when chaos is and is not applied, with a near-perfect overlap of 99.8%. Furthermore, **BL** has a slightly lower overlap percentage of 98.7%.

The previous claims are further supported by Table 4.7 (p. 46), in which we may observe that the number of state transitions for both systems are identical to those that occurred in **S1E1**.

Table 4.6: **S1E4** Overlap Percentages in comparison with the base experiment (**S1E1**).

	$S1E1_{BL}$	$S1E1_{SH1}$
$S1E3_{BL}$	99.8%	-
$S1E3_{SH1}$	-	98.7%

In sum, this experiment did not cause a significant enough deviation from the base experiment's behaviour for **BL**, which is corroborated by the high overlap percentage for **BL** between **S1E1** and **S1E4** ($S1E1_{BL} \cap S1E4_{BL} = 98.7\%$), as well as the fact that the number and type of alarm state transitions are identical to those of **S1E1** (*cf.* Table 4.7, p. 46). As such, and also given the aforementioned analysis of this experiment's results, we conclude that this dataset may not be the best candidate for this type of fault injection. Furthermore, it may be necessary to have a higher chance of message suppression in order for the injected faults to become more meaningful.

4.7.2 Timing Experiments (S2)

The experiments detailed in the following sections focus on evaluating the system's ability to detect and recover from faults injected in the timing of the sensors messages.

4.7.2.1 S2E1

This experiment serves as a baseline to establish the systems' behaviour in the presence of just the original dataset (no additional faults are injected). It is also insightful to be able to compare the behaviour of **BL** with that of **SH2** since we can then better understand the self-healing mechanisms' recovery profile when there is no added entropy. This experiment is extremely similar to **S1E1** in its purpose, and its aim is to confirm that the additions made to **SH1** do not impact the core functionality of the system (much like **S1E1** was meant to confirm this by comparing **BL** and **SH1** in a stable environment).

We expect that the systems under observation remain stable during this experiment since there are no injected faults, correctly outputting the respective alarm levels for the sensor readings' thresholds. Despite both systems likely having similar alarm levels over time, it is also expected that **SH2**'s alarm level output will be more stable than that of **BL**. This is due to the latter not implementing any type of consensus or majority voting and instead simply using the received values directly as a stream.

Figure 4.18 (p. 46) shows the experiment results for **BL**, while Figure 4.19 (p. 46) shows the experiment results for **SH2**. Despite the alarm output being very similar for both experiment outputs, its stability is higher for **SH2**. This can be observed by the lack of quick alarm state changes for borderline values for **SH2**, which occur several times for **BL** (*e.g.*, around 35 to 40 seconds into the experiment or around 415 to 420 seconds). The cause of this is likely to be **BL**'s lack of consensus mechanism, given that this system instead simply considers the most recent reading in order to determine the alarm state. When the three sensors report in quick succession,

Table 4.7: Alarm level state transitions for S1E4.

	$S1E4_{BL}$	$S1E4_{SH1}$
OFF (0)	8	4
WARN (1)	20	13
ALERT (2)	11	8
Total	39	25

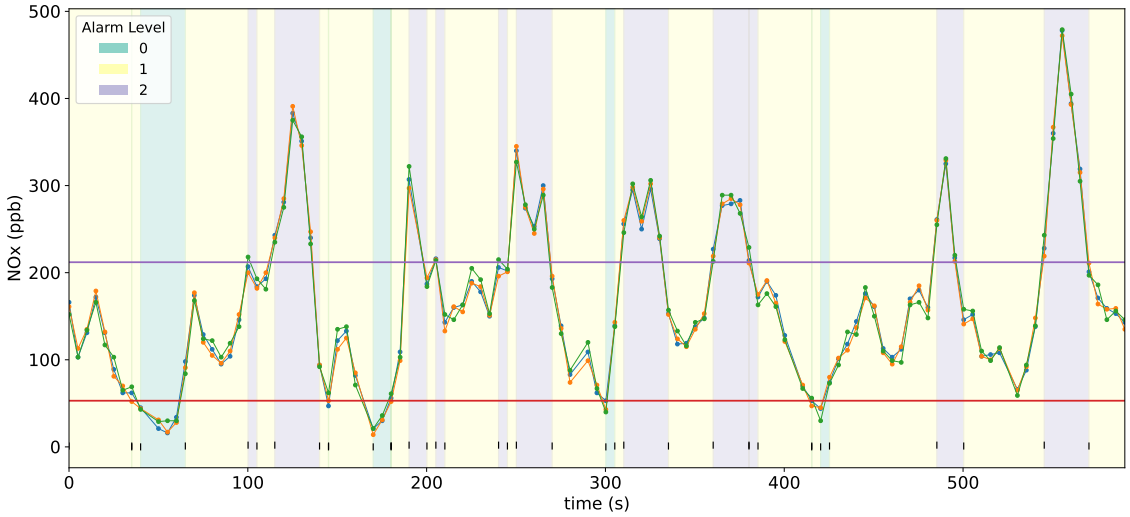

Figure 4.18: NOx concentration and alarm status without self-healing ($S2E1_{BL}$)

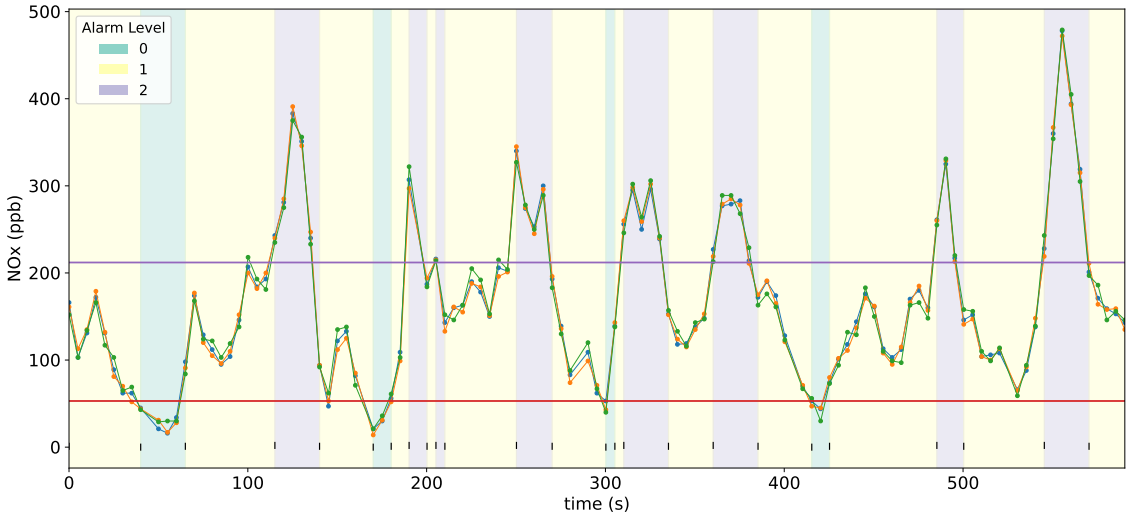

Figure 4.19: NOx concentration and alarm status with self-healing ($S2E1_{SH2}$)

if the values of their readings are near the alarm level thresholds, it is understandable to observe

this fluctuation (which is what occurred in the examples above).

$$S2E1_{BL} \cap S2E1_{SH2} = 97.4\% \tag{4.2}$$

The output similarity is confirmed by the calculated alarm level overlap percentage (*cf.* Section 4.7, p. 37) between these two outputs, which is 97.4% (*cf.* Equation 4.2). This is also a good sanity check to confirm that the addition of self-healing capabilities to the base system does not significantly change the alarm status output, which means that comparisons for **SH2** between **S2E1** and further experiments will be meaningful in validating correct self-healing recovery of injected faults.

Table 4.8: Alarm level state transitions for S2E1.

	$S2E1_{BL}$	$S2E1_{SH2}$
OFF (0)	8	4
WARN (1)	20	13
ALERT (2)	11	8
Total	39	25

Similarly to **S1E1**, Table 4.8 clearly supports the previous claim that **SH2** provides an improvement in system stability in comparison to **BL**, as shown by the lower number of alarm state transitions.

This experiment also confirms that both **BL** and **SH2** correctly implement the expected core functionality (triggering the different alarm levels for different sensor reading thresholds), given that the alarm level at a given point in time (*cf.* Section 4.4, p. 32) corresponds to the sensor readings' distribution along the thresholds (represented in the mentioned figures by the horizontal lines).

4.7.2.2 S2E2

In order to introduce additional noise into the system, this experiment simulates a fault in which messages for a sensor are repeated after a certain delay (6 seconds). Since the periodicity of the system's readings in ordinary circumstances is of 5 seconds, the repeated message will be output in close proximity to the next reading.

We expect that **BL** will have an output that is less stable than it was for **S2E2** due to the injected faults. It is not easy to predict how much this will affect the system. However, we believe it may be problematic for **BL** since it does not have any concept of message timing. On the other hand, **SH2** should be able to cope with the injected faults without much issue due to the `debounce` nodes that it contains, which will filter out the additional messages.

Figure 4.20 (p. 48) shows the experiment results for **BL**, while Figure 4.21 (p. 48) shows the experiment results for **SH2**. **BL** performed significantly better than expected, despite struggling at times when messages were near the alarm level thresholds, since repeating the previous message would cause the system to output the previous alarm level once again until it received the following

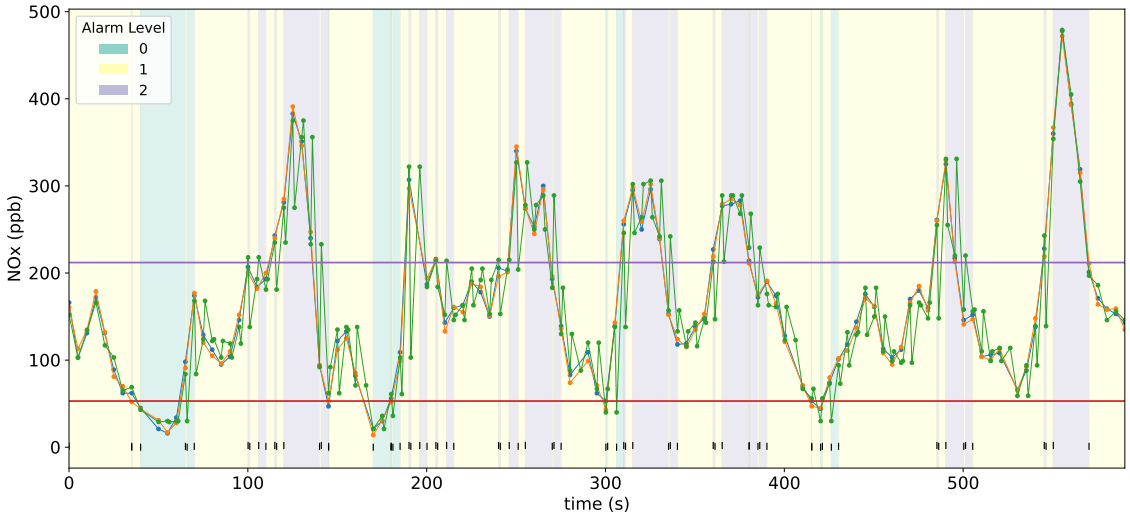

Figure 4.20: NOx concentration and alarm status without self-healing ($S2E2_{BL}$)

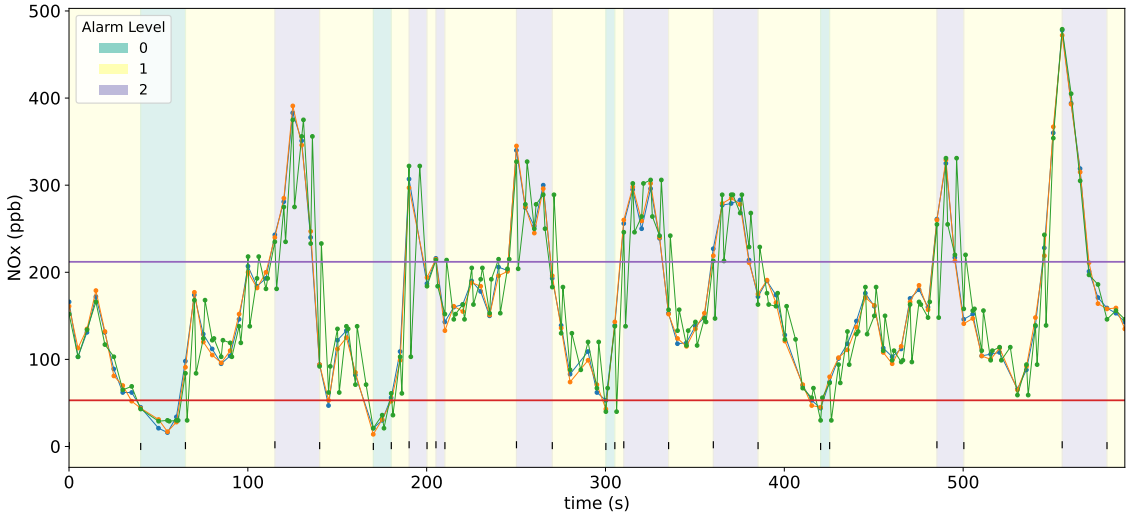

Figure 4.21: NOx concentration and alarm status with self-healing ($S2E2_{SH2}$)

message and went back to the expected state. This may be caused by the fact that the periodicity of the sensor readings messages is quite high and thus whenever a fault is injected, it is not "in effect" for a long duration. Furthermore, as expected, **SH2** was able to cope with the applied Chaos Engineering quite well, having a near-perfect behaviour in comparison to the base experiment (**S2E1**).

These statements are supported by Table 4.9 (p. 49), which shows that **SH2** has a similar behaviour when chaos is and is not applied, with an overlap of 95.7%. Furthermore, **BL** has a slightly lower overlap percentage of 83.4%.

Despite the high overlap percentages for **BL** (*cf.* Table 4.9, p. 49), Table 4.10 (p. 50) shows that for **S2E2**, **BL** has output nearly two times the amount of alarm level state transitions in comparison

Table 4.9: **S2E2** Overlap Percentages in comparison with the base experiment (**S2E1**).

	$S2E1_{BL}$	$S2E1_{SH2}$
$S2E2_{BL}$	83.4%	-
$S2E2_{SH2}$	-	95.7%

to its performance in **S2E1**.

S2E2 shows that despite **BL** having had a slightly worse performance when compared to **S2E1**, the difference in performance (show by the overlap percentage) is not very large. Despite this, since **SH2**'s performance was better, we may attribute its better ability to cope with the injected faults to its additional Self-Healing capabilities. Furthermore, **BL** also had a noticeably worse performance in regards to the number of alarm level state transitions in comparison to **SH2** (*cf.* Table 4.10, p. 50). Finally, while it is not easy to conclude if this experiment caused a significant enough deviation from the base experiment's behaviour for **BL** judging solely based on the overlap percentage, the difference in alarm level state transition for **BL** in comparison to **S1E1** provides evidence that the injected Chaos has caused issues on the baseline system which the self-healing system is able to address.

4.8 Hypothesis Evaluation

The validation process described in this chapter aimed to prove the hypothesis presented in Section 3.2 (p. 22). Given the experimental results, we can conclude that we focused on the hypothesis' core ideas, the exercise of Self-Healing mechanisms using Chaos Engineering.

S1E1, **S1E2**, **S1E3**, **S2E1**, and **S2E2** allowed us to confirm our hypothesis, concluding that: (1) the self-healing systems (**SH1** and **SH2**) do not deviate too much in behaviour from the baseline system (**BL**); (2) the faults injected via Chaos Engineering are meaningful since there is a deviation on the baseline system in comparison to the base experiment when no Chaos is being injected; (3) when the Chaos is meaningful, the self-healing systems were able to recover from it, replicating the original behaviour (when no Chaos was present) despite the injected Chaos, and thus confirming that the Self-Healing mechanisms were being exercised.

On another hand, **S1E4** was equally useful in understanding that it is paramount for the behaviour of **BL** to be *noticeably* different when faults are being injected in comparison to the regular operating circumstances. This factor made it so that we considered this experiment inconclusive. Nevertheless, it has helped validate that it is necessary to find this stark difference in expected versus observed output for the baseline system. Otherwise, one cannot be sure if the Self-Healing components are doing any work at all, since a naïve system would already be able to "recover" from the injected faults well enough.

It is also noteworthy that for the created scenarios, due to the used dataset (*cf.* Section 4.4, p. 32), some types of chaos did not result in much instability. This is due to the fact that all three sensors output readings with values in close proximity to each other. As such, if one or even two sensors fail, it is likely that a naïve system (*e.g.*, **BL**) will still perform as expected, outputting the correct

Table 4.10: Alarm level state transitions for S2E2.

	$S2E2_{BL}$	$S2E2_{SH2}$
OFF (0)	12	4
WARN (1)	36	13
ALERT (2)	25	8
Total	73	25

alarm levels for most cases. This is an indicator that further research should be done with other types of datasets, as well as different types of injected faults via Chaos Engineering.

Finally, it is also of note to mention that from the observed results during the experiments, we can conclude that both self-healing systems (**SH1** and **SH2**) are able to cope with even drastic faults in one sensor, as long as two others remain in working condition.

4.9 Summary

This chapter presents the results from the validation process of the developed solution, as well as this work's hypothesis. Section 4.1 (p. 25) starts by defining the methodology used for validation, and Section 4.5 (p. 33) defines the experimental scenarios, also providing some insight on the used dataset.

Furthermore, Section 4.6 (p. 35) further details the experiments for each scenario and their purpose, and Section 4.7 (p. 37) explores these further by showcasing the experiments' results and starting some analysis and discussion upon them.

Finally, in Section 4.8 (p. 49), this work's hypothesis is revisited, and we build upon the experimental results to validate our hypothesis.

Furthermore, the validation process was helpful in identifying possibilities for improving the implemented solution. We developed several companion scripts, especially for collecting data and generating graphs to support the validation process. An especially relevant addition made due to the evaluation process was that of the `startAfter` and `stopAfter` attributes for Chaos Engineering configuration due to the need of limiting the application of the Chaos Engineering pipeline to a specific interval of messages.

Chapter 5

Conclusions

5.1 Conclusions

As IoT devices and systems become commonplace in everyday life, it is increasingly important that they work correctly. Failures in their functioning may result in consequences ranging from slight discomfort to life-threatening situations. Fault-tolerant systems have been a subject of research for some time in order to tackle these issues. These systems have increased reliability due to being able to handle faults that may occur. Additionally, more recently, approaches based on autonomic computing and self-healing have been introduced, providing valuable tools to handle and recover from failures in complex IoT systems.

Due to the complexity and heterogeneity of Internet-of-Things systems, as well as the lack of technical knowledge of most of their users, it is becoming paramount that faults and issues of IoT devices and systems are fixed or mitigated without the need of technical intervention by the users. Therefore, the application of autonomic computing and self-healing is becoming progressively relevant. As consequence, the need also arises to ensure that these capabilities are working correctly, further increasing the confidence in the adequate functioning of complex IoT systems.

The literature review showed that Chaos Engineering's applications currently focus mainly on cloud infrastructure, without much focus in IoT systems. Moreover, both Fault Injection and Fuzing were identified as possible techniques to use alongside Chaos Engineering in the scope of this work.

This work focuses on the core problem of being able to exercise fault-tolerance mechanisms — more concretely, self-healing strategies — using Chaos Engineering. Our work revolves upon the following central hypothesis:

> ***H:*** *The application of Chaos Engineering leads to a system with higher reliability since we can (a) exercise self-healing mechanisms, and (b) know when these mechanisms are not working correctly.*

In order to validate our hypothesis, we implemented an extension to an MQTT Broker which allowed us to specify Chaos Engineering pipelines which would be applied to certain MQTT

topics. This made it straightforward to emulate a wide variety of faults in messages, according to the given configuration.

Afterwards, we performed experiments in order to validate our hypothesis. Initially, we created scenarios which would focus on injecting faults in message values and payloads, and in message timings. While some experiments were not conclusive due to the applied Chaos not causing a significant divergence from the system's behaviour when Chaos was not applied, others allowed us to clearly validate our hypothesis. The inconclusive experiments also added to the idea that the experiments must be carefully crafted given that the used dataset must be taken into account, as well as the types of faults to inject.

In conclusion, we developed a straightforward way to apply Chaos Engineering along pre-determined rule-sets to a given system by extending an MQTT Broker. Then, we used this method to create several test scenarios and experiments, which validated our initially proposed hypothesis.

5.2 Contributions

The work developed for this dissertation resulted in the following contributions:

Literature Review on Fault Injection, Chaos Engineering and Fuzzing in IoT: We researched about Fault Injection, Chaos Engineering, Fuzzing and their applicability and current applications to IoT systems and devices, as well as some security-related implications of these techniques (*cf.* Chapter 2, p. 5);

MQTT Broker for Chaos Engineering using user-defined pipelines: We implemented extensions to a pre-existing MQTT Broker which allow it to perform Chaos Engineering and Fault Injection via user-defined pipelines with rules on how to modify packets for specified topics (*cf.* Section 4.2, p. 26);

Experimental Validation of the proposed Hypothesis: We conducted experiments with the aim of validating this work's hypothesis, providing insight on how to perform these kinds of experiments as well as extracting some conclusions from the observed results (*cf.* Chapter 4, p. 25).

5.3 Future Work

For this work, we extended a pre-existing MQTT Broker in order to be apply to modify messages that pass through it according to certain rules (Chaos Engineering pipeline). However, this solution has some room for improvement, namely: (1) further simplify the Chaos configuration by supporting more native language constructs (*e.g.*, arrow functions) and other configuration abstractions (*e.g.*, leverage visual notations); (2) consider switching to a more user-friendly format since the definition of `JavaScript` functions for the `map` operator is especially cumbersome; (3) support wildcard topics as per the MQTT specification; and (4) enable switching configuration at

run-time instead of having to specify the configuration file when starting the broker. Further, it would be of relevance to understand how such a tool could be adapted to provoke *chaos* in IoT systems were the different entities of the system perform computation (instead of relying on a single computation unit, viz. Node-RED) [37, 45, 46].

In regards to the evaluation, it would be interesting to: (1) expand the scenarios with more experiments; (2) attempt similar experiments using different datasets (since we had some issues due to specific aspects of the used dataset); (3) use other self-healing implementations to further validate this work's hypothesis (no other implementations as complete as the one used are known to us as of yet).

Concerning this work as a whole, several possible avenues of research were identified during the Literature Review (*cf.* Chapter 2, p. 5) which were not explored, such as: (1) Fuzzing (which may be applicable by *e.g.*, injecting rogue messages according to identified patterns); (2) replication or injection of physical faults in the system; and (3) deauthentication, denial-of-service and other similar security attacks which may cause system instability.

Appendix A

NOx Dataset Statistics

```
NOx_s1 count    114.000000
mean     165.973684
std       89.350009
min       16.000000
25%      104.500000
50%      152.500000
75%      212.250000
max      478.000000
NOx_s2 count    114.000000
mean     165.175439
std       89.994419
min       14.000000
25%      102.500000
50%      152.500000
75%      211.000000
max      472.000000
NOx_s3 count    114.000000
mean     166.043860
std       89.335045
min       21.000000
25%      103.000000
50%      151.000000
75%      214.750000
max      479.000000
```

Listing 4: NOx Dataset Statistics obtained by using `pandas`'s `.describe` method.

Appendix B

Chaos Engineering JSON Configurations

```
[]
```

Listing 5: Chaos Engineering Broker configuration for S1E1.

```
[
    {
        "topic": "scenario1_mirror/nox/sensor3",
        "startAfter": 10,
        "stopAfter": 102,
        "operators": [
            {
                "type": "map",
                "func": "function (packet) {
                    packet.payload = Buffer.from('1000');
                    return packet;
                }"
            }
        ]
    }
]
```

Listing 6: Chaos Engineering Broker configuration for S1E2. Presented with added line-breaks for ease of viewing.

```
[
    {
        "topic": "scenario1_mirror/nox/sensor3",
        "startAfter": 10,
        "stopAfter": 102,
        "operators": [
            {
                "type": "map",
                "func": "function (packet) {
                    if (Math.random() > 0.6) {
                        const min = 0.2;
                        const max = 2.2;
                        const factor = Math.random() * (max - min) + min;
                        packet.payload = Buffer.from(
                        (
                            parseFloat(packet.payload.toString()) * factor
                        ).toString());
                    }
                    return packet;
                }"
            }
        ]
    }
]
```

Listing 7: Chaos Engineering Broker configuration for S1E3. Presented with added line-breaks for ease of viewing.

```
[
    {
        "topic": "scenario1_mirror/nox/sensor3",
        "startAfter": 10,
        "stopAfter": 102,
        "operators": [
            {
                "type": "randomDrop",
                "chance": 0.20
            }
        ]
    }
]
```

Listing 8: Chaos Engineering Broker configuration for S1E4.

References

[1] Eric Sesterhenn aka. Snakebyte and Martin J. Muench aka. mjm. BED: Bruteforce exploit detector. Available at https://tools.kali.org/vulnerability-analysis/bed, before 2012. Accessed: 2021-01-26.

[2] Ammar A. Al-Sultan, Ghufran F. Jumaah, and Faris H. Al-Ani. Evaluation of the dispersion of nitrogen dioxide and carbon monoxide in the indoor café – case study. *Journal of Ecological Engineering*, 20(4):256–261, 2019.

[3] Syaiful Andy, Budi Rahardjo, and Bagus Hanindhito. Attack scenarios and security analysis of mqtt communication protocol in iot system. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 1–6. IEEE, 2017.

[4] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A model-based API tester for event-driven systems. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, volume 8244 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2013.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[6] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[7] John Bellardo and Stefan Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *USENIX security symposium*, volume 12, pages 2–2. Washington DC, 2003.

[8] Tiago Boldt Sousa. *Engineering Software for the Cloud: A Pattern Language*. PhD thesis, Faculty of Engineering, University of Porto, 2020.

[9] Anil Chacko and Thaier Hayajneh. Security and privacy issues with iot in healthcare. *EAI Endorsed Trans. Pervasive Health Technol.*, 4(14):e2, 2018.

[10] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[11] Chaos Engineering Community. Principles of chaos engineering. Available at https://principlesofchaos.org/, May 2018. Accessed: 2021-01-21.

[12] S. De Vito, E. Massera, M. Piga, L. Martinotto, and G. Di Francia. On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. *Sensors and Actuators B: Chemical*, 129(2):750–757, 2008.

[13] Jyoti Deogirikar and Amarsinh Vidhate. Security attacks in iot: A survey. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 32–37. IEEE, 2017.

[14] Ganesh Devarajan. Unraveling scada protocols: Using sulley fuzzer. In *Defcon 15 Hacking Conf*, 2007.

[15] J. P. Dias, F. Couto, A. C. R. Paiva, and H. S. Ferreira. A brief overview of existing tools for testing the internet-of-things. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 104–109, 2018.

[16] Joao Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. Testing and deployment patterns for the internet-of-things. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, New York, NY, USA, 2019. Association for Computing Machinery.

[17] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. Visual self-healing modelling for reliable internet-of-things systems. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, pages 357–370, Cham, 2020. Springer International Publishing.

[18] João Pedro Dias, José Pedro Pinto, and José Magalhães Cruz. A Hands-on Approach on Botnets for Behavior Exploration. In *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*, pages 463–469. SCITEPRESS - Science and Technology Publications, 2017.

[19] Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Empowering visual internet-of-things mashups with self-healing capabilities. In *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*, 2021.

[20] João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. A pattern-language for self-healing internet-of-things systems. In *EuroPLoP '20: European Conference on Pattern Languages of Programs 2020, Virtual Event, Germany, 1-4 July, 2020*, pages 25:1–25:17. ACM, 2020.

[21] Martin Fowler. Phoenixserver. Available at https://martinfowler.com/bliki/PhoenixServer.html, July 2012. Accessed: 2021-01-21.

[22] Alan G. Ganek and Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, 2003.

[23] Google. AFL american fuzzy lop. Available at https://github.com/google/AFL, November 2013. Accessed: 2021-01-29.

[24] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS International Journal of Computer Science and Network Security*, 10(8):239, August 2010.

[25] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner. Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks. In *2015 Euromicro Conference on Digital System Design*, pages 530–533. IEEE, 2015.

[26] Hugo Jernberg, Per Runeson, and Emelie Engström. Getting started with chaos engineering - design of an implementation framework in practice. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, pages 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Zahra Kazemi, David Hely, Mahdi Fazeli, and Vincent Beroulle. A review on evaluation and configuration of fault injection attack instruments to design attack resistant mcu-based iot applications. *Electronics*, 9(7):1153, 2020.

[28] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[29] Xinyao Liu, Baojiang Cui, Junsong Fu, and Jinxin Ma. Hfuzz: Towards automatic fuzzing testing of nb-iot core network protocols implementations. *Future Generation Computer Systems*, 108:390–400, 2020.

[30] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[31] Samuel J. Moore, Chris D. Nugent, Shuai Zhang, and Ian Cleland. Iot reliability: a review leading to 5 key research directions. *CCF Transactions on Pervasive Computing and Interaction*, 2(3):147–163, 2020.

[32] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3), February 2016.

[33] Netflix. Chaos monkey. Available at https://github.com/Netflix/chaosmonkey, 2011. Accessed: 2021-01-22.

[34] Charles Nwatu and Aaron Rinehart. Security chaos engineering: A new paradigm for cybersecurity. Available at https://opensource.com/article/18/1/new-paradigm-cybersecurity, January 2018. Accessed: 2021-01-23.

[35] Optum. Chaoslingr: Introducing security into chaos testing. Available at https://github.com/Optum/ChaoSlingr, September 2017. Accessed: 2021-01-22.

[36] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[37] D. Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8, October 2018.

[38] Aaron Portnoy and Pedram Amini. Sulley - a pure-python fully automated and unattended fuzzing framework. Available at https://github.com/OpenRCE/sulley, 2006. Accessed: 2021-01-26.

[39] Aaron Portnoy, Pedram Amini, and Ryan Sears. boofuzz - a fork and successor of the sulley fuzzing framework. Available at https://github.com/jtpereyda/boofuzz, around 2016. Accessed: 2021-01-26.

[40] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. Patterns for Things that Fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs*, PLoP '17. ACM - Association for Computing Machinery, 2017.

[41] Aaron Rinehart. The case for security chaos engineering. In Casey Rosenthal and Nora Jones, editors, *Chaos Engineering: System Resiliency in Practice*, chapter 20, pages 249–260. O'Reilly Media, 2020.

[42] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. *Chaos Engineering: Building Confidence in System Behavior through Experiments*. O'Reilly Media, Incorporated, 2020.

[43] Francoise Sailhan, Thierry Delot, Animesh Pathak, Aymeric Puech, and Matthieu Roy. Dependable sensor networks. In *Atelier sur la GEstion des Donn? es dans les Syst? mes d'Information Pervasifs (GEDSIP) au sein de la conf? rence INFormatique des ORganisations et Syst? mes d? Information et de D? cision (INFORSID)*, page 6, 2010.

[44] Mozilla Security. Peach. Available at https://github.com/MozillaSecurity/peach, 2015. Accessed: 2021-01-29.

[45] Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Visually-defined real-time orchestration of iot systems. In *Proceedings of the 17th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MOBIQUITOUS 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. A review on visual programming for distributed computation in iot. In *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021.

[47] Sean W. Smith. *The internet of risky things: trusting the devices that surround us*. O'Reilly, 2017.

[48] Danny Soares, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Programming iot-spaces: A user-survey on home automation rules. In *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021.

[49] John A. Stankovic. Research directions for the internet of things. *IEEE Internet Things Journal*, 1(1):3–9, 2014.

[50] A. Sun, W. Gong, R. Shea, and J. Liu. A castle of glass: Leaky iot appliances in modern smart homes. *IEEE Wireless Communications*, 25(6):32–37, 2018.

[51] Kennedy A. Torkura. From dependability to resilience → security chaos engineering for cloud services. Available at https://run2obtain.medium.com/9c6d6d152ed2, November 2019. Accessed: 2021-01-24.

[52] Kennedy A. Torkura, Muhammad I. H. Sukmana, Feng Cheng, and Christoph Meinel. Cloudstrike: Chaos engineering for security and resiliency in cloud infrastructure. *IEEE Access*, 8:123044–123060, 2020.

[53] Ivan Vaccari, Maurizio Aiello, and Enrico Cambiaso. Slowite, a novel denial of service attack affecting MQTT. *Sensors*, 20(10):2932, 2020.

[54] Aron Warren. Using sulley to protocol fuzz for linux software vulnerabilities. Available at https://www.sans.org/reading-room/whitepapers/testing/sulley-protocol-fuzz-linux-software-vulnerabilities-36907, April 2016. Accesed: 2021-01-29.

[55] Wireghoul. Doona - network based protocol fuzzer. Available at https://github.com/wireghoul/doona, 2012. Accessed: 2021-01-26.

[56] Wireghoul. Doona: Network fuzzer, bed fork. Available at https://tools.kali.org/vulnerability-analysis/doona, 2012. Accessed: 2021-01-26.

[57] Jun Yoneyama, Cyrille Artho, Yoshinori Tanabe, and Masami Hagiya. Model-based network fault injection for iot protocols. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019*, pages 201–209. INSTICC, SciTePress, 2019.

[58] David Zooker Zabib, Maoz Vizentovski, Alexander Fish, Osnat Keren, and Yoav Weizman. Vulnerability of secured iot memory against localized back side laser fault injection. In *2017 Seventh International Conference on Emerging Security Technologies (EST)*, pages 7–11. IEEE, 2017.

[59] Youmin Zhang and Jin Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annu. Rev. Control.*, 32(2):229–252, 2008.

[60] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, Santa Clara, CA, August 2019. USENIX Association.