FAULTYNET: A FAULT-INJECTION TOOL SET FOR DEVELOPMENT IN SDNS

ANTONIO CARLOS DIMEO

Faultynet: Ein Fehlerinjektionswerkzeugkasten für Entwicklung in SDNs

Internet-Technologien und Softwarization

07.02.2024

Supervised by Prof. Dr. Holger Karl

Antonio Carlos Dimeo: *Faultynet: A fault-injection tool set for development in SDNs,* Faultynet: Ein Fehlerinjektionswerkzeugkasten für Entwicklung in SDNs, © 07.02.2024

Authors need stories more than stories need them
— Joe Pug

ABSTRACT

Network emulators currently have minimal capabilities of emulating network fault conditions, and fault injection frameworks have limited capabilities of emulating networks in a scaleable way. This makes it hard to test how software that is deployed onto networks handles different types of errors and outages. In this thesis, I present both a concept for a fault injection framework based on network emulation and a prototypical implementation called Faulynet, which is based on Mininet, and offers both a turnkey solution for injecting faults into Mininet scenarios, and a framework which users can use to implement their own fault scheduling algorithms.

ZUSAMMENFASSUNG

Nezwerkemulatoren haben aktuell nur minimale Fähigkeiten um fehlerhafte Netzwerke zu emulieren, und Fehlerinjektionsframeworks haben nur minimale Fähigkeiten um Netzwerke skalierbar auszuführen. Zusammen erschweren diese Bedingungen das Testen von Software in Netzwerken unter Fehlerkonditionen. In dieser Abschlussarbeit präsentiere ich sowohl ein Konzept für ein Fehlerinjektionsframework auf Basis eines Netzwerkemulators, sowie eine prototypische Implementierung des selben, Faultynet, welche auf Mininet aufbaut, und sowohl eine einfach nutzbare Möglichkeit anbietet Fehler in Mininet Netzwerke zu injizieren, als auch um eigene Fehlerterminierungsalgorithmen zu entwickeln.

ACKNOWLEDGMENTS

There are many people who have made this work possible. I want to thank Professor Holger Karl and Valentin Kirchner for their supervision, and the Friedrich-Naumann-Stiftung für die Freiheit for their continued financial support during my studies. I am indebted to my family and friends, who kept my mind on-topic when I wanted it to and off-topic when I needed it to, supported me with last-minute reviews, and in and in general supported a lifestyle during these last few months which frankly isn't sustainable.
I'll answer the messages now, I promise!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

VM Virtual Machine

NFV Network Functions Virtualization

SDN Software-defined Network

GLOSSARY

cgroup Linux kernel feature, used to limit access to system resources. 21, 34, 37

CI/CD Continuous Integration/Continuous Deployment. Practices that include automated incremental testing on dedicated hardware, among others. 4, 10, 15, 17, 18, 20, 49

fault expressiveness Measure of how varied and configurable fault types are. 8, 10, 11, 18, 20, 40, 42, 43, 45, 47, 49

fog computing The combination of devices that run centrally within data centers, often referred to as the cloud, with devices that run decentralized, so called edge devices, to provide services. 9, 10

host system The system that runs a Mininet scenario. Not to be confused with host, which is a specific type of Mininet node. One host system can run many hosts.. 21, 33, 34, 36

link-based fault Fault that affects an interface, and therefore a link, like packet loss, bandwidth limitation, or increased latency. 11, 26, 28, 32, 42

multi-fault Multiple faults that affect one interface at the same time, like a combination of packet loss, bandwidth limitation, or increased latency. 11, 13, 15, 43, 44

netem See tc-netem. 42, 43

node-based fault Fault that affects a node, like CPU stress. 33, 34, 43

qdisc tc internal, short for queueing discipline. Can be attached to an interface and applies rules to selected traffic from that interface. 33, 42, 47

tc Linux utility to configure traffic control (shaping, scheduling, policing, dropping) in the Linux kernel. 11, 32, 33, 40, 42, 43, 47

tc-netem Queue that can be attached to interfaces with tc. Simulates network characteristics like delay, packet loss, duplication. 11, 13, 32, 33, 42, 43, 47

1 INTRODUCTION

As anyone who has created large-scale software can attest, building a reliable large-scale system is no easy feat. The reasons for this are manifold, as are the suggested remedies. However, one aspect is that complex systems, by their nature, come with complex interactions between system components, which can be hard to identify and understand. In the world of SDNs, formal verification methods historically focus on modelling single applications[18], which means that interference between different applications needs to be observed through testing.

However, testing SDNs at scale comes with its own challenges since hardware test beds are unaffordable and inflexible, and Virtual Machine (VM)-based testbeds are hard to scale.[19]

A potential solution to this dilemma is using network emulators, which allow researchers and testers to run and test both single virtualized network functions and so-called service function chains in large virtual networks by running multiple switches and hosts on a small number of physical machines, often just a single one. This massively reduces the overhead cost of tests and changes many of the required setup steps to configuration management tasks, simplifying both the process of initially setting up a network and the overhead of reproducing or sharing an existing one.

Since network emulators require limited resources to run and, in some cases, require only features that are present in any recent Linux distribution, they can also be used on personal systems, further lowering the barrier to testing SDNs constantly and in different variations.

This does not mean that a network-emulator-based approach comes with no disadvantages. Their ability to run on consumer hardware means that some behaviours, like latency, won't reflect the circumstances the SDN will have to face in production.

While testing SDNs in emulators is often a good enough approximation of behaviour in the real world, there are some circumstances where emulator behaviour deviates significantly from what can happen in the real world.

One such example is the sudden presence of faults within the network. While real-world infrastructure will face a variety of faults, [13] short-running tests on consumer systems usually won't face such issues.

This can lead to a blind spot in testing, where the behaviour of the system under faulty conditions is not tested for and is potentially

unknown. This, in turn, can lead to negative outcomes in the real world.

With this thesis, I try to remediate this blind spot for a specific type of faults, those present on network links, by creating Faultynet, a network emulator which allows the injection of various faults into a virtual network, which can be seamlessly used for both manual hands-on testing sessions, and automated CI/CD pipelines.

The rest of this thesis is structured as follows:

Chapter 2 gives a general introduction to the topic, defines base requirements for fault injection tools in an SDN context, and contrasts my approach with the related but not overlapping chaos engineering term.

Chapter 3 summarizes specifically related works and analyses them based on both their fulfilment of the previously defined base requirements and their fault expressiveness.

Chapter 4 introduces the design objectives for Faultynet before introducing both a potential design for Faultynet and how that design was adapted into a prototype.

Chapter 5 discusses implementation details and decisions, as well as trade-offs.

Chapter 6 evaluates whether Faultynet achieves the previously introduced design objectives and formulates into which directions Faultynet could be expanded and what challenges such expansions might face before discussing more general future works that could build upon Faultynet.

Chapter 7 concludes this thesis.

2 BACKGROUND

2.1 MOTIVATION

In general, high software reliability, meaning "the probability of failure-free software operation for a specified period of time in a specified environment"[21], is a positive property in systems, especially distributed ones. While the literature appears to use the terms "fault" and "failure" inconsistently, I use the definition used by Fonseca and Mota[12], whereas "A failure occurs when the network is not able to correctly deliver a service, whereas a fault is the root cause of a failure".

In the case of SDNs, researchers are exploring at least three different approaches for making SDNs more reliable.

Reliability can be achieved by detecting and recovering from failures more quickly [12] , minimizing the splash radius of any single failure through smarter provisioning algorithms [11][31][28] , and by building more reliable software in general, e.g. through the usage of dedicated test beds [26][29] or the usage of automated testing and validation methods. [26] [28] [23] [29]

Ideally, such approaches should be verified experimentally.

Making such experiments repeatable, comparable, and work-unintensive requires a simple way of creating and running networks, injecting something potentially failure-causing into those networks, and observing how the system under test handles that something.

In principle, this approach goes back to at least 1989, [2] and using a similar fault injection approach for SDNs was proposed by Chang et al. in 2015 with *Chaos Monkey* [5], a concept which was intended to systematically inject failures into SDNs in real-time at post-deployment, emphasizing network failures. However, Chang et al.'s proposal never included a publicly accessible implementation.

Following the *Chaos Monkey* proposal with a practical implementation and translating the fault-injection concept into the reality of SDNs requires awareness of some domain differences and peculiarities: Cotroneo et al.[6] identifies virtual and physical storage, network, and memory in its fault model for Network Functions Virtualization (NFV), and classifies possible faults further by type, specifically unavailability, delay, and corruption. Bhardwaj et al.[3] observe that bugs in the realm of SDNs are caused by configuration, external libraries, network events, and hardware reboots. Put plainly, there are a lot of potential causes for errors in SDNs. Since it is not feasible to treat all these potential causes for failures in the course of a thesis, I

decided to focus primarily on failures in SDNs, which could be caused by faults in the network.

Thus, the starting point of this thesis is "What low-friction tools are there for a researcher who wants to observe the impact of suboptimal network conditions on an SDNs?"

2.2 BASE REQUIREMENT ANALYSIS

I believe that a low-friction tool for evaluating the impact of suboptimal conditions needs to cover a number of requirements.

1. *SDN Compatibility*. The tool needs to be able to deploy networks with multiple nodes that can communicate with each other and support switches and, potentially, controllers. While testing singular nodes might yield results, emergent behaviour makes those results difficult to extrapolate. Peuster and Karl[22] argue that performance measurements of chained functions are more representative than those obtained when measuring a single node, which leads me to believe that measurements on reliability can also benefit from end-to-end tests.

2. *CI/CD Pipeline Compatibility*. The tool needs to be compatible with automated testing pipelines. Testing setups manually in handcrafted environments severely limits the reproducibility of tests and adds unwanted overhead, both for infrastructure costs and for how much time the tester needs to invest for each test. This also means that it must not rely on resource-hungry technologies since these may allow for scaleable tests in theory but not in the reality of resource-limited research.

3. *Configurable Faults*. The tool needs to be able to inject faults into the network. While faults may occur organically, an organic approach is neither repeatable nor time-effective. Additionally, the faults need to be configurable, and injection needs to be automatic. A GUI click-ops approach is neither repeatable nor time-efficient.

4. *Local compatibility*. The tool needs to be able to run on a local machine for localized tests. While testing frameworks without this property can and do exist, they add a noticeable amount of friction, which this thesis tries to avoid.

5. *Public availability*. A tool which isn't publicly available can't be used.

2.3 DEMARCATION FROM CHAOS ENGINEERING

Chang et al.'s Chaos Monkey paper explicitly aligns itself with the concept of Chaos Engineering. Chaos Engineering is generally defined as a series of experiments which consists of four steps: First, testers define a known-good "steady state", as defined by, e.g. a system's output. Second, testers assume that a specific change won't affect the steady state. Third, they implement that change, e.g., by shutting down a node. Fourth, they try to disprove their original assumption that the change won't impact the steady state by evaluating the system's new output. [20]

The concept traces back to a blog post from Netflix from 2011,[4] which describes the process in less precise, but perhaps more understandable terms: There's a simulated monkey in our data center. It randomly breaks stuff. We assume that that won't impact our services since our services are built to recover from stuff randomly breaking. If services do break, then our hypothesis is disproven, but at least during business hours and not at three AM during a holiday.

The foundations of this thesis are definitely philosophically aligned with some aspects of Chaos engineering: Both share the assumption that injecting faults into complex systems can lead to uncovering previously unknown behaviour and that automating fault injections to unearth that unknown behaviour in a more seamless fashion is a worthwhile endeavour.

However, there are also some clear differences: The Book Chaos Engineering: System Resiliency in Practice[24] states that chaos engineering is often misunderstood as identifying issues by breaking things in production, whereas the actual approach can be more accurately described as identifying issues when trying to fix things in production, with a focus beyond reliability, into detectability and processes.

In contrast, this thesis will focus primarily on a framework that simplifies the automated, seamless injection of network faults during testing. The reasons for this divergence are twofold. First, during my literature research, I identified what I believe to be a lack of such tooling.

Second, my literature review also identified close to no works with a focus on Chaos Engineering in the context of SDNs that were published after 2015. While the reasons for this lack of work could be manifold, it leads me to believe that the merits of using automated fault injections in this context aren't properly explored yet. I believe that an approach that focuses on automated fault injections during testing is better suited to explore those merits and to build trust in the general approach of using fault injections to experimentally uncover hidden complexities in SDNs.

3 RELATED WORK

3.1 FAULT INJECTION FRAMEWORKS

Based on the base requirement analysis, I tried to identify fault injection frameworks that fit the previously defined base criteria. My search happened primarily via Google Scholar in mid-2023. Terms used for this search were "SDN Fault injection", "SDN Chaos", "SDN Mano Chaos", "VNF Fault injection", "VNF Testing Chaos", "NFV resiliency testing", "SFC Testing Fault Injection", "MANO fault injection", "Mininet Fault injection","Mininet Chaos", "Emufog Chaos", "Emufog Fault injection" and "mockfog". For each of these terms, I generally evaluated the first five to eight pages of Google Scholar results, more if results appeared promising and less if results overlapped completely with previously seen works.

In addition, I reviewed Google Scholar for papers released in or after 2022 that came up when using the keywords "VNF testing emulation literature review", "vnf development emulation literature review", "VNF testing emulation state of the art", "vnf development emulation state of the art", "vnf testing", "vnf reliability", "vnf reliability testing" and "sdn testbed chaos", and reviewed all papers available on google scholar citing the papers [5][25].

The oldest fault injection framework focusing on networks dates back to 2015 and is called *Armageddon*[25] Armageddon was proposed as a controller-agnostic framework for introducing faults into networks. A prototypical implementation for OpenVirteX, a network hypervisor, is discussed in the paper, but I have been unable to find a publicly accessible copy of this prototype. Beyond conceptual novelty, Armageddon's main draw lies in the introduction of "GREEDY KILLER", an algorithm that calculates injection targets while trying to both maximize coverage and preserve network-wide invariants. The Armageddon paper mentions several different possible failure scenarios but does not explore them in detail and mostly discusses failing links when talking about Armageddon's properties, leading me to believe that other types of failures weren't explored in detail. This, combined with Armageddon's lack of configurabillity, and missing public implementation, greatly limits its usefulness for researchers.

Du et al. [9] propose a high-level architecture for combining existing measurement and fault-injection tools via microservices but do not implement any tools by themselves. This reliance on off-the-shelf tools makes knowledge about possible monitoring, orchestration, and fault injection tools a prerequisite for the usage of their ap-

proach. An additional weakness is that their tool does not handle any of the required deployment work for the system under test. In their case study, their tool interacts with a previously set-up network running on OpenStack. This limits their usefulness for automated testing within a pipeline since multiple tests may run at the same time, and access to testing servers may be limited.

This focus on deployments of OpenStack appears to be common and is shared by several other tools:

ThorFI, [7], a hypervisor-level fault injection framework, allows for fault injections into networks hosted on OpenStack. The faults to be injected are defined via a REST API, simplifying the integration of user-developed fault schedulers. ThorFI's code base is openly accessible, simplifying both the process of expanding ThorFi where needed and simplifying the integration work of fault schedulers. Unlike many other papers, ThorFI allows for the injection of faults with different timing patterns, including bursts and degradation, and supports a relatively large variety of faults. While ThorFI expects its target network to be hosted on an OpenStack instance, it is specifically designed to affect only the target tenant, which means that multiple networks can be tested on one OpenStack instance at the same time. However, even setting up just one OpenStack instance may require a significant amount of effort. A current limitation of this approach is that ThorFI does not come with pre-defined failure scenarios, increasing the work that is required before a setup can be tested.

Another fault injection framework with a focus on OpenStack is *FT Analyzer*.[17] FT Analyzer attempts to make its tests less resource-intensive by creating a stripped-down duplicate environment of an OpenStack topology and running its test on that copy. Its main limitation is a complete lack of fault expressiveness. The only fault type considered is the shutdown of servers and services, severely limiting its applicability for testing the impact of network-based faults.

HAVerifier [10] also operates on OpenStack deployments. HAVerifier is scenario-driven, meaning that users define custom scenarios, including fault injection and recovery steps and SLA measurements to be taken during injection time. The scenarios appear to rely exclusively on timers. HAVerifier's approach towards injecting faults relies on a collection of tiny bash scripts, each of which can be called to inject a specific fault. A cursory review of a number of these bash scripts leads me to the impression that most are single-line wrappers around varying different tools. This makes it relatively simple to extend what types of faults HAVerifier supports, but it does mean that potential incompatibilities between different failures can't be handled by the framework itself. The source code to HAVerifier is publicly accessible.

MockFog [16][14][15] is a project that comes from a fog computing background, and is one of the few options for frameworks that don't

use OpenStack. Fog computing in this context is the combination of devices that run centrally within data centers, often referred to as the cloud, with devices that run decentralized, so-called edge devices, to provide services.[8] Instead, Mockfog tries to emulate a network in the cloud based on a network configuration defined by the developer. Each emulated host runs on a separate VM, which is configured to mimic that host's connection- and computation characteristics. Applications that run on the emulated network need to be dockerized, and faults are injected based on a state machine that developers define in a JSONC (sic!) configuration file. While this setup allows for great scalability, it also makes large-scale experiments potentially expensive since each emulated host requires a running VM. Applications are also statically mapped onto machines, and there appears to be no mechanism to integrate an SDN controller with the setup, a limitation which is explained by Mockfog's focus on fog application testing and benchmarking. Like ThorFI, Mockfog is relatively expressive with regard to its faults and is one of the few fault injection frameworks that allows the combination of multiple faults on one interface, albeit with some significant limitations in place. This is discussed in detail in subsection 3.2.2.3. Multiple implementations of Mockfog exist. When discussing Mockfog, I refer to [15], which is the latest version.

Fogify [27] also has a fog computing background but suggests yet a different approach. Instead of creating each of its nodes in a VM, it runs nodes by extending the configuration options of docker-compose and emulating them using docker-swarm Users can define network and node properties such as memory constraints, latency or packet loss, as well as actions, which change these properties based on a timer. This approach leads to a much simpler initial setup than frameworks which rely on OpenStack and reduces the overhead required to run a network massively. However, since the Fogify orchestrator is docker-swarm, Fogify does not allow the evaluation of more dynamic network setups or controllers. In addition, faults in Fogify can only be scheduled by writing the fault schedule into a configuration file. Unlike ThorFi, Fogify offers neither an external API that can be used for scheduling nor supports more complex fault patterns, severely limiting the complexity of fault combinations that can be injected with a reasonable level of effort.

Finally, Vilchez and Sarmiento [30] create a custom emulation environment based on bash scripts, stating that network emulation tools like Mininet hide underlying technologies and only allow for limited configuration commands. Faults appear to be injected via GUI operations and are limited to changing bandwidth or severing links, which massively limits what types of tests can be performed. I have been unable to identify a publicly accessible version of this tool.

Framework	SDN Compatibility	CI/CD Pipeline Compatibility	Configurable Faults	Public Availability	Local Compatibility
Armageddon[25]	Yes	No	No	Unknown	No
Du et al.[9]	Yes	No	Custom	Unknown	No
ThorFI[7]	Yes	Yes	Yes	Yes[a]	No
FT Analyzer[17]	Yes	Yes	No	Unknown	No
HAVerifier[10]	Yes	No	Yes	Yes[b]	No
Mockfog[15]	No	Yes	Yes	Yes[c]	No
Fogify[27]	No	Yes	Yes	Yes[d]	Yes
Vilchez and Sarmiento[30]	Yes	Yes	No	Unknown	Yes

a https://github.com/dessertlab/thorfi
b https://github.com/XLab-Tongji/HAverifier
c https://github.com/OpenFogStack/MockFog2
d https://github.com/UCY-LINC-LAB/fogify

Table 3.1: Existing fault injection frameworks compared with requirements

3.2 COMPATIBILITY ANALYSIS

3.2.1 *Base requirement compatibility*

My review of the related literature has identified a number of tools which could potentially be used. These tools are compared with the requirements defined in 2.2 in table 3.1.

What stands out in this comparison is that no tool fulfils all the requirements defined in 2.2. Tools with SDN compatibility achieve this compatibility by interacting with OpenStack deployments, which automatically limits their local compatibility. Additionally, while ThorFI does explicitly mention the issue of handling multiple tenants on one OpenStack host, other OpenStack-based tools don't, which may limit their CI/CD pipeline compatibility. Those tools that don't rely on OpenStack either come from a fog computing background and don't support SDNs or are severely hampered in their fault expressiveness.

3.2.2 *Fault Expressiveness*

While the evaluation of SDN compatibility, CI/CD pipeline compatibility, and public availability is relatively uniform, within the realm of configurable faults, there is a much greater degree of variability. This leads me to the term of fault expressiveness, which broadly refers to both how varied and how configurable fault types are. Greater fault

expressiveness can be achieved through the support of more types of faults, more configurable faults, and more configurable fault injection schedules, among others.

This section compares the fault expressiveness of the various tools I have identified during my literature research. Excluded from this comparison are Du et al. due to their exclusive reliance on external fault injection tools, FT Analyzer due to its exclusive focus on node shutdowns, and Vilchez and Sarmiento due to their fault types being limited to bandwidth and changing links, in combination with GUI operations being the only way to inject faults.

This section will give a detailed comparison of the fault expressiveness of the other works. This evaluation is based on both the relevant paper and the underlying code, except for Armageddon, since I was unable to identify a repository containing the code.

3.2.2.1 *Fault types*

Most injection frameworks support similar fault types: Except for Armageddon, which focuses on packet loss only, all frameworks support a variety of link-based faults, meaning faults that affect a link between two network interfaces and usually overlap in the specific types of faults offered. For all cases I have evaluated, they are injected by modifying the configuration of the attached interface. Most frameworks use the tc utility, and specifically the tc-netem function, under the hood, again except for Armageddon, whose usage of tc I could neither verify nor disprove due to a lack of source code. This leads to two possible explanations for the overlap in techniques: The first is that those fault types are the most relevant types to emulate, causing tc-netem and the frameworks to independently decide to support those faults.

The alternative is a causal explanation: Since tc-netem already supports injecting faults regarding bandwidth, reordering, duplication, latency, and packet loss, frameworks decide to support these fault types since they are easy to implement. This may explain why none of the frameworks supports redirection faults since these need to be implemented using other tc functionality.

HAVerifier is a special case, both regarding the fault types it supports and in many aspects of its fault expressiveness. Unlike the other injection frameworks, HAVerifier doesn't natively implement any faults but relies on minimalist bash scripts, which are executed by the framework to inject faults. This makes HAVerifier hard to categorize since the supported fault types, multi-fault compatibility, and protocol- and application-specific faults depend on the module the user selects.

The great advantage of HAVerifier's approach is its great extendability. The answer to the question "Can HAVerifier do X?" is usually

Framework	Fault types	Fault patterns	Multi-faults	Application specific faults	Fault scheduling	Protocol specific faults
ThorFI	Bandwidth, Latency, Loss, Reorder, Duplication, Host reboot, Interface down	Degradation, Burst, Persistent	No	No	User triggered, Timer	Yes
Mockfog 2.0	Bandwidth, Latency, Loss, Reorder, Dispersion, Corruption, Duplication, Memory and CPU limit	Persistent	Yes, grouped	No	State machine	Yes
HAVerifier	Diverse, Module dependent	Persistent	Module dependent	Module dependent	Timer	Module dependent
Fogify	Latency, Bandwidth, Loss, CPU stress, Arbitrary commands, Firewall rules, Interface down	Persistent	Yes, grouped	No	User triggered	No
Armageddon	Loss	Persistent	No	No	Computed	No

Table 3.2: Fault expressiveness of various injection frameworks compared

"Potentially, if the module exists, but I can't find it", instead of a plain no. However, there are strong trade-offs.

While HAVerifier comes with a library of already implemented modules, these are not documented and not easily discoverable, potentially leading to duplicate development efforts. Potentially more impactful, the reliance on simple shell scripts exposes core aspects of the fault injection to users and outsources the potentially most error-prone and technically complex parts of the development process to them without offering them any guardrails. It also gives the illusion of simplicity: A user of HAVerifier may see that modules for packet loss and packet corruption exist and assume that these can be easily combined when that is very much not the case.

3.2.2.2 *Fault Patterns*

ThorFI is the only fault injection framework in this evaluation that supports advanced injection patterns natively, meaning specifically a burst and degradation pattern. The burst pattern injects and ejects a fault repeatedly. The degradation allows users to automatically increase the intensity of a fault over time. While these patterns can be emulated by other frameworks, the manual effort required to do so is significant, which gives ThorFi a clear advantage in their expressiveness.

3.2.2.3 *Multi-Faults*

Multi-faults refers to multiple faults being active on the same interface at the same time. This multi-fault definition does explicitly not include multiple faults being injected into a node or multiple faults being injected at the same time on different interfaces since both of these are trivial, whereas having multiple faults active on one interface is comparatively complex, as I discuss in detail in 6.2.1.1.

The frameworks under evaluation have two different approaches to dealing with this issue: The first is not dealing with it at all and not supporting the injection of multiple faults on the same link. This approach is chosen by ThorFi, Armageddon, and the HAVerifier modules I have evaluated. While this leads to a simpler implementation, it does limit the expressiveness of faults.

The second approach is chosen by Mockfog and Fogify. For this approach, multiple faults of different types are combined into a single expression. This approach works because tc-netem supports multiple different types of faults, which can be active on a single tc-netem instance at the same time, which greatly simplifies the implementation of multi-faults. While this approach is relatively simple to implement, it effectively groups the injected faults into one, which means that, e.g. packet loss and latency are enabled and disabled at the same time. As a result, this approach is not suited to inject additional faults into

an already faulty interface, at least not as currently implemented in Mockfog and Fogify.

3.2.2.4 *Application specific faults*

The only framework that supports application-specific faults is HAVerifier. Its library currently implements operations for RabbitMQ, Swift, Keystone, and Neutron, among others.

3.2.2.5 *Fault scheduling*

Fault scheduling mechanisms are perhaps where I see the greatest variety between the different frameworks.

Fogify uses timestamped scenarios, which initiate an injection at a specific point in time, as defined in a configuration file.

ThorFi uses a similar approach and allows users to define pre- and post-timers for faults, as well as the fault duration. Since ThorFis faults are activated by a REST API it can be integrated into other scheduling engines with relative ease.

The HAVerifier paper introduces a test case template in which specific testing steps should be defined, but the HAVerifier framework appears to exclusively use timers to activate and deactivate faults, except for faults that do not rely on an injection time to have an impact, like the removal of a file.

Mockfog is more expressive and relies on a state machine that users define in a configuration file. Beyond time-based state transitions, it also supports state changes based on messages, which can be generated when, for example, certain CPU usage thresholds are reached. This allows users to define test cases beyond what would be possible with purely time-based restrictions.

Armageddon chooses yet another approach: Instead of allowing users to define faults by themselves, Armageddon relies on an algorithm which computes where and when faults should be injected. While less configurable, this automatic computation minimizes the effort required before users can actively use the framework. Additionally, deciding where to inject faults computationally allows for potentially greater coverage since faults can be reactively injected into where they produce the most interesting results.

3.2.2.6 *Protocol specific faults*

Link faults can configured to affect only specific protocols or be limited to traffic on certain ports. Such functionality is supported by Mockfog and ThorFI. In ThorFI, the target protocols can be indicated via arguments in a REST call, and lead to the creation of a filter, which limits the effect of an underlying rule to a single protocol. Under the hood, Mockfog works the same, but the conditions for a targeted fault are created differently than in ThorFI since Mockfog doesn't rely on

REST calls to modify arguments and accepts a JSON file that predefines an interface configuration.

3.2.2.7 *Summary*

The different fault injection frameworks appear to prioritize fault expressiveness along a number of different dimensions.

While the categories of fault types and fault patterns appear to be relatively uniform, there is great variety in fault scheduling, with at least four different types of fault scheduling mechanisms.

None of the existing frameworks surpasses the others in all aspects of expressiveness. Instead, each has strengths in its specific dimension: ThorFI with its patterns, Mockfog with its state-based fault scheduling, Fogify and Mockfog with their multi-fault support, HAVerifier with its application-specific faults, and Armageddon with its computed fault scheduling.

3.3 SUMMARY

Based on my review of the related works, I can summarize two key findings:

1. Currently, no framework exists which allows for scaleable fault testing of **SND!**s (**SND!**s) on both a local system and within CI/CD pipelines.

2. While ill-suited for the specific use case of scaleable testing with CI/CD integration, a number of tools exist that have interesting capabilities in related fields. All these tools have some unique aspects to them, and especially in the area of fault expressiveness, no tool is strictly more expressive than the others.

This concludes the initial research phase of this thesis.

4 HIGH LEVEL DESIGN

4.1 BASE DESIGN CONSIDERATIONS

In chapters 2 and 3, I argued that there is a lack of CI/CD pipeline and SDN-compatible configurable fault injection frameworks.

As a next step, I will introduce a design, implementation, and evaluation for a prototype that closes this gap.

This section will describe the base design. Section 4.2 will define design objectives for the prototype, and section 4.3 will discuss the resulting high-level architecture of the implementation of the fault injection framework and compare and contrast it with the idealized base design.

The prototype fault injection framework is called Faultynet and is currently publicly accessible at `https://github.com/ADimeo/Faultynet`.

4.1.1 *Deploying SDNs in a local compatible way*

One of the key aspects I identified during literature research was that most SDN-focused tools rely on the target network running in an OpenStack instance.

OpenStack is a cloud computing platform used to administer clouds. It is fully featured and far from trivial or lightweight to deploy, which is why I consider tools that exclusively rely on it to not be compatible with local usage. Additionally, in some cases, the fault injection frameworks also seemed to rely on exclusive access to the OpenStack instance, limiting CI/CD pipeline compatibility.

Any tool that wants to overcome OpenStack's limitations must decrease the work and hardware requirements for creating an SDN while staying compatible with existing SDN controllers.

One way of achieving this outcome is using an emulator and a popular[19] option for such an emulator is Mininet. Mininet creates an emulated network, including switches, on a single Linux host using various operating system features. This allows testers to emulate an SDN, which can be automatically set up and torn down with relatively little effort on relatively inexpensive hardware. While some limitations apply to running multiple Mininet instances on the same host at the same time,[1] these limitations are not inherent and can be circumvented, making Mininet an ideal candidate for creating a local and pipeline compatible SDN fault injection framework.

1 Specifically, Mininet may attempt to reuse already existing interface names when recreating an existing network

This compatibility also allows the fault injection framework to expand an existing ecosystem instead of creating a new one, thereby increasing usability and lowering the adoption barrier.

4.1.2 *Designing for configurable faults*

The relatively large variety in fault scheduling mechanisms identified in 3.2.2.5 indicates that there is a need for configurability and customizability in this dimension.

A relatively simple way of achieving these properties is creating a replaceable module responsible for controlling which faults are active. I call this module a fault controller. This fault controller is responsible for deciding when and where a fault should be injected and is, therefore, the entity responsible for the fault scheduling dimension of fault expressiveness.

Since the operations performed when injecting faults are the same regardless of how the fault controller makes its scheduling decisions, the fault injection logic can be extracted into a different module, a so-called fault injector.

Since fault patterns are also independent of scheduling decisions, the fault injector module also handles fault patterns, making it the entity responsible for the fault type, fault pattern, multi-fault, application-specific fault, and protocol-specific fault dimensions of fault expressiveness. This leads to a design where fault controllers are smart and contain all necessary logic and state to make fault scheduling decisions. In contrast, fault injectors are dumb and only aware of themselves.

4.1.3 *Achieving CI/CD compatibility*

While the ability to run SDNs at scale is necessary to achieve CI/CD pipeline compatibility, it is not sufficient since running SDNs at scale does not necessarily imply the ability to gain insights from those runs.

Specifically, I believe that the most searched-for insights will be of the nature "What network state caused my SDN to behave this way?", and its inversion, "What impact did network state X cause in my SDN?".

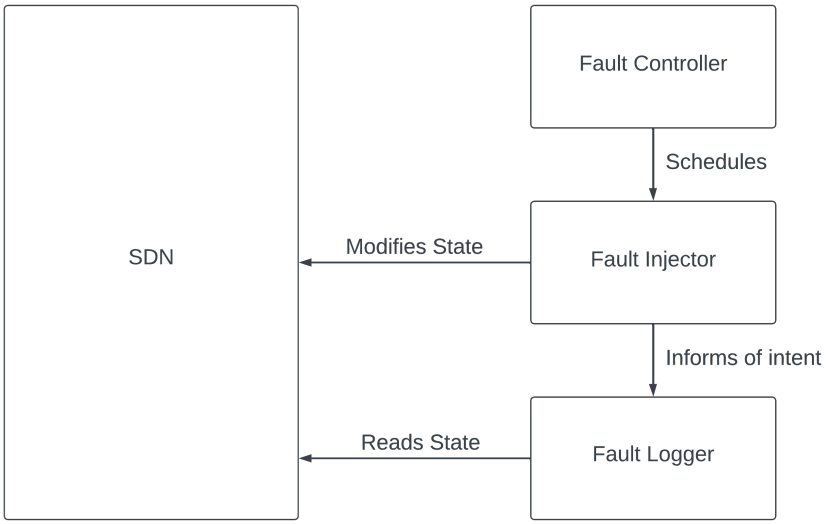
To answer those questions, testers need logs that describe the system's state at different times. In Mininet and many other software, this is not how logs behave since they log operations as they occur instead of logging the system's state at indicated times. For this reason, creating a logging system that operates independently from Mininet's default system logs makes sense. The module responsible for this is the fault logger.

Ideally, the logs generated by this fault logger should log not just system state but also intent since implementation errors may cause

system state and fault injection intent to diverge, and both are important, the former for analyzing fault impact on SDNs, the latter for verifying correct fault injection behaviour.

To track system intent, the fault controller or the fault injector must inform the fault logger of their intent. I believe having fault injectors inform the fault logger of their intent is more valuable since fault injectors can log both the commands they used for injecting a fault and the returning response provided by the operating system.

Figure 4.1: Module structure and relations of the conceptual fault injection framework



4.2 DESIGN OBJECTIVES

While the base requirements defined in 2.2 are useful to identify the lack of an SDN and pipeline-compatible fault injection framework, they are less suited to evaluate the implementation of a new prototype since they are very broad.

This section redefines those base requirements into more useful design objectives for Faultynet.

The requirement of public availability is not reflected in these design objectives since that is a given based on my publishing of the source code of Faultynet.

1. *Containernet Compatibility*. All functional Containernet scenarios should run in Faultynet without issue. This design objective supersedes the base requirement of SDN compatibility, as discussed in 4.1.1. Using Containernet instead of Mininet is motivated in 4.1.1.

2. *High Fault Expressiveness*. Faultynet should match or supersede the fault expressiveness of the tools identified in 3.1. Ideally, it should be able to combine arbitrary faults on arbitrary links in arbitrary combinations. This design objective directly results from the base requirement of configurable faults and assumes that the other tools had reasonable justifications for implementing their level of expressiveness.

3. *Pipeline and Local Compatibility*. Faultynet should be built so that it can easily be added to a CI/CD pipeline and executed as part of a pre-deployment process. Additionally, it should also be able to run on a consumer system. This design objective directly results from the base requirements for CI/CD pipeline compatibility and local compatibility.

4. *Functionality*. Faultynet should be functional, meaning it should be able to make a net faulty.

5. *Usability*. Faultynet should be simple to understand and use from a user's perspective. The work required to start injecting faults into an existing net should be minimal.

6. *Extendability*. Faultynet should be modular enough that users can easily create custom code for scheduling faults.

4.3 IMPLEMENTING THE ARCHITECTURE

To judge the feasibility of the architecture proposed in 4.1, it needs to be tested in praxis.

This section describes the prototypical implementation of the suggested architecture on top of Containernet.

4.3.1 *Mininet/Containernet base architecture*

Faultynet uses Containernet to implement its SDN emulation and is implemented as a fork of Containernet. Being a fork of Containernet, large parts of Faultynet's architecture are equivalent to Containernet's architecture, which itself is highly similar to the structure of Mininet. Since both the source code and the documentation for Mininet are publicly accessible[2] and relatively comprehensive, this chapter will briefly introduce Mininet's design for reader comfort before highlighting the newly added elements.

4.3.1.1 *Forking Containernet instead of Mininet*

The decision to fork Containernet instead of Mininet was made based on the assumption that Containernet implements some additional fea-

2 https://mininet.org/

tures which might be helpful in the context of Faultynet. While this assumption held true, the decision to fork Containernet instead of Mininet had close to no impact on the design or implementation of Faultynet since Containernet made no architectural changes to Mininet. For this reason, when describing interactions or differences between original subsystems and those introduced by Faultynet, I will usually use the Mininet moniker since those interactions would be the same had Faultynet been a direct fork of Mininet instead.

4.3.1.2 *Mininet and Containernet base architecture*

Mininet is a network emulator. As such, it emulates networks, specifically nodes, which can represent hosts, switches, or controllers, and links, represented by connected network interfaces, which connect those nodes. Each network in Mininet has a topology, defining which nodes are connected via which links. Under the hood, nodes are usually emulated by a shell process that runs in a network namespace, and in some cases, cgroup, to further limit the access to the host system's resources. Commands that are executed by that shell process have limited visibility into, e.g., network interfaces, which limits their access to the host system's resources and gives the command the impression of running on a smaller host. Containernet expands on this concept by implementing new APIs, which significantly simplify the execution of docker containers as nodes. It does not modify any aspects of Mininet's base architecture.

4.3.2 *Faultynet Additions*

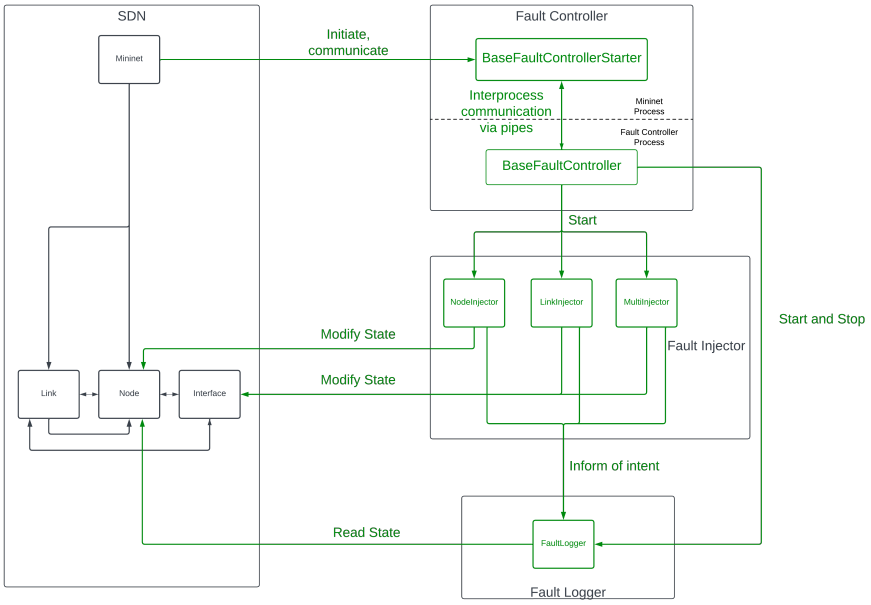
Based on the architecture introduced in 4.1, Faultynet needs to add elements that act as fault injector, fault controller, and fault logger to Mininet.

First, Faultynet implements three fault injector classes: MultiInjector, LinkInjector, and NodeInjector. Each fault injector is responsible for creating one fault on one node or interface. It accepts arguments which define that fault. These include the target, which can be either a node or a network interface, as well as the type, intensity, pattern, and duration of the fault. Fault injectors and their differences are described in detail at 4.6.

Second, Faultynet implements four different fault controllers that all inherit from a shared BaseFaultController. This BaseFaultController also offers the library methods required to simplify users' implementation of new fault controllers. All fault controllers can be configured through a YML configuration file. Fault controllers are described in detail at 4.4.

Third, Faultynet implements a fault controller starter for each fault controller, which reads a configuration file, enriches the data within it, and initiates the fault controller in a new process. The fault controller

Figure 4.2: Faultynet's additions to Mininet base architecture highlighted in green, with general responsibilities annotated



starter is also responsible for allowing Mininet to communicate with the fault controller.

Fourth and finally, the fault logger is implemented by a single FaultLogger class, which tracks both which faults are currently active and offers another option of executing arbitrary commands on nodes, this time with a focus on simplifying the user's understanding of the state of the system at runtime. The fault logger is started by the fault controller and runs within the fault controller process. It is described in more detail in 4.7.

4.3.2.1 *User interface*

From the perspective of a user who has already defined a network in Mininet, the usage of Faultynet requires three additional steps. First, users need to decide on a fault controller. Different fault controllers come with different capabilities, and each fault controller is implemented in a separate class. Second, users need to define a YML configuration file. The configuration file's contents depend on the chosen fault controller class and usually define which faults the fault controller should inject. Finally, the configuration file and the fault controller starter corresponding to the fault controller need to be passed to Mininet's `Mininet()` method or a comparable method, like Containernet's `Containernet()`.

Listing 4.1: Example configuration for a RandomLinkFaultController that randomly disables an additional link each 10 seconds, except links originating from node s1

```
---
fault_type: "link_fault:down"
pattern: "persistent"
injection_time: 10
nodes_blacklist: ['s1']
...
```

Fault controllers are automatically started by their corresponding fault controller starter when Mininet is started and may be controlled during run time if the chosen fault controller supports this functionality.

4.4 FAULT CONTROLLERS

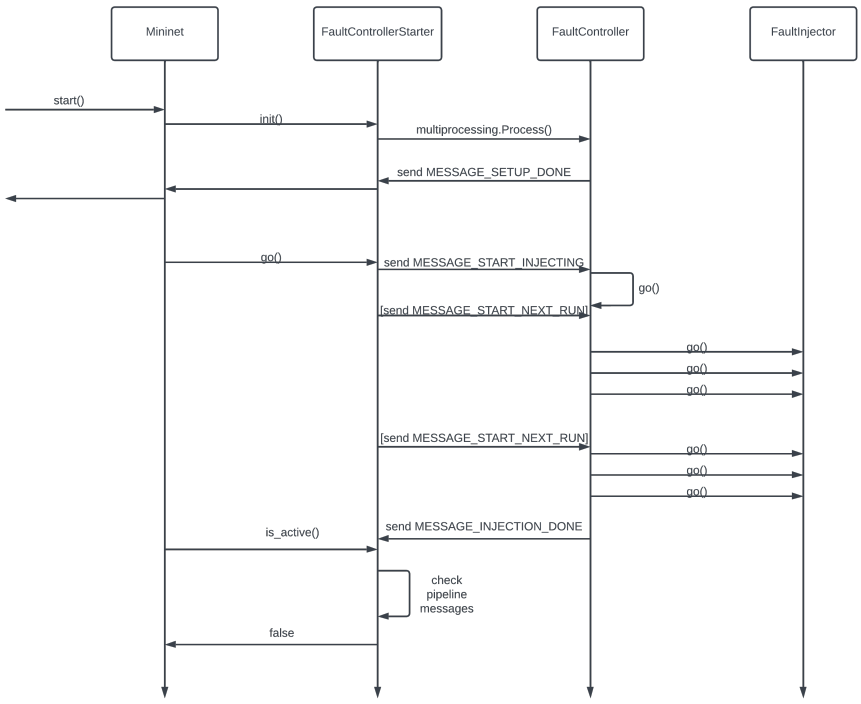
Within Faultynet, fault controllers are responsible for fault scheduling.

Faultynet implements four fault controllers. These fault controllers are:

- LogOnlyFaultController, which is a minimalist fault controller. It doesn't inject any faults but does provide easy access to Faultynet's logging system, which allows for scheduled execution of arbitrary commands on arbitrary hosts, for example, to log system state. It also serves as an implemented illustration of how a new fault controller can be implemented.
- ConfigFileFaultController, which accepts a YML configuration file containing a list of faults and injects those faults according to an internal timer. It is best suited to create repeatable fault scenarios that can be used in testing pipelines and offers limited interactivity.
- RandomLinkFaultController, which accepts one type of fault and injects this fault into random links, adding an additional randomly chosen link during each iteration. This Controller was designed for simple chaos-monkey-style tests.
- MostUsedFaultController, which accepts one type of fault and injects this fault into the link that was busiest during the last iteration. It was designed to illustrate the possibilities of injecting faults based on runtime net behaviour.

Additionally, Faultynet comes with a BaseFaultController, which is responsible for most framework-specific operations, including the automated initialization of the fault logger. BaseFaultController was specially designed to simplify the implementation of new fault controllers. BaseFaultControllers also comes with documented helper methods for some of the tasks controller-specific tasks users would need to implement while building new fault controllers.

Figure 4.3: Faultynet control flow. Sending MESSAGE_NEXT_RUN is only used in interactive controllers.



4.4.1 *Interactive and non-interactive fault controllers*

Fault controllers can be interactive or non-interactive. Non-interactive fault controllers are started once and accept no further inputs.

ConfigFileFaultController is non-interactive since all faults are defined and scheduled statically in the configuration file.

Interactive controllers support iterations. They are started once but do not start injecting faults until they receive a specific message from their fault controller starter. When the fault controller receives the message, it starts an iteration. During an iteration, the fault controller injects faults, as defined in its configuration. Once these faults are done injecting, the iteration is considered finished, and the fault controller awaits its following iteration message to start the next iteration.

Alternatively, an interactive fault controller can also receive a termination message, prematurely terminating the fault controller.

Currently, RandomLinkFaultController and MostUsedFaultController are interactive. Both also support an automatic mode, in which the end of one iteration automatically triggers the next one.

4.5 LAUNCHING FAULT CONTROLLERS

Each fault controller is launched by a corresponding fault controller starter. The fault controller starter launches the fault controller in a

separate process and holds references to the pipes that are used to communicate with the fault controller. This architecture is discussed in more detail in 5.1.

Before launching its fault controller, the fault controller starter reads the user-provided configuration file and enriches it with additional information.

For example, all user-provided configurations accept node names as injection targets. However, since the fault controller runs in a separate process, it can not map those node names to process IDs independently. Instead, the FaultControllerStarter identifies the corresponding process IDs for target nodes and adds them to the configuration that is passed to the fault controller. I call this process enrichment.

4.5.1 *Running fault controllers*

Once configured and initialized, all fault controllers can be started by calling their starters `go()` method. At this point, non-interactive fault controllers will start injecting their faults, whereas interactive fault controllers will wait for another function call before starting an iteration.

Little needs to be said on the fault scheduling decisions implemented within the controllers: ConfigFileFaultController will start already created faults, RandomLinkFaultController will choose a random link from its internal data structures and inject faults into that link, and MostUsedLinkFaultController while `ifconfig` and its internal data to identify how much traffic went over each link, and then inject faults into the busiest link.

Regarding the handling of the actual injectors, all fault controllers use a similar pattern to track their injectors:

```
async def go(self):
    ...
    fault_coroutines = []
    for i in fault_to_run:
        fault_coroutines.append(i.go()) # Faults are started here
    # All faults are running now
    await asyncio.gather(*fault_coroutines) # C
    ...
```

Injectors are put into coroutines, those are placed into a list, and the fault controller waits until all fault injector coroutines have terminated before continuing. This pattern allows fault controllers to execute multiple faults concurrently. Blocking code is not a concern since fault injector commands are not user-controlled.

For interactive controllers, this pattern would run for each iteration.

4.6 INJECTING FAULTS

Faultynet injects faults into nodes by executing commands on a node or into links by executing commands affecting the interfaces which define a link.

All faults are executed within the namespace of the process that emulates the target node. This allows the fault injector to access the same resources as the target node and allows Faultynet to run commands "on" nodes while not impacting Mininet's ability to execute commands on nodes.

Faultynet relies on the `NodeInjector` class to inject faults on nodes. `NodeInjector` supports CPU stressing and the execution of arbitrary commands.

When injecting faults into links, Faultynet primarily relies on two classes: `LinkInjector` and `MultiInjector`. However, due to its versatility, link-based faults can also be injected by using a `NodeInjector` with a custom command.

`LinkInjector` supports delay, reordering, loss, corruption, duplication, and redirection of packets, limiting the data rate of an interface, and disabling interfaces. It should be noted that the `LinkInjector` class is directly based on a similar injector from ThorFi [7][3].

`MultiInjector` supports most of the above, except for redirection and disabling interfaces. In return, `MultiInjector` can simultaneously inject multiple types of faults into a single interface.

Link faults are asymmetrical, and injecting a fault into a link requires a FaultInjector on each interface the link connects to. Instances of both `LinkInjector` and `MultiInjector` assume exclusive access to their target interface during their runtime. This means that at any time, at most one injector can be active on an interface.

It should also be noted that `MultiInjector` is currently only supported by ConfigFileFaultController and requires users to know the name of the target interface due to `MultiInjectors` late addition to Faultynet.

4.6.1 *Fault Expressiveness*

4.6.1.1 *Fault types*

Faultynet supports a variety of different fault types. These are grouped by name into those created by LinkInjector, NodeInjector, and MultiInjector, respectively. Each of these fault types can be configured further regarding their duration (including an optional wait time before and after the fault), pattern, and, if applicable, target and intensity of the fault. All faults apply to egress traffic, except for the link_fault:redirect, which redirects ingress traffic.

3 For a detailed overview of the difference, see the Faultynet source code

All fault types are listed at 5.1.

4.6.1.2 *Fault targets*

Link faults support target definitions, which restrict them to specific protocols or ports. link_fault:down is an exception since it is always applied to an entire interface, not individual packets. Node faults do not support target definitions for the same reason.

If set, a fault target limits the fault's impact to specific protocols or traffic on specific source or destination ports. Supported protocols are ICMP, IGMP, IP, TCP, UDP, IPv6, and IPv6-ICMP.

4.6.1.3 *Fault patterns*

Faults support four patterns: Persistent, random, burst, and degradation.

- *The persistent pattern* applies the indicated fault to all packets that pass through an interface. It is also the pattern used for node-based injections that do not change in intensity.
- *The random pattern* applies the indicated fault to random packets which pass through an interface. It is not supported for node-based injections, link-based bottleneck injections, and interface state injections due to a lack of semantic need. For link_fault:redirect faults, the random pattern is only implemented for faults that target all traffic on an interface due to time limits in the thesis's implementation phase.
- *The burst pattern* enables a persistent fault in bursts, meaning the repeated regular injection and ejection of faults in a timed pattern. Burst length and time between bursts can be configured.
- *The degradation pattern* enables a fault with increasing intensity. Starting intensity, ending intensity, step size, and step duration can be configured. node_fault:custom supports this mode with a Python-like replacement syntax. multi_fault does not support the degradation pattern.

Like much of the implementation for link-based faults, the concept of these timing patterns and fault targets comes from ThorFi. [7]

4.7 FAULT LOGGING

The `FaultLogger` class acts as the class fulfilling the role of the fault logger. `FaultLogger` then writes logs to a JSON file when Faultynet terminates.

The fault logger logs what faults are currently active at a set user-defined interval, e.g. every second.

Beyond logging which faults are currently active, the fault logger can also execute arbitrary commands on arbitrary nodes during each logging step and log their outputs. This allows for greater customizability of the fault logger's capabilities.

Any fault controllers which are based upon BaseFaultController automatically support the `FaultLogger` class. Like other aspects of the fault controllers, the fault logger is customizable via a configuration file, specifically the same configuration file used to configure the fault controllers. An example configuration that would log active faults every second and execute one additional command might look like this:

Listing 4.2: Example fault logger configuration

```
log:
  interval: 1000 # in ms
  commands:
      - tag: "debug-command-1"
        host: "h1"
        command: "ip link show h1-eth0"
```

This does mean that Faultynet makes users responsible for evaluating the impact of their fault injections. 6.3.3 discusses this decision in more depth.

The `FaultLogger` class works on a best-effort basis based on internal state. For a detailed discussion of what this means, see 5.4.

5 IMPLEMENTATION DETAILS

While the proposed design mostly overlaps with the implemented design, there are several details where aspects of the implemented design deviate or at least aren't directly informed by the proposed design. This chapter explores those deviations.

5.1 FAULT CONTROLLER AND FAULT CONTROLLER STARTER

In the implemented design, two different elements take the place of the fault controller: The BaseFaultController and the BaseFaultControllerStarter, or classes inheriting from each, respectively.

This separation is motivated by the need to run fault scheduling and fault injection logic independently from the rest of Mininet.

Injecting faults is inherently a parallel task: Users will want to inject and eject faults while other aspects of the system are running. Implementing this parallelism closer to Mininet, e.g. by using `asyncio`, one of the ways of writing asynchronous code within Python, means that users who build more complex or parallel systems might interact with Faultynet internals in unwanted ways. To put it in simple terms, running the fault controller and fault injectors in a separate process moves it farther away from Mininet, which makes it less likely that the underlying implementation interferes with code that users have written.

Additionally, some Mininet functions, like `cmd()`, are completely blocking when executing long-running commands, live `iperf`, and do not yield execution rights to other tasks. This means that a fault controller or fault scheduler can not modify the state of faults during this time, which is a problem, especially for short-running faults like faults in a burst pattern, which require many injections and ejections in a short amount of time.

As already mentioned, the solution to this problem is running scheduling and injection code in a different process than the rest of Mininet. This approach effectively isolates the fault controller from the rest of Mininet, allowing users to write code within Mininet without the need to keep in mind potentially blocking behaviour. It also forces users to cleanly separate fault code from non-fault code, leading to greater modularity.

The downside of this approach is that the FaultController classes lose their direct access to Mininet objects, which makes it harder to direct faults from outside of the fault controller. Specifically, they lose access to the `cmd()` `sendCmd()`, and `pexec()` commands, which allow

callers to execute commands on a node and are an obvious way of injecting faults into nodes.

5.2 INJECTING FAULTS

As described in 5.1, the fault controller and fault injector run in a different process than the rest of Mininet and, therefore, don't have direct access to the node's shells to inject faults. An alternative way of executing commands within a namespace and, therefore, within a node is the `nsenter` command. Since Mininet represents nodes by using namespaces, this means that for all intents and purposes, Faultynet executes commands on the emulated node, even while not accessing the node's shell.

The `nsenter` command requires a process id as an argument. This property is accessible in Mininet's node objects but can't be directly accessed since the fault controller and fault injector run in a separate process. Instead, the fault controller starter needs to provide the fault controller with this information, either during launch or during runtime.

In principle, Faultynet can use both options. In praxis, passing this information during launch is the preferred option since this approach makes sure that all of the required information to initialize a fault controller is available in its `init()` function, which I believe improves accessibility by making the fault controller initialization more similar to the initialization of synchronous objects.

5.2.1 *Passing information during launch*

When starting a new process with Python's `multiprocess` module, the process starter can pass arguments to the started process. Faultynet uses this functionality to pass the process IDs for all relevant nodes from the fault controller starter to the fault controller during launch. Since process IDs are runtime information, a fault controller starter can only be started after Mininet has been started through `Mininet.run()`.

Beyond the process ID, the fault controller starter can also enrich the configuration file provided by the user in other ways.

Here, a fitting example is how `FaultLogger` deals with interface names: Instead of asking users to enter this information manually, Faultynet identifies it autonomously.[1] This means that users can define an injection between two nodes named `a` and `b` by writing `a->b` as an injection target, optionally appending an interface name if multiple such links exist.

1 This functionality is currently only compatible with LinkInjectors, not MultiInjectors, due to their late addition to Faultynet

While the enrichment is currently relatively limited, this process could be expanded on to add additional syntax and features to the configuration files users need to write.

5.2.2 *Passing information during runtime*

While not used to pass information about process IDs, Faultynet does implement duplex communication between fault controller and fault controller starter, which can be used to read the fault controller's state, start a new iteration, or to terminate a fault controller early.

To implement duplex communication between Faultynet's FaultController classes and the rest of Mininet, Faultynet uses two instances of `multiprocessing.Pipe`. Pipes are duplex, but Faultynet uses each pipe to send messages exclusively in one direction since my development experience leads me to assume that separate pipes will lead to fewer bugs when dealing with two-way asynchronous communication.

Within a BaseFaultController object, an `asyncio.Task` constantly listens on one of the pipes and takes actions based on messages as they arrive. In the BaseFaultControllerStarter, messages within the pipe are not constantly monitored but read on an ad hoc basis, not due to technical limitations but because no currently implemented BaseFaultController class requires this behaviour. This means that arbitrary messages can be sent in both directions. Alternative ways of achieving duplex communication exist, but the practical differences are negligible and won't be discussed further.

Currently, interprocess communication is used by the fault controller to signal its state to the fault controller starter, meaning when it has finished initialization and terminated, respectively. In the opposite direction, the fault controller starter signals when the fault controller should start running, when it should terminate, and, for interactive fault controllers, when a new iteration should start. For details about interactive vs non-interactive fault controllers, see 4.4.1.

As discussed in 5.2.1, this architecture is not currently used to pass details about node state from the fault controller starter to the fault controller. However, such functionality could be added relatively quickly if required.

5.3 FAULT INJECTION COMMANDS

While the abstract architecture only states that the fault injector should modify the state of the SDN, a practical implementation needs to decide how exactly those state changes should happen.

Section 5.2 discussed that fault injectors use the `nsenter` commands to execute commands in a node. This section discusses how the commands executed with `nsenter` are constructed.

Fault type	created using
link_fault:delay	tc-netem
link_fault:reorder	tc-netem
link_fault:loss	tc-netem
link_fault:corrupt	tc-netem
link_fault:duplicate	tc-netem
link_fault:redirect	tc-mirred
link_fault:bottleneck	tc-tbf
link_fault:down	ifconfig
multi_fault	tcset
node_fault:stress_cpu	stress-ng
node_fault:custom	sh

Table 5.1: Fault types supported by Faultynet, and underlying tools which are used to inject them

Faultynet implements three different types of fault injectors. `NodeInjector`, which targets nodes and `LinkInjector` and `MultiInjector`, which target interfaces, and therefore links. The separation between LinkInjector and MultiInjector is due mainly to a difference in underlying tools, as well as configurability: Faults relying on LinkInjector support automatically generate interface names due to their more straightforward structure, whereas MultiInjector-based faults currently don't.

This section mentions but does not discuss trade-offs in detail. Those are discussed in 6.2.1.

5.3.1 *LinkInjector*

`LinkInjector` is responsible for the injection of `link_fault` faults. `link_fault:down` disables a network interface using `ifconfig` Disabling an interface does not warrant additional discussion.

All other link-based faults are built upon the tc (traffic control) utility.

To add or remove a fault from an interface, Faultynet sets new tc rules on that interface. The specific rule is impacted by the fault patterns, types, and targets previously passed to the `LinkInjector`. It should be noted that the `LinkInjector` class does not consider the current state of the interface when setting new tc rules. This behaviour is discussed in detail in 6.2.1.

Since Faultynet's link-injection functionality is based upon ThorFis, Faultynet, like ThorFi, calls tc directly and does not rely on intermediary tools. This does slow down iteration speed during development since the programmer needs to be more aware of the tc implementa-

tion details, and it means that any modification or adaptation of LinkFaults requires a deeper understanding of tc than perhaps otherwise necessary, but it also allows Faultynet to use the full expressiveness of tc, which higher-level APIs, like `tcconfig`, may not support.

While a commonly used tool, using tc for fault injections is niche. While the documentation is usually complete, it isn't always correct,[2], and few additional resources are available.

5.3.2 *MultiInjector*

`MultiInjector` is responsible for the injection of `multi_fault` faults, and uses a higher-level API: `tcconfig`, which translates higher level commands into a tc configuration.

Unlike `LinkInjector`, which directly modifies tc's underlying data structures, `tcconfig` offers a more straightforward command line interface, including an option to recreate an interface configuration based on a JSON file, which is how Faultynet injects multiple tc-netem-based faults at the same time.

This makes initial development faster, but `tcconfig` is limited to faults which limit bandwidth, reorder packets, or inject latency, duplication, or corruption, which coincidentally are the faults which are implemented by the tc-netem module, and can be applied and combined in a relatively simple way.

Redirects are not supported, and support of any additional fault types would require either modification of the `tcconfig` tool and support for generalized use cases or fall back to plain tc commands, which would need to make assumptions about how the high-level tool applies its rules.

5.3.3 *NodeInjector*

`NodeInjector` is responsible for the injection of `node_fault` faults. Faultynet currently implements two node-based fault types: `node_fault:stress_cpu`, and `node_fault:custom`.

Additional faults like stressing memory or simulating filled hard drives were considered, but for node-based faults, Faultynet starts to touch upon Faultynet's inherent limitations as an emulator: Out of the box, nodes have access to most of the host system's resources, and I'm convinced that injecting lower-lever system faults fails due to abstraction beyond usefulness. The behaviour exhibited by a namespaced process, potentially running on a consumer system, will be very different from that of the same fault in a production system.

2 I have observed at least two behaviours which I assume to be bugs: First, ingress qdiscs don't respect their assigned tags, and second, the basic filters random object generates a 64-bit random value, which is documented as a 32-bit value, and can only be compared with 32-bit values. I plan to report these bugs after thesis hand-in

Therefore, the insights gained by such an injection will be limited and potentially misleading.

Even the `stress_cpu` fault is not useful on all nodes since, by default, nodes have unlimited access to the host system's CPU. This access can be limited with `CPULimitedHost`, which uses cgroups to limit CPU access, which I believe is the only case where the `node_fault:stress_cpu` is potentially useful.

5.3.3.1 *Stressing the CPU*

When stressing a CPU, the `NodeInjector` automatically calculates the intensity of the stress based on the node's CPU slice, meaning that a `stress_cpu` fault with a 40% intensity on a node, which is limited to 50% of the host system's CPU will lead to a stress of 20% of the host system's CPU, leading to 40% of the node's CPU cycles being stressed.

CPUs are stressed using the `stress-ng` utility using the `decimal64` method. `stress-ng` offers a large number of stressing methods. I chose `decimal64` because the actual CPU usage of the stressing method reflected the value that was passed into the command when investigated with `top`, unlike other methods, which sometimes deviated greatly from the requested intensity.

5.3.3.2 *Custom node faults*

The second type of node-based fault is `node_fault:custom`. Custom faults allow for the scheduled execution of arbitrary commands using the different patterns that Faultynet supports. The degradation pattern is implemented via string replacement: Users can indicate where the changing value should be by using"{}" in the command to be executed. Faultynet supports only one changing argument since what should happen if multiple arguments are present is ill-defined: Users might want tests with all combinations of all arguments or see all arguments increasing simultaneously. The suggested workaround is to write a wrapper bash script, which transforms the single changeable argument into the multiple arguments the user needs for their tests.

The `node_fault:custom` command operates within the host system's file system. This is especially important to support docker containers, within which users may want to run commands without installing the needed utilities in the containers themselves. Should users require direct access to the containers file system, the recommended workaround is to use `docker-exec`.

Faultynet tries to support chained custom commands and handles pipes that are present gracefully by also executing commands that come after pipes within the indicated namespace. Should some syn-

tax not be supported, the suggested workaround is a simple wrapper bash script.

5.4 FAULT LOGGING

5.4.1 *Initialisation*

Faultynet's logging system is implemented in the `FaultLogger` class. Since all fault controllers use the same fault logger, configuration and initialization are the responsibility of the BaseFaultController class.

There is a conflict of responsibility here: On the one hand, each newly implemented fault controller should have total autonomy in the design of its configuration file. On the other hand, a pre-defined structure would allow BaseFaultController to configure and initialize the fault logger automatically, simplifying the addition of new fault controllers and thereby making Faultynet more expandable.

Faultynet tries to solve this dilemma by not making any assumptions about the user-facing configuration file but making some assumptions about the dictionary that the fault controller starter passes to the fault controller. That way, fault controllers can use a custom syntax to define the logger, but the BaseFaultController can automatically initialize the `FaultLogger`.

Specifically, BaseFaultController assumes that within the root level of the dictionary, a `log` key exists, which contains a dictionary that is structured according to the dictionary assumed by the other controllers.

Custom fault controllers that do want to fall back on the configuration syntax that the already existing fault controllers use can pass the unenriched controller configuration to the BaseFaultControllerStarters `get_controller_log_dict()` function, which automatically enriches it and returns the logger configuration dictionary.

5.4.2 *Best-effort logging*

The `FaultLogger` class works on a best-effort basis: Faultynet builds internal state by tracking which command it executes when, and whether those commands executed successfully, and log entries reflect that internal state, not the state of the actual system.

Whenever a fault is injected, the fault injector is responsible for calling the fault loggers `set_fault_active` function, which stores the corresponding fault command, tag, type, and return code in a dictionary. This function assumes that operations on dictionaries are thread-safe, which the CPython implementation guarantees.

In parallel, the fault logger runs a timer in an `asyncio` task, which in regular intervals appends all currently active faults to the log structure and executes all custom commands in the designated nodes.

Thus, the log does not reflect the system state but instead reflects the state of the internal dictionary.

While in theory, the expected state won't diverge from the actual state, in praxis, it certainly might, either through the unexpected failure of commands or external processes modifying the system state. While not measured, to date, this design has not caused issues since command execution success is also tracked, and I consider it an acceptable limitation of the logging system. In fact, during development, this style of logging has been more useful than the alternative since the logger tracks both Faultynet's intent through the internal state, which is modified regardless of whether the injection command returned with or without error, and command outcome, through the inclusion of return codes in the logs.

The original reason for this state-based logging system is the relative ease of implementation. Correctly reading the system state and logging it in an actionable way requires significant effort. Additionally, this work is fault-dependent: As faults start to impact new aspects of the system state, the logger must also expand to take these aspects into account, whereas state-based logging only requires the injector to notify the logger of its intent.

In cases where relying on this type of best-effort logging is not sufficient, custom log commands can be used to track the actual state of the system.

5.5 OPERATING SYSTEM RESTRICTIONS

While operating, Faultynet interacts with specific aspects of the operating system. Faultynet attempts to minimize assumptions by including relevant binaries and dependencies in its distribution, but some of its requirements are too intrusive to enforce on the host system automatically. The Linux ecosystem is relatively diverse, and since broad deployability was not one of Faultynet's design objectives, some aspects of Faultynet will not be functional on non-compliant systems.

5.5.1 *Network interface configuration*

Faultynet's `link_fault:down` assumes that the underlying host system uses `ifconfig` to manage its interfaces. Additionally, MostUsedLinkFaultController uses `ifconfig` to determine how much traffic went over a link. This is an easily modifiable assumption since only the injection, ejection, and traffic volume reading commands must be modified to support an alternative program.

Removing this restriction entirely would require detecting what program is managing interface state on the host system and implementing support for a wider variety of programs. This work is abso-

lutely feasible, but as mentioned, it is not a priority for Faultynet in its current iteration.

5.5.2 *Control Group management*

Cgroups are a Linux kernel feature that allows callers to specify resource limitations for collections of processes. They are used by Mininet to limit the CPU capacity of nodes. In addition to Mininet's usage, Faultynet acquires information about the resource limits associated with nodes to calculate CPU stressing intensity.

5.5.2.1 *cgroup driver*

Using a cgroup requires a cgroup driver. Mininet and Containernet assume that the operating system uses `cgroupfs` as a driver[3], whereas Faultynet expects `systemd` to be the cgroup driver. The underlying incompatibility is the cgroup parent name: Mininet and Containernet use `/docker` as parent name, whereas Faultynet uses `mininetcgroup.slice`. The former is incompatible with `systemd` since `systemd` expects a parent name in the pattern of `<name>.slice`.

The switch from `cgroupfs` to `systemd` was made to support more recent versions of Ubuntu, which is the system on which Faultynet was developed. Changing the compatibility back to `cgroupfs` is as simple as renaming the cgroup parent again.

5.5.2.2 *Control Group versions*

Two different cgroup versions exist: v1 and v2. Faultynet, like Mininet and Containernet, supports version 1. This limitation results from making assumptions regarding how different cgroups are named, and which files need to be read to gain insights about them, which hold true in cgroup v1, but not cgroup v2.

Since Mininet relies on some of those assumptions, and improving Mininet's compatibility with other systems was not a development priority, I did not do an in-detail evaluation of how complex the process of adding cgroup v2 support would be.

Recent Ubuntu distributions default to cgroup v2, but Faultynet's documentation does come with a script that switches a system to cgroup v1 by modifying kernel parameters.

3 My belief that Containernet and Mininet have this assumption is based on the name of the parent, which is incompatible with the systemd driver, and the fact that cgroupfs-mount is installed as a Mininet dependency. However, it should be noted that I could not find a specific reference for this assumption, and my knowledge about what exactly causes this incompatibility may be incorrect.

6 EVALUATION

There are different options for evaluating the implemented prototype. An obvious one would be to run a case study where various users use Faultynet in order to identify unwanted behaviour in previously defined SDNs. While I did reach out to a large number of representative potential users via various avenues, this process did not lead to sufficient testing data. Since I do not believe that testing with users who are unfamiliar with the subject matter would lead to meaningful results, this chapter instead contains a detailed assessment of the tested and perceived strengths, weaknesses, and limitations of Faultynet. Additionally, it discusses what steps would be necessary for it to achieve all its design objectives.

6.1 STRENGTHS, AND ACHIEVED OBJECTIVES

As discussed in 4.2, the core design objectives for Faultynet are Functionality, Usability, Containernet Compatibility, Extendability, Expressiveness, and Pipeline and Local Compatibility.

Out of these, I believe Faultynet fully achieved Functionality, Usability, and Extendability, and partly achieved Containernet Compatibility, Expressiveness, and Pipeline and Local Compatibility.

6.1.1 *Fully achieved*

Faultynet achieves its objective of *Functionality*. Manual testing for a large number of examples confirms that the fault injectors do run when scheduled and that the net's behaviour changes when faults are active: Packets are delayed, dropped, or redirected, and CPU cycles are burned.

Second, I believe that it is fair to claim that the objective of *Usability* was also achieved. Faultynet comes with thorough documentation, and its user-facing interfaces consist of writing simple YML and referencing both a YML file and a fault controller. There is a caveat here: Faultynet does not seamlessly integrate with Mininet's command-line interface. This feature was not included due to the assumption that the command-line interface is only used for minimalist setups, which users use to become familiar with Mininet instead of for actually testing how nets behave. Assuming that the command-line interface won't be used for actual tests, Faultynet's current lack of support does not impact usability. Of course, such functionality could be added to Faultynet if required.

Third, *Extendability*, specifically extendability with regards to custom fault-scheduling modules. Faultynet satisfies this requirement by making it simple to add new fault controllers. Implementing a new fault controller requires very little understanding of Faultynet, and the required boilerplate is minimal. LogOnlyFaultController serves as an illustrative example: It demonstrates how to build upon BaseFaultController in less than 40 lines of code. Any fault controllers will likely be more complex than that, but the limiting factor for new fault controllers will always be the scheduling logic, not the integration with Faultynet.

6.1.2 *Partly achieved*

I consider the requirement of *Containernet compatibility* to be partly achieved. Faultynet is compatible with all nets deployed in Containernet, but there is a limitation as to which links are available for fault injections. Specifically, Faultynet does not currently support links which are limited by using tc, such as Mininet's `TCLink` class. The reason for this behaviour is explained in more detail in 6.2.2.1. Similarly, Faultynet's fault controllers do not currently support injections into nodes which were added to a net during runtime. The foundations for passing arbitrary data, including the process IDs of newly created nodes, from Mininet to the fault controllers exist, and the feature can likely be added relatively quickly if required, but it wasn't a priority during initial development. To assess this requirement as fully achieved, Faultynet would need to be able to not just run all nets defined in Containernet but also inject faults on all nodes and links within those nets.

The requirement of *Expressiveness* was also only partly achieved. This does not mean that Faultynet's faults aren't expressive: In most aspects, Faultynet reaches or surpasses the expressiveness of other frameworks, as can be seen in 6.1. Faultynet supports more fault types and scheduling mechanisms and is among the most expressive with regard to fault patterns, multi-faults, application-specific faults, and protocol-specific faults. While it does not support node reboots, it is the only framework that implements a redirection fault.

However, it does fall short of being able to combine arbitrary faults on the same link, which is what would be necessary for me to assess this objective as fully fulfilled. The challenges of achieving full fault expressiveness are discussed in detail in 6.2.1.1

Finally, *Pipeline and Local Compatibilty*. Faultynet can run on consumer systems without issue. In fact, this is where most development took place. Faultynet can also automatically run in pipelines and terminate itself, and be seamlessly used for user-built pre-deployment tests. However, there is still a relatively high level of friction to use Faultynet for regression tests: If a fault controller identifies a specific

Framework	Fault types	Fault patterns	Multi-faults	Application specific faults	Fault scheduling	Protocol specific faults
ThorFI	Bandwidth, Latency, Loss, Reorder, Duplication, Host reboot, Interface down	Degradation, Burst, Persistent	No	No	User triggered, Timer	Yes
Mockfog 2.0	Bandwidth, Latency, Loss, Reorder, Dispersion, Corruption, Duplication, Memory and CPU limit	Persistent	Yes, grouped	No	State machine	Yes
HAVerifier	Diverse, Module dependent	Persistent	Module dependent	Module dependent	Timer	Module dependent
Fogify	Latency, Bandwidth, Loss, CPU stress, Arbitrary commands, Firewall rules, Interface down	Persistent	Yes, grouped	No	User triggered	No
Armageddon	Loss	Persistent	No	No	Computed	No
Faultynet	Bandwidth, Latency, Loss, Reorder, Duplication, Host reboot, Interface down, Redirection, Stress CPU, Arbitrary commands, CPU limit[a]	Degradation, Burst, Persistent	Yes, grouped	No, but compatible with HAVerifier modules	Timer, Computed	Yes

Table 6.1: Fault expressiveness of Faultynet compared with other fault injection frameworks compared

a This functionality is already present in Mininet

fault combination which causes unwanted behaviour, there is currently no way to automatically read the corresponding log file and create a configuration to recreate the specific conditions that caused the behaviour. In an ideal world, any log by any fault controller could be easily transformed into a configuration file, which could then be used to recreate that exact behaviour. That is currently not the case, and the translation would need to happen manually, which is why I consider this requirement only partially achieved.

6.2 FULLY ACHIEVING PARTIALLY ACHIEVED OBJECTIVES

6.2.1 *Achieving full Expressiveness*

6.2.1.1 *Multiple faults on one link*

Possibly, the most notable limitation of Faultynet's fault expressiveness is its inability to arbitrarily combine multiple faults per link at the same time. At its core, this limitation is inherent, and removing it would require modifying some core aspects of Faultynet's fault injector classes.

As discussed in 5.3, Faultynet injects link-based faults using tc. tc applies rules to packets based on a tree-like structure, which consists of qdiscs, classes and filters. In theory, chaining faults is a simple as adding another leaf node to this structure. In praxis, this process is highly fiddly, both due to technical limitations and complexity, which is inherent in the domain.

To explain, first, the domain's complexity. In tc, each packet is enqueued in exactly one qdisc, but one packet might be affected by faults with more than one target: As an example, an ICMP packet with source port 33790 would be affected by a fault with an "all" targeting all source ports, a fault with "all" targeting port 33790, and a fault targeting ICMP traffic specifically. This behaviour can be modelled by appending each new rule with a corresponding filter to all currently existing leaf nodes and a recreation of all child nodes if a fault is removed from an interface, but this approach requires either an internal representation of the current state of each interface, or parsing logic, which generates that state on each fault injection and ejection.

This approach also assumes that all faults can be arbitrarily chained, which may not be the case, specifically for the netem qdisc, which is responsible for faults which induce delay, loss, corruption, duplication, or reordering of packets. [1][1] Assuming that this is true, combining multiple netem-based faults implies generating a tc-netem command per leaf node based on the set of currently active faults affecting

1 While this reference refers to behaviour which was observed in 2008, a review of commits to the tc command since then has not indicated that this behaviour was changed. However, I did not attempt to reproduce this behaviour.

that leaf node, which again requires either an internal representation or corresponding parsing logic.

This raises additional challenges: If multiple faults are combined into a single netem command the removal of an existing fault requires external state, since not all currently active faults can be reconstructed from just reading the current state of an interface. As an example, a short-running fault that limits bandwidth to 1 mb/s would hide a long-running fault that limits bandwidth to 2 mb/s. Removing the former without also removing the latter requires awareness that the latter is currently active, which is information that tc does not provide.

A further challenge that implementations need to be aware of is that faults are not commutative, which becomes a problem when assessing possible workarounds like moving traffic through multiple interfaces. Traffic which was limited in bandwidth after applying packet loss looks different than traffic where packets are lost before a bandwidth limit is applied.

All of these challenges are surmountable, given sufficient implementation and testing time. However, due to a lack of testing time, I decided not to focus on implementing full support of multi-faults.

6.2.1.2 *Application faults*

Faultynet does not currently implement any application-specific faults out of the box. In this context, application-specific faults refer to executing actions on a node that directly impact an application running on that node, like removing a file or changing a configuration setting. This type of fault can be integrated into Faultynet relatively easily through its custom node-based faults, which is why I don't consider an application-specific fault type necessary to achieve full fault expressiveness.

6.2.2 *Achieving full Containernet Compatibility*

6.2.2.1 *Injections into TCIntf*

Faultynet's inability to inject faults into links which are based on Mininet's `TCIntf` class, which can simulate latency and bandwidth limitations, leads back to the same cause as its limited multi-fault support: `TCIntf` adds bandwidth and latency limitations to links by using the tc-netem utility. As discussed in 6.2.1.1, it is necessary to track a links state externally to properly combine tc-netem-based faults on one interface since chaining instances of tc-netem may lead to errors and not all currently active faults can be reconstructed from reading the current interface state.

The limitations imposed by `TCIntf` are less expressive than those of Faultynet's faults, so the combination of link limitations and faults

would likely be simpler than full multi-fault support. However, since full multi-fault support would also lead to full compatibility with `TCIntf`, it might be prudent to prioritise the former before working on the latter.

6.2.2.2 *Injections into runtime nodes*

Faultynet's fault controllers require process IDs to inject faults into Mininet nodes. Fault controllers can't directly read the process IDs of nodes, and currently, all fault controllers receive all process IDs they require as part of their initialisation arguments just after Mininet's launch. At this point in time, the process IDs of not yet started nodes aren't available and thus can't be passed to the fault controller, which makes it impossible for the fault controller to inject faults into these runtime nodes.

Modifying this behaviour is straightforward since the fault controller and fault controller starter can already exchange arbitrary messages, including process IDs. However, the current, more limited implementation was chosen since I believe it is simpler to understand and extend, and I'm uncertain as to how frequent nets with nodes added during runtime are, which is why adding support for nodes added during runtime was not a development priority.

6.2.3 *Achieving full Pipeline and Local Usability*

6.2.3.1 *Test case generation*

The main hindrance to considering Faultynet fully pipeline compatible is a lack of surrounding tooling, not a lack of Faultynet's inherent functionality. As already discussed in 6.1.2, there currently exists no mechanism to automatically translate a log file into an injection file which can be used for future testing. Such a mechanism would be especially useful for fault controllers that rely on randomness when scheduling injections, such as the RandomLinkFaultController.

Luckily, all the pieces required to effectively turn Faultynet runs into reproducible test cases already exist. Faultynet has a logger that takes note of which faults were injected when, and the ConfigFileFaultController can inject faults based on a configuration file that contains timers.

Two pieces are missing to implement this functionality fully: First, the log needs to contain additional information about the fault that is being injected in order to properly reproduce it. Since the required information is known at log time, and the log format is already structured in JSON, logging this additional information should be trivial.

Second, the JSON log file needs to be transformed into a YML configuration file, which is compatible with the ConfigFileFaultController. I believe it is fair to assume that this step, too, is relatively

easy to implement since all the information that is required by ConfigFileFaultController is present in the log file that is to be converted, including activity times, fault types, target nodes and injection arguments.

6.3 BEYOND FAULTYNET'S REQUIREMENTS

While Faultynet does build a solid foundation for Mininet-based fault testing, it could be improved across many dimensions beyond its initial requirements.

This section aims to highlight some possible dimensions, including upsides and challenges. Unlike future works, which suggest work which could build upon Faultynet, the scope of this section is strictly improvements to Faultynet itself.

6.3.1 *Improving fault expressiveness further*

There are few technical limits when thinking about how Faultynet's fault expressiveness could be further increased. Some options include new fault patterns, targets, or adding new types of faults.

While it's hard to make generalisable statements about all fault types, in general, the limiting factor is not ease of implementation but usability.

More expressiveness doesn't improve Faultynet if the added expressiveness doesn't improve coverage of what types of unwanted behaviours can be tested for. In fact, there's an argument to be made that too much expressiveness detracts from Faultynet's usability by hiding the most commonly useful options among too many alternatives, thereby overwhelming users.

This is why I would suggest delaying additional improvements in fault expressiveness until a proven need arises, either through user requests or through other frameworks expanding their expressiveness.

6.3.2 *New fault controllers*

Much like increasing fault expressiveness, the main challenge for adding additional fault controllers isn't technical but design-based instead. However, unlike with fault expressiveness, I believe that Faultynet's fault controllers are still far from the point of diminishing returns: There are a number of test cases that haven't yet been explored by Faultynet's existing fault controllers.

Some potentially interesting examples include

- Failover controllers, which accept groups of nodes based on a naming pattern and iteratively inject faults into one group of nodes, simulating outages in one of many data centers

- Pass/Fail controllers, which inject faults randomly, evaluate whether the system-under-tests reaction constitutes a pass or a fail, e.g. by running a bash script, and logs failure-inducing injection states
- A searching controller, which accepts a fault log and a pass/fail criterium and attempts to identify a minimalist combination of faults which still results in a fail
- Machine-learning-based controllers, which inject based on maximum impact and could be used to identify time-effective injection strategies

Faultynet was very much designed to simplify the development of these and other new fault controllers, and the primary limiting factor is user creativity.

6.3.3 *Automated injection impact evaluation*

Some users might be interested in adding automated injection impact evaluation to Faultynet without relying on a specific fault controller or manually programmed pass/fail conditions.

While Faultynet logs which faults are active, it does not currently track the impact of faults and does not explicitly implement any hooks for pass/fail reporting, which would be especially useful when integrating Faultynet into automated tests.

The main impediment to implementing such a feature is the ill-defined nature of a pass compared with a fail: Measuring impact on systems in a generalised way is challenging since systems under test don't offer a uniform API to assess "impact", and users may not have a unified definition regarding how severely impacted a service must be before a pass becomes a fail.

The alternative to measuring pass vs fail would be to measure generalised metrics, like the percentage of lost packages on a link. While certainly possible, I do not believe that such metrics would be useful since they only attest to the presence of faults, a tautological metric, since we are measuring the presence of faults we ourselves are injecting.

For users who do know what constitutes a pass vs a fail, Faultynet's custom commands offer a simple way of integrating that knowledge into a new fault controller. However, I believe that generalising these commands beyond a case-by-case basis into a holistic solution would require a longer-term development project.

6.3.4 *Improving Faultynet's tenant status*

Faultynet currently relies on operating tools to inject faults.

The reason for this decision is obvious: Writing an entirely different stack or forking tc to grant Faultynet privileged exclusive access to interface configurations would be a prohibitively time-consuming process. The downside is equally obvious: Faultynet's fault injector is just another process asking the operating system nicely to please follow its instructions, and any other process with comparable privileges can do the same and overwrite the fault configuration.

This is not a hypothetical problem: Testing has shown that running `tcpdump` on an interface overwrites and removes Faultynet's redirection faults, which are based on tc ingress qdiscs. Since tools might not announce their implementation details up front, this behaviour might lead to faults not being injected as instructed at unknown times, making Faultynet less reliable as a whole. In this specific case, duplicating all traffic from one interface to a second one and running `tcpdump` on that second interface might be a workaround, but other, as of yet, unidentified tools might require different solutions.

Increasing Faultynet's authority would require a major effort to re-architect core fault injection logic, and I find it hard to judge whether this material effort would be justified by user requirements.

A less work-intensive workaround, which would add some stability and much greater detectability of this condition, might be achieved by modifying Faultynet's fault logger to not just log state but also to compare the current system state with the expected state and raise a warning should those two diverge. This workaround, too, would require a significant amount of work but would lead to fewer compatibility issues and keep Faultynet's stack more consistent with other frameworks, which, in my opinion, makes this the preferred option.

6.4 FUTURE WORKS

The most obvious next step is expanding Faultynet's capabilities, fully achieving design objectives that were only partially achieved, and expanding its feature suite beyond that. Possible options for work of such type are detailed in 6.2 and 6.3.

Beyond just features, exploring the limits of useful fault expressiveness seems to be a currently underdeveloped area in research. Faultynet itself can be configured to inject various fault patterns, and users can define specific injection target protocols. tc-netem has additional expressiveness, which Faultynet currently doesn't actively use, like delaying packets with different distributions or dropping packets based on different loss models. Yet it is unclear up to what point this added expressiveness results in actual differences when testing software. Further works exploring this space could help to define and hopefully minimise the fault corpus that a testing framework needs to support before being considered feature complete and would allow developers to focus on useful faults instead of novelty expressiveness.

On a different axis, Faultynet's usefulness for software testing would be greatly improved with a larger corpus of fault controllers. Ideally, those fault controllers would come with experience reports about what types of behaviour they were able to unearth and what kind of testing they were most useful for. Once developed, this corpus of fault controllers could be used outside of Faultynet, either to schedule faults in other fault injection frameworks or to inject faults efficiently in a production environment, which would return the Faultynet fault controllers back to their chaos engineering origins.

Another avenue of research which could effectively lead to test corpus minimisation would be the exploration of different fault injection schedules through the development and comparison of fault controllers. Can an ideal fault injection strategy be identified based on the static properties of a given network? Are fault controllers which have access to runtime properties more effective? This is an area where the introduction of a machine-learning-based fault controller could lead to the discovery of potentially interesting properties.

There are also a number of more production-centred projects that could lead to useful results. The first option here is to see whether faults from a production system can be automatically translated into Faultynet test cases. Such a system would likely require agents to be developed and installed into the target network, but it could be a big step towards fully automated test case generation and infrastructure regression testing.

Research in the opposite direction could also lead to interesting results: Faultynet is an emulator, and while it is reasonable to assume that emulated faults on Faultynet lead to similar behaviour to real faults in production, this is currently unproven. Re-integrating Faultynet's features into an OpenStack-based fault injector like ThorFi and injecting identical faults into identical networks to compare differences in impact could go a long way towards proving that Faultynet's behaviour accurately reflects that of the real world.

Much of my work assumes, based on different conversations, that there are inherent differences in the scalability of a Mininet-based approach when compared with the scalability of an OpenStack one. A quantitative study contrasting both in detail could help highlight where exactly those differences are and which approach is preferable in what kind of environment.

7 CONCLUSION

In this thesis, I Introduced a concept and prototypical implementation for an SDN and CI/CD compatible fault injection framework called Faultynet, based on Mininet and Containernet, and designed on the principles of Functionality, Usability, Containernet Compatibility, Extendability, Expressiveness, and Pipeline and Local Compatibility.

Faultynets compares favourably with existing fault injection frameworks regarding its fault expressiveness and comes with a logging framework that can be used to support the creation of new injection schedules and support for modular fault controllers, allowing users to define their own fault injection scheduling algorithms.

While not battle-proven, Faultynet already shows that building a fault injection framework upon Mininet is not just possible but feasible and that building a system that automatically injects faults into SDNs as part of CI/CD pipelines is a realistic proposition.

In doing so, this thesis makes a big step towards demonstrating that integrating fault tests into continuous testing pipelines for networks is feasible and lays the foundation for future works which can explore using Mininet to develop fault scheduling algorithms and to automatically evaluate networks' fault resistance during automated testing.

BIBLIOGRAPHY

[1] Rafael C. de Almeida. *[Netem] Combining netem rules*. June 2008. URL: https://lists.linuxfoundation.org/pipermail/netem/2008-June/001280.html (visited on 12/30/2023).

[2] J. Arlat, Y. Crouzet, and J.-C. Laprie. "Fault injection for dependability validation of fault-tolerant computing systems." English. In: IEEE Computer Society, Jan. 1989, pp. 348,349,350,351,352,353,354,355–348,349,350,351,352,353,354,355. DOI: 10.1109/FTCS.1989.105591. URL: https://www.computer.org/csdl/proceedings-article/ftcs/1989/00105591/12OmNzayNsN (visited on 07/10/2023).

[3] Ayush Bhardwaj, Zhenyu Zhou, and Theophilus A. Benson. "A Comprehensive Study of Bugs in Software Defined Networks." In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. ISSN: 2158-3927. June 2021, pp. 101–115. DOI: 10.1109/DSN48987.2021.00026.

[4] Netflix Technology Blog. *The Netflix Simian Army*. en. Sept. 2018. URL: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (visited on 06/25/2023).

[5] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. "Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction." In: *ACM SIGCOMM Computer Communication Review* 45.4 (Aug. 2015), pp. 371–372. ISSN: 0146-4833. DOI: 10.1145/2829988.2790038. URL: https://dl.acm.org/doi/10.1145/2829988.2790038 (visited on 05/19/2023).

[6] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. "NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems." In: *IEEE Transactions on Network and Service Management* 14.4 (Dec. 2017). Conference Name: IEEE Transactions on Network and Service Management, pp. 934–948. ISSN: 1932-4537. DOI: 10.1109/TNSM.2017.2733042.

[7] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. "ThorFI: a Novel Approach for Network Fault Injection as a Service." en. In: *Journal of Network and Computer Applications* 201 (May 2022), p. 103334. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2022.103334. URL: https://www.sciencedirect.com/science/article/pii/S1084804522000030 (visited on 05/19/2023).

[8] Resul Das and Muhammad Muhammad Inuwa. "A review on fog computing: issues, characteristics, challenges, and potential applications." In: *Telematics and Informatics Reports* (2023), p. 100049.

[9] Qingfeng Du, Zheng Ni, Ruian Zhu, Mengyi Xu, Kecheng Guo, Weiya You, Ruolin Huang, Kanglin Yin, and Qibin Zheng. "A Service-Based Testing Framework for NFV Platform Performance Evaluation." In: *2018 12th International Conference on Reliability, Maintainability, and Safety (ICRMS)*. ISSN: 2575-2642. Oct. 2018, pp. 254–261. DOI: 10.1109/ICRMS.2018.00055.

[10] Qingfeng Du, Juan Qiu, Kanglin Yin, Huan Li, Kun Shi, Yue Tian, and Tiandi Xie. "High availability verification framework for OpenStack based on fault injection." In: *2016 11th International Conference on Reliability, Maintainability and Safety (ICRMS)*. Oct. 2016, pp. 1–7. DOI: 10.1109/ICRMS.2016.8050168.

[11] Le Duytam Ly, Mahsa Sadeghi Ghahroudi, and Victor Ponce. "A Systematic Literature Review of Reliable Provisioning for Virtual Network Function Chaining." en. In: *Applied Sciences* 13.9 (Jan. 2023). Number: 9 Publisher: Multidisciplinary Digital Publishing Institute, p. 5504. ISSN: 2076-3417. DOI: 10.3390/app13095504. URL: https://www.mdpi.com/2076-3417/13/9/5504 (visited on 06/01/2023).

[12] Paulo César Fonseca and Edjard Souza Mota. "A Survey on Fault Management in Software-Defined Networks." In: *IEEE Communications Surveys & Tutorials* 19.4 (2017). Conference Name: IEEE Communications Surveys & Tutorials, pp. 2284–2321. ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2719862.

[13] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. "Understanding network failures in data centers: measurement, analysis, and implications." In: *Proceedings of the ACM SIGCOMM 2011 conference*. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, Aug. 2011, pp. 350–361. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018477. URL: https://dl.acm.org/doi/10.1145/2018436.2018477 (visited on 05/24/2023).

[14] Jonathan Hasenburg. "Data distribution for fog-based IoT applications." en. In: (2021). URL: https://depositonce.tu-berlin.de/handle/11303/13322 (visited on 06/28/2023).

[15] Jonathan Hasenburg, Martin Grambow, and David Bermbach. "MockFog 2.0: Automated Execution of Fog Application Experiments in the Cloud." In: *IEEE Transactions on Cloud Computing* 11.1 (Jan. 2023). Conference Name: IEEE Transactions on Cloud Computing, pp. 58–70. ISSN: 2168-7161. DOI: 10.1109/TCC.2021.3074988.

[16] Jonathan Hasenburg, Martin Grambow, Elias Grünewald, Sascha Huk, and David Bermbach. "MockFog: Emulating Fog Computing Infrastructure in the Cloud." In: *2019 IEEE International Conference on Fog Computing (ICFC)*. June 2019, pp. 144–152. DOI: 10.1109/ICFC.2019.00026.

[17] Fizza Hussain, Syed Ali Haider, Abdullah Alamri, and Mohammed AlQarni. "Fault-tolerance analyzer: A middle layer for pre-provision testing in OpenStack." en. In: *Computers & Electrical Engineering* 66 (Feb. 2018), pp. 64–79. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2017.11.019. URL: https://www.sciencedirect.com/science/article/pii/S004579061730887X (visited on 05/24/2023).

[18] Yahui Li, Xia Yin, Zhiliang Wang, Jiangyuan Yao, Xingang Shi, Jianping Wu, Han Zhang, and Qing Wang. "A Survey on Network Verification and Testing With Formal Methods: Approaches and Challenges." In: *IEEE Communications Surveys & Tutorials* 21.1 (2019). Conference Name: IEEE Communications Surveys & Tutorials, pp. 940–969. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2868050.

[19] ONF. *Project Mininet Winner of inaugural ACM SIGCOMM SOSR Software Systems Award*. en-US. May 2017. URL: https://opennetworking.org/news-and-events/blog/project-mininet-winner-of-inaugural-acm-sigcomm-sosr-software-systems-award/ (visited on 09/24/2023).

[20] *PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering*. URL: http://principlesofchaos.org/ (visited on 06/25/2023).

[21] Jiantao Pan. "Software reliability." In: 1999.

[22] Manuel Peuster and Holger Karl. "Profile your chains, not functions: Automated network service profiling in DevOps environments." In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2017, pp. 1–6. DOI: 10.1109/NFV-SDN.2017.8169826.

[23] Manuel Peuster et al. "Introducing Automated Verification and Validation for Virtualized Network Functions and Services." In: *IEEE Communications Magazine* 57.5 (May 2019). Conference Name: IEEE Communications Magazine, pp. 96–102. ISSN: 1558-1896. DOI: 10.1109/MCOM.2019.1800873.

[24] Casey Rosenthal and Nora Jones. *Chaos Engineering: System Resiliency in Practice*. en. Google-Books-ID: iVjbDwAAQBAJ. "O'Reilly Media, Inc.", Apr. 2020. ISBN: 978-1-4920-4383-6.

[25] Nick Shelly, Brendan Tschaen, Klaus-Tycho Förster, Michael Chang, Theophilus Benson, and Laurent Vanbever. "Destroying networks for fun (and profit)." In: *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. HotNets-XIV. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–7. ISBN: 978-1-4503-4047-2. DOI: 10.1145/2834050.2834099. URL: https://dl.acm.org/doi/10.1145/2834050.2834099 (visited on 06/08/2023).

[26] Vlada Strazdina. "A hybrid automated framework for testing cloud-native and virtual core network applications." en. Accepted: 2022-08-28T17:15:00Z. PhD thesis. Aug. 2022. URL: https://aaltodoc.aalto.fi:443/handle/123456789/116352 (visited on 06/02/2023).

[27] Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. "Fogify: A Fog Computing Emulation Framework." In: *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. Nov. 2020, pp. 42–54. DOI: 10.1109/SEC50012.2020.00011.

[28] Kostis Trantzas, Christos Tranoris, Spyros Denazis, Rafael Direito, Diogo Gomes, Jorge Gallego-Madrid, Ana Hermosilla, and Antonio Skarmeta. "Implementing a holistic approach to facilitate the onboarding, deployment and validation of NetApps." In: *2022 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. Sept. 2022, pp. 261–267. DOI: 10.1109/MeditCom55741.2022.9928721.

[29] Sebastian Troia, Marco Savi, Giulia Nava, Ligia Maria Moreira Zorello, Thomas Schneider, and Guido Maier. "Performance characterization and profiling of chained CPU-bound Virtual Network Functions." en. In: *Computer Networks* 231 (July 2023), p. 109815. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2023.109815. URL: https://www.sciencedirect.com/science/article/pii/S1389128623002608 (visited on 06/05/2023).

[30] José Manuel Sanchez Vilchez and David Espinel Sarmiento. "Fault Tolerance Comparison of ONOS and OpenDaylight SDN Controllers." In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. June 2018, pp. 277–282. DOI: 10.1109/NETSOFT.2018.8460099.

[31] Ziqiang Wang. *Design and Implementation of a Reliable Container-based Service Function Chaining Testbed in Cloud-native System: An Open Source Approach*. eng. Publisher: Carleton University. Sept. 2022.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of February 7, 2024 (`classicthesis` version 4.5).