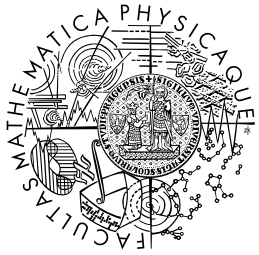

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Piroutek

Fuzz testing of network subsystem in PikeOS

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Pavel Parízek, Ph.D.

Study programme: Computer Science

Study branch: Programming and Software Development

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date January 8th, 2024 Jan Piroutek

I want to thank my supervisor, doc. RNDr. Pavel Parizek, Ph.D., for his help and advice, whenever I needed anything. Another important thank goes to SYSGO GmbH for providing a platform of PikeOS with all necessary tools and a supporting consultant who help was beneficial to finish this thesis.

I am also very grateful to my family, friends, and significant other, who supported me throughout my studies and helped me through all of this.

Title: Fuzz testing of network subsystem in PikeOS

Author: Jan Piroutek

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Stability under every possible circumstance is a goal for a lot of applications. This problem applies to the network stack ANIS of the real-time operating system PikeOS developed by SYSGO. PikeOS requires security and stability because it is used in areas, e.g., airborne systems, where unstable software could cause severe damage. A proven way to ensure the stability and security of software is testing. Fuzzing is an automated testing technique that generates randomized inputs for the application to find bugs, vulnerabilities, or crashes within the application. Another testing technique is long-run testing, which exposes an application to some input for longer periods.
Because ANIS is a product usually shipped with PikeOS, it must follow the same security standards. We have developed a testing tool for the ANIS network stack, using the two mentioned techniques and emphasizing the option to configure such a test. This testing tool exposes the ANIS to various scenarios that could stress the stack and uses fuzzing to create a combination of these scenarios automatically, which could crash the network stack. The developed test is implemented with a small set of scenarios that expose ANIS to various network traffic. The test can be extended to work with more scenarios. All scenarios have a predefined set of parameters determined by the fuzzer. Changing the parameters of the scenarios diversifies generated network traffic. The scenarios and their parameters are automatically generated every round by the fuzzer. The fuzzer has another set of parameters that give users a way to influence how the data for the test is generated.

Keywords: Long run test Fuzzing RTOS Network stack PikeOS

Název práce: Fuzz testování síťového subsystému v PikeOS

Author: Jan Piroutek

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Pavel Parízek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Dosáhnout stability za jakékoli situace je cílem spousty aplikací. Tento problém se týká také síťového stacku ANIS, který je součástí operačního systému reálného času PikeOS vyvíjeného společností SYSGO. PikeOS vyžaduje bezpečnost a stabilitu svých komponent, protože je používán v průmyslu jako je např. letectví, kde by nedostatek těchto vlastností mohl způsobit veliké škody. Vyzkoušená cesta pro ověření stability a bezpečnosti programu je jeho testování. Fuzz testování je technika automatického testování, která se snaží v programu najít chyby skrz generování náhodných vstupů. Jejím cílem je najít zranitelnosti a odhalit potenciální chyby, které mohou mít závažné důsledky na provoz aplikace. Další testovací technikou je long-run testing, přes který je aplikace vystavena náporu po delší časový úsek.

Jelikož ANIS je běžně dodávám jako součást PikeOS, musí také splňovat stejné bezpečnostní standardy jako PikeOS. My jsme s pomocí long-run a fuzz testování vytvořili testovací program pro síťový stack ANIS. Při tvorbě jsme kladli důraz na možnost nastavování našeho testu. Tento test vystavuje ANIS různým scénářům, které mají za úkol zatížit ANIS. Test používá fuzzing jako nástroj pro generování kombinací těchto scénařů a snaží se s jejich pomocí donutit ANIS k chybám. V rámci vývoje jsme opatřili test malým vzorkem scénářů, které vystavují ANIS různému síťovému provozu. Všechny scénáře mají předem definovanou množinu parametrů. Změnou hodnot těchto parametrů jsme schopni generovat různorodější scénáře. Scénáře s jejich parametry jsou generovány vždy před začátkem testování. Jako možnost konfigurace má fuzzer svou vlastní množinu parametrů, kterou je uživatel schopen ovlivnit způsob, jakým budou data pro test generována.

Klíčová slova: Dlouhodobé testování Fuzz testování RTOS Síťový stack PikeOS

Contents

1. Introduction

SYSGO is a leading provider of real-time operating systems focusing on safety-critical embedded applications. With customers operating in areas from healthcare, industrial automation, and transportation up to aerospace, defense, or space programs. When using SYSGO software products, customers often require certification against industry-specific Safety and Security standards. SYSGO spends many resources on verification to ensure their systems follow these standards. The leading standard SYSGO uses is DO-178C, which describes development processes and other needs required by certification authorities. These standards are abstract, so SYSGO needs to process them so they are more useful in software development. These standards can have levels that correspond, for example, with several errors that occur in some time interval. PikeOS is certifiable to the highest levels of some certification standards, for example, Security standards Common criteria, and Evaluation Assurance Level.

SYSGO's flagship product is the real-time operating system PikeOS. PikeOS needs to be certified for safety and security because it's primarily used in critical systems. PikeOS offers strict separation of multiple partitions that can contain other operating systems or applications. Not only are resources strictly separated, but PikeOS also offers strict time partitioning that distributes a defined CPU time to different logic units. These features enable building applications with strong demands on security and safety.

With SYSGO software being delivered to avionics and defense industries, it's important for all applications to be as secure and reliable as much as possible. Users need to be sure about the stability of the software before its deployment to production. Therefore, SYSGO, with its software, ships documents describing the test suites and their results. These test suites are the results of severe verification efforts and testing the software.

Many applications need support for a connection to the internet. For that purpose PikeOS has its own certifiable IP stack CIP, internally known as ANIS. ANIS is a UDP/IP stack compatible with most of the standard RFC specifications for IP and UDP protocols. The ANIS allows developers to use its socket interface for easier communication over the network without creating their own. For communication on the lower network layer, ANIS uses the support of ethernet drivers, so ANIS only implements the networking and transportation layer of the OSI model.

Similar to PikeOS, ANIS also strictly follows standards. To ensure these standards, verification engineers' spent thousands of hours of work. SYSGO ensures the validity of ANIS with multiple test suits that contain hundreds of test cases and other verification tools. These tools check if the network stack is compatible with RFC specifications and can handle all situations.

When it comes to software testing, ensuring the desired coverage of a program state space can be a challenging task. When the number of all possible inputs is small, we can generate them all and feed them through a series of tests in the program. That is not the case with a network stack. It becomes difficult with different networking protocols and the fact that it needs to function over longer periods. The longer periods create more complexity with varying combinations of

everything that can happen in the network and what the network stack has to deal with. The goal of this project was to address this challenge and develop a testing framework for ANIS. We would like to explore these areas when there is a lot of random traffic going into the network stack if it behaves correctly under stressful situations, like long test case running time or excessive and diverse network traffic, or if the software can't handle those situations and crashes.

We have created a test prototype that generates network traffic as the input for ANIS and monitors its behavior. One desired feature was integrating the test with existing test suites for ANIS. Because this would ease the verification process for SYSGO and its engineers. We will aim for a framework with a couple of premade scenarios. Ideally, we want the framework to be extensible with more scenarios to explore other areas of the network stack.

However, we discovered that there are more efficient ways to test our stack than having hard-coded scenarios with all their parameters in a test . What if we wanted to swap the execution order of two different scenarios or slightly adjust their parameters? We randomized our predefined scenarios so we could extend the coverage of the program state space. There exists a technique for the automated generation of random inputs for tests. It's called fuzzing. Recently, fuzzing has gained more popularity because it is more and more used by software companies to find bugs in their code and improve reliability of the software. Fuzzing aims to generate inputs that lead to unwanted behavior, like crashes, memory leaks, or security problems. Although this method might seem very simple, it proved to have successes in the testing field[1].

We wanted to explore fuzzing as a tool to help us generate network traffic and explore obscure or extreme scenarios that probably weren't thought of by testers. But first, we needed to create a test using scenarios to test ANIS. In the end, we extended this test with a fuzzer that can generate diverse network traffic using our scenarios.

1.1 Thesis structure

First, in the next chapter 2, we will describe technologies used in this project or closely tied to it. Chapter 3 focuses on other tools or research linked to the fuzzing of a network stack. Chapter 4 formulates our project's features, which we implemented. In chapter 5, we will look at the project's architecture. We will talk about the design of the test, individual parts, their purpose, and their evolution to the current state. In chapter 6, we will discuss implementation details and what problems we had to solve during implementation. The last chapter 7 is dedicated to discussing how we can extend the test and other future work that could be done on this project.

[1] One of the examples could be the trophy room of the afl fuzzer [Heuse et al.,]

2. Background

In this chapter, we introduce concepts tied to the project. We will look more closely into the world of real-time operating systems, specifically PikeOS. We will discuss networking and what a network stack is, continuing to ANIS. The following section will be dedicated to the test suite that SYSGO already has and long-run tests. In the last section, we introduce fuzzing in detail, and primarily, we will focus on theories and techniques that can improve the efficiency of fuzzers.

2.1 Real-time operating systems

Embedded systems are a combination of software and hardware designed to perform some specific function. One example could be a self-driving car system that hits breaks on some input. When a signal is sent into this system, it must react in a certain time, or the crash will probably occur. Breaks could be implemented only as a single microchip without more significant issues. But if we put more and more systems like an automatic shift, computer vision, and others, the car needs more and more hardware to perform all computation. That's time for the real-time operating systems, RTOS. RTOS schedules the tasks of the car or other complex system. RTOS must finish some tasks before the deadline, e.g., breaks need to be activated fast, or a crash can happen. RTOS allows us to configure the hardware, so there is enough computation power to finish all tasks in time.

In contrast to commonly used operating systems, like Unix, RTOSes need to meet task deadlines. The scheduler then needs to focus on meeting deadlines for individual tasks. The strict deadlines lead to some sort of scheduler determinism when we can predict the order of tasks. Usually, the order is done through priorities for tasks. The scheduler might need to replan the execution of tasks, if there is a new task with higher priority coming. Higher-priority tasks usually have earlier deadlines, so the scheduler needs to ensure that this deadline won't be missed.

PikeOS

PikeOS is an RTOS with hard deadlines. For more detailed information there is a SYSGO product overview[1]. It is designed for any safety-relevant context where timing is a critical factor. In that environment, we always consider the worst-case execution time instead of asymptotic complexity. For example, instead of a hash map with asymptotic complexity for search in constant time, it is better to use structures that perform tasks fast every time. The hash map search sometimes needs to check all elements. Therefore it is better to use something like red-black trees, that always search in logarithmic time.

PikeOS provides a hardware abstraction layer, which makes it easier to port applications to different hardware platforms. The great value of PikeOS lies in the strict management of resources. It offers strict resource partitioning, preventing failures from propagating between multiple partitions. Resources are allocated

[1] `https://www.sysgo.com/fileadmin/user_upload/data/flyers_brochures/SYSGO_PikeOS_Product_Overview.pdf`

during the build time of the system image. PikeOS then assures that partitions won't use more resources than assigned during the build. PikeOS allows us to use different interfaces inside those partitions called personalities. These can be guest-host operating systems or real-time environments. Some examples would be native PikeOS, ELinOS, which is the SYSGO Linux distribution or APEX API from ARINC 653 standard, which is a standard for avionics embedded systems[2]. Throughout this thesis, we only talk about PikeOS native personality and don't use the others.

PikeOS requires a memory management unit for virtual to physical address translation. Memory space for partitions is separated for all applications by default, but there is an option for shared memory. Some certified versions require all memory to be pre-allocated, meaning dynamic allocation with malloc and free is not supported. Other versions have some dynamic allocation, but the dynamic allocation can only be used during initialization, not while the system is running. The dynamic allocation is not implemented, because it is unstable, leads to fragmentation, and is error-prone.

When developing for PikeOS, there are two types of projects: Integration and Application. Application projects are the single application running in separate partitions. Integration projects tie together all applications and configure the PikeOS. The whole PikeOS image is generated during build time, and this generated structure doesn't change during run time. System integrators[3] can assign resources and memory to partitions, which need to operate with them. They won't get more during run time. Integrators create communication channels in integration projects between partitions, messaging queues, or priorities for partitions and their processes.

2.2 Network stack

A network stack is a set of protocols trying to achieve successful communication over the network. The most relevant protocols for us are UDP on the transportation layer and IP on the networking layer. Communication is split into multiple layers that communicate vertically with each other. When a packet is sent, the layer usually wraps its information with the data as a header or footer and passes it to the lower layer. The lowest layer sends the packet to the physical device through some physical connection. A similar process occurs on the receiving end when layers read information from their headers and push the rest to the higher layer.

ANIS

PikeOS uses a certifiable network stack called ANIS, more information about it is on the SYSGO website[4]. ANIS is a UDP/IP stack. This means that the protocol

[2] `https://www.sysgo.com/arinc-653`

[3] When developing projects for PikeOS, the work is split into different roles. The same person might occupy multiple roles. Developers are responsible for creating the components of the final product. Verification engineers validate the code. System integrators, whose goal is to put everything together into one final product.

[4] `https://www.sysgo.com/popups/cip-certifiable-ip-stack`

chosen for the network layer is the IP, and on the transport layer, the UDP is used. On the lower layers, ANIS implements ARP protocol. ANIS is responsible for answering ARP requests or cache translations of IP addresses to MAC. All protocols follow the RFC specifications. RFC is the standard for communication over the internet, so the ANIS is fine communicating with other network stacks following these standards.

Features that ANIS supports are limited compared to other network stacks. SYSGO wants this stack to be certifiable so their customers can save resources for certification. With more features, certifying ANIS to the required standards would be more challenging, so SYSGO goes for smaller but still sufficient implementations. Except for the protocols mentioned earlier, ANIS also supports ICMP and IGMP. ANIS has implemented standard communication methods: unicast, multicast, and broadcast.

ANIS has a very static architecture because it needs to be safety-certifiable. Therefore, ANIS has a predefined amount of memory that it can use, which necessarily leads to dropping packets when it's under too much pressure. Usually, it would be impossible to certify for stability unstable protocol like UDP. With ANIS, it is possible, because ANIS has semi-formally defined behavior, when it drops packets, and this behavior can be certified for stability.

2.3 Certification

SYSGO critical software needs to follow strict standards for safety and security. The software can achieve different levels of assurance for the standards. The purpose of this is to show that the software is working correctly, and without these assurances, it cannot be used in production.

The working system should be reliable and secure. To verify their products, mainly PikeOS, SYSGO adopted DO178-C certification[5], which is a standard for dealing with safety in safety-critical software in airborne systems. This standard defines development and verification processes. SYSGO is centered around this certification norm. The standard is adopted and developed into the company standard, that is used in development and verification processes. There are multiple levels of Design level assurance, DAL, that are saying more about the safety of the product. The second part of the guidelines is verification, ensuring that the code works. Most time on verification is spent on requirement-based testing. A test suite is a set of tests that verify the system under test. SYSGO has many test suites for its products. For ANIS, test suites already exist, one containing integration tests and one for testing the API. To use these certification test suites SYSGO uses the test framework, referred to as TFW. This test framework is meant to reduce the time for certification and testing of embedded software.

TFW test cases

Let's provide more details about the test framework TFW. TFW creates an environment for test cases and is responsible for building, running the test suite or other supporting features.

[5] `https://www.do178.org/`

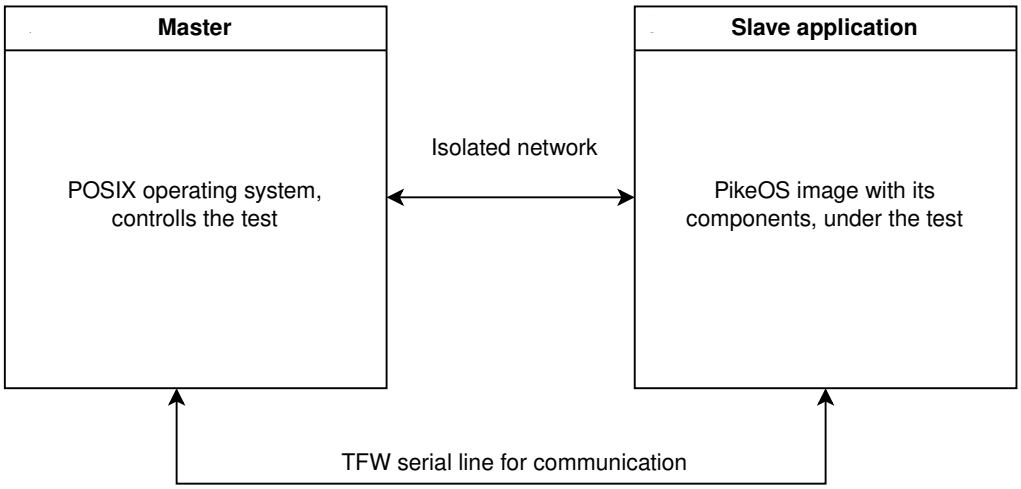

Figure 2.1: Structure of typical TFW test case

Every test case inside TFW follows the same structure. There are two sides, Slave and Master. Master is a UNIX application controlling the test case. The Slave is the application under the test.

TFW has some features that help certification engineers write test cases. They can rely on TFW formatted logging and test case evaluation. For stable communication between Master and Slave, TFW offers serial line communication, physical or virtual, if we use virtualization tools. This serial line is a great tool for synchronization between Master and Slave. We already mentioned the execution and building of the test cases. If developers need to use PikeOS, TFW gives us some tools to configure the PikeOS build process and provides many ways to configure the test cases.

2.4 Virtualization

Sometimes, developers or testers don't have access to the correct hardware or any hardware to run our system. Virtualization is a tool that creates an abstraction layer in your computer. This layer allocates resources and can simulate the behavior of specific hardware, which testers would need to run the system and run tests on the running system.

Virtual machines are virtual representations of real physical computers. They emulate their CPU, memory, or other resources and components. Virtualization allows us to run multiple computers inside one computer. The computer, the virtual machines are running on, is called host, and the machines are called guests. Guests are separated from each other, they don't share resources, and also separated from the host. Virtualization creates an environment, that is close to multiple physical computers.

One of the most popular tools for Virtual Machines is QEMU[6]. It is an open-source software that can emulate different types of CPUs, manage memory resources, and support networking. It simulates the whole machine. Important for

[6] `https://www.qemu.org/`

us is that TFW supports the QEMU platform.

2.5 Fuzzing

Our last piece of information is about fuzzing. Fuzzing is a testing technique introduced at the University of Wisconsin in 1989. What was originally an assignment for students became a common and very popular testing technique. It was first used to test 88 common Unix utilities. Surprise or not, in around 30% of these utilities, bugs were found according to the original report [Miller et al., 1990]. The most common types are invalid pointers to arrays, wrong return codes, and even race conditions.

The general idea behind fuzzing is to feed tested system randomized data through its input channels. Randomized input could trigger bugs that regular testing wouldn't discover because it usually covers only common use or edge cases. The fuzzing may try any stream of data that can come to the system, which ranges from common use cases up to anything possible, like completely random strings of characters. The input relies on random fuzz generators that automatically create more and more inputs to be fed to the systems.

The most basic automated fuzzing tests consist of connection to the tested application, fuzz data generator and success check to see what happened. The test flow would be to generate data, send it to the application and observe and evaluate results.

Firstly fuzzing was used by individual hackers trying to find exploitable vulnerabilities, but now it's being adopted by big tech companies to better test their systems. Over the years, fuzzing proved to be a very useful tool, when looking for possible crashes and improved reliability of the software systems. SYSGO has started to use fuzzing somewhere between 5 and 10 years ago.

This section provides an overview of basic concepts of fuzzing and is greatly inspired by the fuzzing book [Zeller et al., 2023], which can be source of more details.

Types of fuzzing

The evolution of fuzzers slowly leads to more complex tools and theories about them. Here, we describe the standard concepts of fuzzing.

We start with some normal fuzzer generating some completely random input. This works fine when we don't know anything about the input structure, but it is not the most efficient solution when the input needs to follow some syntax. In this case, most of the generated inputs might be invalid, so we are wasting much time because, the application usually rejects invalid inputs. We would like to generate invalid inputs, but in a way that the application considers valid. This leads to mutation-based fuzzing[7]. The generator takes some valid input and uses it as a seed for creating new inputs from the old ones. These fuzzers are aware of the input structure and trying to replicate it. But this approach still might generate invalid input.

[7] `https://www.fuzzingbook.org/html/MutationFuzzer.html`

One step further goes grammar-based fuzzing and generative fuzzing. It uses grammar to specify how exactly to generate input. This method restricts what fuzzers can generate. For it to work correctly, it requires some language to determine the grammar. With this language, the user can specify what should be generated. This can be useful when the input structure is known, and all invalid inputs are rejected. Then, we can use grammar to generate only valid inputs, which gives us more information about the application.

We looked at fuzzing from the perspective of the input, but how would we describe fuzzing from the point of view of the tested application. We might want to reconsider our approach based on how well we know the application structure. If we have no information, we call it black-box fuzzing. The approach lies in the random generation of inputs. At this point, we don't know anything about the application and need to resort to random generation without any hints. The black-box fuzzing seems very simple, but it has many downfalls. We probably test only the surface of the application by using this method and not reach any bugs hidden a bit deeper. This approach is often paired with some learning algorithms that try to learn how the application works inside and, based on that, generate better inputs, [Böhme et al., 2017].

On the other side, when we know how the program looks inside, we have white-box fuzzing. This approach uses the information about the application to cover the whole code base effectively. These fuzzers slowly collect constraints on which branches of code they visited. Then, with the help of constraint solvers, it generates input, so more branches are covered. In theory, this approach leads to full code coverage. But there is a problem with constraints. Either they might be unsolvable, or it takes too long to solve them. This wastes much time, so one might want to trade off targeted inputs for generating more inputs that are less targeted.

In the middle exists a gray box fuzzing. Gray box fuzzing is often tied to mutation-based fuzzing when it mutates the original input and the inputs it generated. If the fuzzer generates some input, it just scratches the surface, but with this method, the fuzzer goes a bit deeper every time it mutates the line of generated inputs. Unfortunately, this also has a big downfall in mutating already crashing inputs, but there are techniques to help with these issues. Grey-box fuzzers have the advantage that they still can use code analysis tools and, for example, keep priorities for inputs based on how many new code branches they discovered. This type of fuzzer is considered to be the most efficient.

3. Fuzzing applications

In the previous chapter, we mentioned different types of fuzzers. We have also noticed that their usefulness depends on what we want to test. Now, we look at some popular fuzzing frameworks that could help us with improving the ANIS testing. We introduce some frameworks shortly, and in the end, we summarize some ideas that helped us with the development of our fuzzer.

3.1 General purpose fuzzer

The General purpose fuzzer, or GPF, is a generative fuzzer built for UNIX systems. GPF focuses on the low cost of running a fuzzer and can generate more inputs per time unit than other fuzzers, [Sutton et al., 2007]. GPF has multiple modes that can help us in different fuzzing scenarios.

The most basic mode of GPF is PureFuzz, which is a generic black-box fuzzer that prints out seed. The seed is beneficial when we want to replay a whole sequence of fuzzed inputs. The main mode is a mutation-based fuzzer. As input, it takes a internet traffic of some protocol and starts mutating it. Some analytical tools also support the main mode, which puts it among gray-box fuzzers. If users need better support for fuzzing protocols, GPF offers a pattern fuzz. With this, GPF can detect what parts of the protocol are defined and what is plain text and further improve its functionality.

GPF is very flexible and has a significant advantage in allowing users to adapt their fuzzers according to their needs. On the other hand it is also very complex, and there is a lot to learn before configuring this tool for its first test.

It might look like we should aim for a bit more fuzzing tools to test our stack effectively. GPF proves to us that even simple random tools can also be effective. By generating random input with minimal configuration, GPF was able to find bugs in the Microsoft SQL server. So, having only a black box fuzzer can also lead to discovering new bugs.

3.2 American Fuzzing Loop

American fuzzing loop, or AFL, [Zalewski,], is probably the most known fuzzing framework that can be used to fuzz network traffic. It's a gray-box mutation fuzzer with some extensions that further improve the fuzzing experience. AFL needs users to provide an input sample that is later used for mutations and generation of new input. It also uses coverage-based feedback to improve the sample for mutations. If some input shows promising results, e.g., some crash or new response we didn't get so far, it will be mixed within the original sample and used later in the next run. The most significant disadvantage of AFL is the ability to only fuzz input as files or standard console input. It focuses more on C/C++ binary files than on network applications.

Over time, more and more fuzzers based on AFL were created. These fuzzers are built for some more specific purposes than the original AFL. The closest to our problem would be AFLNet, [Pham et al., 2020]. AFLNet follows the same

fuzzing principles as AFL but also checks state feedback to improve its fuzzing capabilities. It acts as a client application that sends messages to the server under testing. Based on the response codes, it identifies interesting regions of state space to explore. For monitoring and debugging, it saves the sequence of sent messages so the user can later resend the whole sequence and reproduce the crash.

AFL is a well-known framework with much success. American fuzzing loop was most of the time used for GNU applications, where it uncovered bugs in tools like Nginx, tmux, clang, and programming languages like PHP, Pearl, or Bash. It has also found success with major graphical applications like LibreOffice, Mozilla Firefox, Wireshark, or VLC.

3.3 SnapFuzz

Following up to the AFLNet is SnapFuzz, [Andronidis and Cadar, 2022], which tries to improve performance of the AFLNet and fix some of its problems. AFLNet fuzzers have problems with barriers for the users, because it still needs them to create clean-up scripts to reset state to the original one or specify delays for changing the state of the tested application. The delays are very important because if the state of the application is altered at an inconvenient times, it might lead to testing the same application states and lower efficiency of the fuzzer.

For performance improvements, SnapFuzz eliminates delays between server and client application fuzzer. The main delays are before initialization, before repeating unsuccessful communication, and wait time after each send or receive. Ignoring these situations might result in lost traffic and wasted time, but we can't send synchronization messages after each packet is sent from the client. Snapfuzz overcomes this through the additional controlling socket, through which comes information about the client's following actions.

3.4 Summary

We very shortly summarize what we are taking from these frameworks, that was useful for our development.

GPF is not a fuzzer that was directly created for fuzzing network traffic, but it is simple and effective. We also aimed for simplicity of our fuzzer as we didn't need to implement a fuzzer with a complicated structure. Even the most straightforward solution can efficiently find bugs inside the software. Another concept we would like to take from this is the seed for the generating inputs. The seed is suitable for generating the same input again, which can help fix mistakes found in the software.

AFLNet is interesting to us because of the state feedback. We wanted to use more prolonged periods of testing so we can see how it reacts with more extensive traffic amount. We could try to fuzz single packets processed by the ANIS, but we, in cooperation with SYSGO, decided to focus more on stressing the ANIS for more extended periods and change its inner state by sending larger amounts of network traffic.

SnapFuzz showed us multiple problems that we had to tackle. We had to make sure that the state of the ANIS changed before we validated it again. We also had to ensure that the different inputs wouldn't interfere with one another, so we had to consider cleaning up after testing. The last important part was synchronizing with the tested application. Running two communicating computers without any synchronization would lead to ineffective testing because, e.g., no packets would reach the other machine because it was not up yet. On the other hand, too much synchronization overhead leads to less efficient fuzzing, and we wouldn't be able to test as many inputs as we would like to.

Now that we have some ideas we would like to follow, we can proceed to design our fuzzing test.

4. Requirements and goals

Now that we have a better understanding of the problem and related technologies, we can move to specify our goals. We will discuss some technologies that we used for the development, discuss some requirements from SYSGO for the test and what part of the stack they could test and we will put everything together to define requirements for the project.

Underlying technologies

At first, we needed to think about how to deliver our software. One option was to ship it as a standalone product. Then, we would have complete control over the build process and communication methods. But we also needed to keep in mind that this test is a part of SYSGO's testing procedure for developers. It might be easier to use if it is integrated into one of the ANIS test suites. We also gained some supporting features of the test suites, so implementing this test was more effortless. That's why we decided to integrate it into the test suite because it is easier for SYSGO engineers, and we could also use already implemented features in the test suite. Integrating the test into the TFW restricted our options because we had to follow the test suite build process and learn more details about it. Because we used TFW, we had to follow the test structure of Master and Slave. Where the Master controls the test, and the Slave is the PikeOS platform running Slave application that is being tested. For us to test ANIS, we had to adapt the Slave application, which uses ANIS for network communication.

TFW is a platform-agnostic framework. For running test suites SYSGO has configured remote hardware to run their tests. TFW can also be configured to run the tests locally inside virtual machines, QEMU. Virtualization is helpful in debugging more complex tests like the one we wrote. By using virtualization, we also won't block SYSGO's hardware to test our project. The platform was not initially configured to run on QEMU, so one of our first steps was configuring everything correctly. With that, we were ready to start designing our software.

4.1 Scenarios

Probably the most crucial decision is what to test. We wanted to feed ANIS stack with network traffic that has the potential to break it. SYSGO provided us with some functionalities of the ANIS and types of network traffic that we could use to test ANIS. We call these types of traffic scenarios. Now, we discuss the provided ideas.

Before writing the test, we decided what scenarios we wanted to implement. ANIS supports only a subset of UDP network protocols. On top of that, it only implements the networking and transportation layer, so we didn't need to consider using protocols working on higher layers, e.g., HTTP or FTP. We now define some edge case behaviors of applications that can stress ANIS and use them as a base for our test. We named these network traffic behaviors scenarios, which we call shortly scenarios.

We will now describe what scenarios were offered us by SYSGO as desired for implementation and what scenarios we have chosen. We try to provide some insight on what the scenarios might be able to test.

One of the scenarios could be multiple applications using ANIS at once. Running multiple applications at once forces ANIS to work in a parallel environment. Parallel applications should be fairly easy to simulate with multiple listening or sending applications.

We also considered the volume and different types of network traffic. We start with the volume of traffic. Some applications might be waiting for an unending stream of data; we will call this an unthrottled scenario. This scenario exposes ANIS to a very strong network traffic, which needs to be handled.

Some traffic can also be fragmented, so we should consider that. Fragmentation is a complicated process that might break the stack, especially when fragments are missing, incomplete, or contain invalid information. We can try to force ANIS to fragment large packets or we can send invalid or incomplete packets to ANIS.

We are still staying at the networking layer. For UDP sockets, there exist multiple modes of handling the traffic; important for us are blocking and not blocking mode. The difference is that the socket blocks the process with blocking mode until a packet is ready to be received.

Sockets might have to be reused, in a scenario when operating systems have multiple applications that are using the same socket.

Let's move on from UDP traffic because ANIS supports other protocols like IGMP and ICMP. Different protocols could be an interesting stress test for the stack. These protocols run on a lower layer and are part of ANIS state space.

Following this, we mentioned support for broadcast scenarios, which might uncover hidden problems. Different broadcasting services could be interesting, especially with multiple ethernet cards connected to ANIS. We could even test ANIS, which deals with multiple ethernet cards.

Lastly, ANIS has some control over lower layer and ARP traffic. ARP traffic is another space of potential problems, especially the translation from IP to MAC addresses. ARP is necessary traffic for stack that is supporting IP, so some scenario looking into that would be welcomed.

After discussing with SYSGO, we have focused only on a subset of all scenarios. We focused mainly on UDP traffic scenarios because they are the typical use case of ANIS. We have also implemented support for multi-socket testing. Because we didn't implement everything, primarily because of the time it would take, we tried to make the test to be able to extend with other scenarios in the future, e.g., IMCP or IGMP scenarios.

4.2 ANIS configuration

Another important feature to consider is the configuration of ANIS. ANIS offers configurable parameters for the System integrator, so they can, for example, set up network interfaces or handle the ARP traffic. These parameters can influence network stack behavior and the results of testing. SYSGO wanted to explore this area as well. We only support the fuzzing of some parameters of the ANIS configurations. Some of the ANIS parameters shouldn't be fuzzed, e.g., the IP

address of the running Slave, because they might influence the test in a way that ANIS won't be tested at all. For example, an incorrect IP address could lead to not receiving any packets, and we wouldn't test anything. Some parameters are more attractive for change, like the size of socket buffers or resolution timeout for ARP, which might influence the ability of ANIS to handle network traffic. Configuring the ANIS also gave us another task of correctly setting up the ANIS. We describe the list of fuzzed configuration parameters and how to configure them in Section 5.8.

4.3 Success validation

The last thing we need to discuss is how to validate the test. Mainly how to detect that the ANIS didn't fail.

With network traffic, there are a lot of possible downfalls, and it doesn't have to be a fault of the ANIS; it might get lost on the way. We can not assume that everything will go smoothly.

When the test is evaluated as failed, even though ANIS worked fine in the provided scenario, someone will have to spend time analyzing the problem. For example some configurations are making it harder for ANIS to intercept packets, like not enough space in receiving buffers. If something like this happens, the error in the test is raised, and someone will have to analyze it. Therefore, we aimed for loose passing criteria.

Because the main concern right now is the stability of ANIS. We are looking for segfaults, race conditions during long runs inside ANIS. It is sufficient for us that ANIS, after the execution of scenarios, is working properly. That means that ANIS didn't crash and can send and receive some traffic, in our case, a simple UDP packet. Some ANIS configuration might endanger the successful control packet send and receive of the UDP, so we will have to find such a configuration and prepare a mechanism to recognize them and ignore false positive findings.

4.4 Summary

Let's summarize what we have decided on so far. The main goal of this project is to create a testing tool that checks ANIS under different loads of network traffic and checks if the ANIS can work properly after dealing with them. This tool has to be integrated into one of the SYSGOs test suites for ANIS verification.

We will use a virtualization machine, specifically QEMU, to simulate the target hardware under which the PikeOS will run. We will have to adapt the test suite to work with virtual machines.

We want fuzzing to help us create different combinations of scenarios. From now on, we will use the term schedule for a set of scenarios in the network during some period, combined with time, when these scenarios start and end. A single scenario is some network traffic happening at a given time. For example, a stream of packets being sent simultaneously is a scenario. To better test the ANIS, we implement multiple different scenarios and create a structure that supports additional scenarios to be implemented. We also create a set of scenarios focusing

on some UDP traffic scenarios that can occur in the real world. The following list describes functions of the stack we should tackle with the implemented scenarios.

1. Unthrottled stream of data
2. Multiple socket applications
3. Blocking mode for network sockets
4. Sending of unfinished UDP packets

The final part of the project is a mechanism that validates the success or failure of ANIS after being exposed to the generated network traffic. The main focus will be on crashes and segmentation faults. This tool must be complex enough to catch such errors but simultaneously simple so it won't report false negative test results.

5. General architecture of test

This chapter follows the architecture of the test and further describes its components and the interactions between them. At the end, we will look at some scenarios that are implemented. But first, we will briefly describe the first unsuccessful attempt, which greatly inspired the final architecture.

5.1 First attempts

At first, we tried to set up a basic Master Slave scenario as it is common for test cases inside TFW.

We must realize that we are in a situation with multiple separate computers. If we send some traffic from the Master while the Slave is still loading up the configuration, all that traffic will just miss the Slave. This way, we won't be able to test anything, or maybe just a small part of the possible state space. We wanted both machines to cooperate on the testing task.

The need for cooperation led us to some synchronized communication between Slave and Master. We solved synchronization through TFW serial line because it is a reliable way to pass information. Now, we will have to decide how to create the network traffic.

The first idea was to generate the traffic that was supposed to be sent from Master. The Master generates the traffic in single packets. After generating the packet, the Master synchronizes with the Slave and sends the information about this packet to the Slave. The Slave reads the information and executes some behavior, and the same does the Master. They can communicate through TFW serial line. They send and receive that packet and repeat the same process. This way, we are guaranteed that the Master and the Slave do the same thing, thanks to synchronization. However, the amount of synchronization is a big downside. We created a constant synchronization traffic on the serial line that needed to be processed. For every packet, we have four steps: generation, synchronization, execution, and synchronization. So much synchronization is inefficient, but this idea was our starting point.

We needed to get rid of the synchronization overhead. First, we realized that the test has to end at some point. If we were just sending packets until infinity, we wouldn't have time to evaluate if it did something to ANIS. So, we had a bounded timeline for when all the traffic needed to be sent and received. We can precompute our data packet before the test starts when we have a bounded timeline. Then, we need to send the generated data to the other side. In our case, we were generating at the Master and sending data to the Slave. We have eliminated some synchronization because we don't need to synchronize for the transfer of each packet. After the data is generated, synchronization can happen so the Slave can receive everything correctly. The last synchronization points will be when the test starts and after the test ends. We don't need any synchronization when the machines send all the packets simultaneously. Every packet has a time when it is supposed to be sent after the start of the test. Both machines know this, and if they synchronize before executing the first packet, they can then use this point to compute the next time to execute the next one, and because the machines

are synchronized, they agree on that time. We have a set of packets and times when they are supposed to be sent. But this implementation had its problems. We had packets and when to send them, but we couldn't support opening multiple sockets or switching socket modes. On top of everything, some scenarios were destined to generate enormous traffic, e.g., the Unthrottled scenario. For this, we would need much memory on the Master side to store all the information about generated packets, e.g., when to send each packet, and we would need to pass all the data to the Slave. It would be better for us to define, for both sides, how to generate the traffic. We could also put information about, for example, the sockets and information on how to create the traffic.

5.2 Schedule and scenarios

We need to define information about when and what traffic should be generated or something else executed on the machines. We have defined two structures to store the information: scenario and schedule. Let's now describe them.

Scenario

A scenario is a defined behavior of two network endpoints communicating with each other. Some might generate UDP packets, others IGMP packets. Their behavior is entirely up to the scenario developer and the network stack's capabilities. Every scenario should define behavior for both the Master and the Slave. One scenario might want the Slave to send packets and listen. Another might want the Slave to listen to incoming traffic. In some cases, e.g., ICMP traffic, Slave behavior can be omitted because ICMP is being resolved only in ANIS and doesn't reach applications using it.

To improve the space coverage by our test, we can use a fuzzer to parametrize our scenarios. For each scenario, we set parameters that will be fuzzed. The parameters create even more options that will be used to test ANIS.

The scenario structure can't exist on its own. It also needs some code that takes the scenario structure and uses the data from it to generate or receive traffic.

Schedule

A schedule is a set of scenarios with additional information. For every scenario, the schedule knows when the scenario starts and stops.
An application that holds this schedule should be able to start the scenarios on time. This application, holding the schedule, had to be implemented for the Master and the Slave in our case.

Use of the structures

Our test now can take a schedule data structure and iterate over it. Each iteration takes the next scenario, looks at what scenario it is, and executes it.

If we want to have multiple scenarios at the same time, we can start multiple threads with different scenarios. Threads create a modular approach because the

threads and their scenarios do not depend on one another. The only shared thing between the scenarios is the ANIS.

When and what packets need to be sent can be defined within the procedure. This leaves the schedule only with the type of scenario, the time when the scenario starts, and possibly some additional data for a specific scenario. The behavior of these scenarios is parametrized, and these parameters can be fuzzed, which gives us a wider variety of scenarios. The only disadvantage is that we will have to define behaviors for both the Master and the Slave because they might differ.

5.3 Test design

We can put all our ideas together and describe how it was designed. We've already mentioned that we need a bounded time frame for our test, so we split the logic into rounds. One round will consist of three main activities: Fuzzing new schedule and configuration, running the schedule, and evaluating the state of ANIS. We will run these rounds in a loop and give an option for the user to set how long this loop should run in seconds.

In every round, the fuzzer must fill the schedule with scenarios and scenario parameters and generate the configuration for the ANIS. This information needs to be shared by the Slave and the Master, so the information has to be transferred to both devices. The machines use the TFW serial line for communication because it is a stable and reliable way to transfer data. Both devices execute the scenarios when they agree on the current round's information. They need some control loop that checks if it is time to run the next scenario and a way for both the Master and the Slave to execute it simultaneously. After the whole schedule is executed, the state of ANIS needs to be evaluated. We talk more about evaluation in Section 5.10. Evaluation completes one round of testing, machines reset everything to the original state, and the next round of testing can start. The pseudocode of the main loop is visualized in Algorithm 1.

Algorithm 1 Master main loop

```
procedure MASTER_MAIN
    max_time ← test_parameters_max_time
    round ← 0
    fuzzer_load_configuration()
    while current_time ≤ max_time do
        schedule, anis_config ← fuzz_this_rounds_data()
        prepare_for_round(schedule)                    ▷ Set up sockets etc.
        synchronize_with_Slave()
        pass_data_to_Slave(schedule, anis_config)
        synchronize_with_Slave()
        run_scenarios_in_schedule(schedule)
        wait_for_Slave()
        run_health_check()
        evaluate_round()
        clear_after_round()
```

5.4 Components of the test

When designing individual test components, we tried to split them into partitions, at least on the Slave side. This division is essential because we want parts of the test to be independent. The component responsible for communication with the Master shouldn't interfere with the components responsible for sending the traffic and validating the state of ANIS. The Slave side can take advantage of the partitioning feature of PikeOS. Interference of different components is not such a big issue for the Master because the Master is not the tested application. We split it into logical components so the concept is easier to grasp.

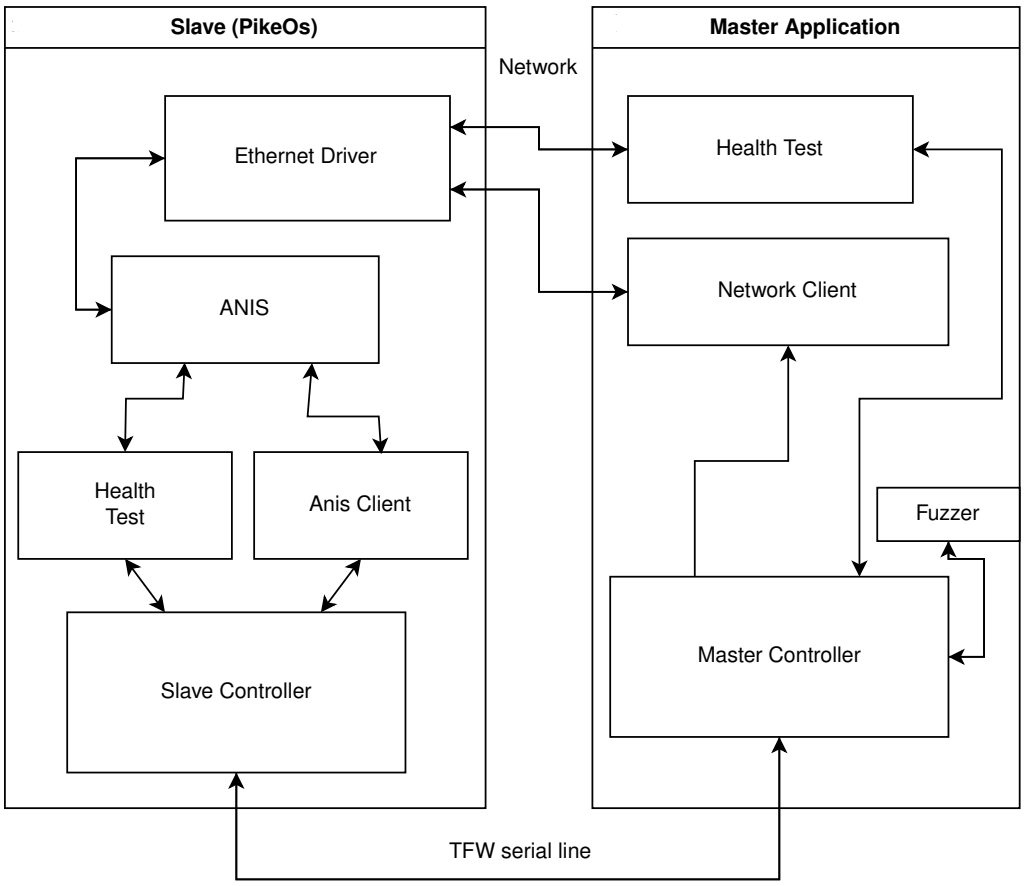

Figure 5.1: Architecture of the test

In Figure 5.1, we show the general layout of the test.

The test has two parts. The first part we call Master application, or shortly Master. The Master is an application running on a Linux-based operating system. The purpose of the Master is to play the role of devices on the network. The Master and the Slave are connected on an isolated network, so other devices can't influence the test. The Master sends and receives packets or provides other data that could appear on the network. It also has two other particular roles: fuzzing and health check. The logic of the Master is split into separate threads that should not share data because we don't want them to influence each other. There is a controlling thread, or the main thread, another thread responsible for

executing scenarios, and a thread responsible for the health check. The fuzzer is implemented within the main thread, even though it is separated in Figure 5.1.

The second part is the PikeOS system running on the QEMU virtualization layer. We call it Slave. Because PikeOS allows us complete separation of resources, we can split everything the Slave does into partitions. We have two partitions using ANIS to communicate over the internet with Master. First is the ANIS Client, which we will use to run different network scenarios. ANIS Client is a partition that runs on this instance of PikeOS and generates or consumes some internet traffic. Health Test checks if the ANIS is still running after execution of ANIS Client. Like Master, Slave also has a controller, Slave Controller, which is a Slave application from the TFW perspective that communicates with Master and controls the test on the Slave side. The Slave Controller has more responsibilities than the Master Controller. It is responsible for executing scenarios at the right time. Then, it sends information to the Master and ANIS Client that they should start executing the next scenario.

Now, we look at the components in more detail. We discuss their responsibilities and how they look from the inside in Sections 5.5 to 5.7. These sections also cover the behavior of scenarios and how they fulfill our requirements. In Sections 5.8 to 5.10 we discuss our fuzzer, how it works, how to configure it, and what can be fuzzed, and we also take a look at the evaluation of the state of the ANIS.

5.5 Master components

This section describes the components of the Master module of the test, how they work, and what needs to be implemented.

Fuzzer

The Fuzzer, as the name suggests, is responsible for generating random data for the test. The fuzzer is a component that first loads its configuration, and then it can produce data on the applications request. We need some control over what is fuzzed. The control is done through configurable parameters that Fuzzer initially loads. After that, Fuzzer generates two different data sets. First is the schedule with all scenarios and their execution times. The second is a configuration for ANIS in the current round of tests. Further, we would like to be able to generate the same data for debugging purposes. Therefore, the Fuzzer accepts some seed by which it generates the data. We want our Fuzzer to be highly configurable, so we had to find a way to configure it. We consider this to be a part of the Fuzzer and will look more into that in Section 5.8.

Network Client

Network Client is just a thread that executes scenarios. It waits for Slave to send some command for execution of next scenario. The Network Client first needs to distinguish between different types of scenarios and then, based on the type of scenario, start acting by predefined behavior. The Network Client starts for every scenario in its thread. We can consider running scenarios part of its logical partition, even though they are running in their own separate threads.

The logic of the thread execution was initially implemented in the main thread. However, there is another reason why we decided to move the Network Client into a separate thread, except for modularity. The issue with this approach was with the blocking mode of the sockets. If no more traffic was coming to the socket, the thread was blocked, and there was no way to exit it. Later we avoided this issue by monitoring sockets, which are internally implemented as file descriptors, with *select* method, but just in case it appears again with some new scenario, we leave this as a separate thread. This way, we can always kill this thread and proceed to the next round of testing.

Health Test

During the run of the test, we also want to check if the ANIS is still able to receive and send packets. For that purpose, the Health Test was designed. For simplicity, we choose the UDP packet as our primary indicator that ANIS can still communicate. This packet is sent to Slave in multiple tries. Multiple tries should minimize chances for the test to fail due to previous traffic that overwhelmed the network stack. The traffic could be stuck in buffers, and the new incoming packet would be rejected. A waiting time makes the check more tolerant, resulting in fewer false positives for ANIS failures. After sending each packet, the Health Test must wait for the response. If the response was received, the test is considered as passed. If no response is received even after multiple tries, we consider the test failed unless other conditions are met. We will discuss those conditions in Section 5.10.

To keep the modularity of the code, and because we also wanted health check to have as little influence from the scenario threads as possible, we decided to run it in a separate thread.

Master Controller

The Master Controller is responsible for the execution of the test on the Master side. For individual tasks, it uses previously described partitions.

For each round, the Master Controller has to create a schedule and configuration of the ANIS. This is done via the service of the Fuzzer component. Then, the Master Controller transfers the schedule and ANIS configuration to the Slave through the serial line so the Slave operates with the same schedule. Master then synchronizes with Slave. After this, the Master Controller starts the Network Client thread responsible for executing scenarios. The Network Client then waits for a signal that comes through the serial line from the Slave Controller. On receiving the signal, the Network Client executes the next scenario if possible. After enough time has passed for the schedule to be executed whole, the Master Controller once again synchronizes with its Slave counterpart. Successful synchronization means the schedule execution was completed, and it is time to check if the ANIS has survived via the Health Test thread. We went through everything that needed to be done for one round. The last step for the Master Controller is to continue to the next round until the total time set by the developer has expired.

The Health Test concludes all components of the Master side. Now, we will

talk about the Slave components. Some are counterparts for the Master components, so they look very similar.

5.6 Slave partitions

This section describes the components of the slave part of the test, how they work, and what we had to implement. Every partition lives in its own PikeOS partition, so they don't share any data or resources. ANIS also lives in its partition, so if it fails or other non-controlling components fail, the components can be restarted for the test to continue. If there is a need for sharing, we use the PikeOS communication methods messaging queues to transfer data from one partition to another.

ANIS Client

ANIS Client is a counterpart to the Network Client thread in Master. After start-up, it needs to receive the schedule and then wait for instructions on when to execute the next scenario. These instructions are coming as signals from Slave controller that is described in Section 5.6. On signal received, it reads the next scenario and by its type, it decides what to execute. The ANIS Client functions as a standalone application inside PikeOS. In case of failure, the whole partition can be shut down, and the test can continue with the next round of fuzzing.

Health Check

Health Check on Slave is a standalone component used to validate whether the ANIS is still working. It uses the same partition with ANIS as ANIS Client. First, the component sets up a socket for communication. Health Check listens on that socket for some time, and upon receiving a packet, it sends it back to the source. Like the Master side, the packet is also sent out multiple times to increase the chance of receiving it on the other end. This component also stores information if the socket could have been opened. This information is passed through the Slave Controller to the Master, where it is used in the evaluation of the test round and plays a role in evaluating some edge cases, which we discuss in Section 5.10.

Slave Controller

The Slave Controller is technically the already mentioned SlaveApp and runs inside its partition. It is the part with the biggest responsibility of keeping track of time and sending signals to the Master Network Client and Slaves ANIS Client. When the signal is sent, both clients on the Master and Slave side should execute the next scenario in order. For sending these signals, the Slave Controller needed two communication channels, one with the Master and the other with the ANIS Client component.

At the beginning of each round, the Slave Controller receives the schedule and configuration for ANIS from Master. Then, the Slave Controller needs to give the configuration to ANIS and restart its partition so the changes are applied. The

configuration is delivered to ANIS through a file in shared memory. The Slave Controller overwrites this file at the start of every round and restarts ANIS. On start ANIS picks up this file and loads the configuration. More technical details can be found in Section 6.3. After that, the ANIS is set and ready for this round of the test, the Slave Controller proceeds to start the ANIS Client component and transfer the schedule to it.

The component's biggest challenge was to execute scenarios on both sides at the same time. We decided to put the executor into the Slave Controller. If the starting scenario logic were implemented on the Master or in the ANIS Client, it would lead to a difference in time, that one informs the other about the execution of the next scenario. If the controlling logic were in the Master, the execution in the Network Client would be instant. However, because there is no direct communication line to the ANIS Client, the scenario would be executed with the delay of $t_{master\_to\_client\_controller} + t_{clinet\_controller\_to\_ANIS\_client}$. We can use the same argument for the case where the execution logic is implemented inside the ANIS Client. We have tried previous approaches and settled down on the Slave Controller being the one that controls the execution and send controlling signals to both the Slave and the Master because it has the smallest delays between the execution of Master and Slave.

After the controller goes through the whole schedule, there is some period of waiting, so the ANIS Client and Master's Network Client can finish executing all scenarios, and ANIS has enough time to process all traffic. In the end, the Slave Controller synchronizes with the Master and starts the Health Test partition so the round can be evaluated. After execution, it can wait for the Master to send data for another round of testing.

5.7 Implemented scenarios

Next to the components, we also had to implement scenarios covering the desired space. We now describe the implemented scenarios, what they cover, and some implementation details.

Multi socket variant

Before we talk about the actual executable scenarios, we discuss the coverage of multiple sockets in use. One of the desired things to test is running multiple scenarios using different sockets. For this purpose, every scenario can run on a deliberate number of ports. How many ports the scenario runs is decided by the randomness of the fuzzer. This forces ANIS to listen on multiple sockets that are all bound to different ports, keep track of fragmentation on those ports and correctly distribute complete packets to correct devices reading from those sockets. Such behavior doesn't rely on underlying network traffic. So, we have decided to join it with other scenarios that must generate some traffic. Therefore, every running scenario is extended for a list of ports on which it should run. The multiple ports extension covers the requirement for multiple simultaneous sockets.

Scenario 1: *Scheduled*

Now, we can proceed to the first executable scenario. In this scenario, we simulate applications that send packets occasionally. Both Master and Slave have a sending and receiving part of the application. How many of these they have depends on the number of sockets to use. The number of sockets is fuzzed, as mentioned previously. This scenario also has a predefined list of UDP packets to send. Each packet has its length and time when it should be sent. The *Scheduled* scenario was the first scenario that we have implemented. It doesn't stress the stack with the number of packets but still can send packets fragmented, or packets sent at some particular time can also stress the stack, especially in combination with packets from other scenarios.

Scenario 2: *Unthrottled*

Our goal with this scenario is to overfill ANIS with packets and test how it is handled. The *Unthrottled* scenario has only one direction for the stream of packets. This means that one side is listening, and the other is sending. The one-sided traffic could seem like it restricts the state space for the test because the scenario of sending and receiving this type of traffic is missing. But because we are running scenarios concurrently, there is a chance to generate two scenarios, one sending from Slave and one receiving on Slave and vice versa. The two simultaneous scenarios cover the described state space for ANIS.

We have also extended this scenario to test the requirement for blocking UDP sockets. We decided to merge this requirement into this scenario. A parameter if the sockets should be blocking or not extends this scenario and is used as a flag for the receiving and sending packets. This way, we won't have to create another executable scenario to cover this requirement.

Scenario 3: *Unfinished packets*

The last scenario we implemented considers a faulty network that sends unfinished UDP packets. This scenario could happen on overwhelmed networks when packets need to be fragmented for transmission. These fragments might get lost, and the receiving machines have to deal with waiting to complete the packets. For this scenario, we had to work on the lower layers of the network. If we want to send an unfinished packet, the easiest way would be to not send the last fragment on the underlying layer. We needed to fragment the UDP datagram to ethernet frames and intentionally avoid transmitting one of the fragments.

We are not looking to test how well ANIS can send fragments of the packets. This is technically covered by sending packets larger than the network's maximal transfer unit, MTU. We want to discover how ANIS is affected by holding the packet fragments, so we only implemented one way for this traffic from Master to Slave.

The *Unfinished* scenario concludes the list of executable scenarios we have implemented. Every single one of the scenarios has a set of fuzzable parameters that makes the scenarios different in the run-time. In the next section, we discuss the fuzzer, how to configure it, and what parameters of the test are generated by it.

5.8 Configuration parameters for fuzzer

We have designed and implemented the components of the test and some scenarios. This section describes in detail the fuzzer, especially how to configure the fuzzer so that the developers can influence values generated by the fuzzer and, through it, what is tested. We also go through the parameters that are being fuzzed for the current test.

The requirement for our fuzzer is a way to parametrize it. This is helpful as a tool for developers to test some specific configuration or to restrict values that can be generated.

Our configuration is defined within a provided file from TFW. This file gets parsed at build time, and all the values are stored as C headers in a separate header file that can be included in our test. This file is always inside the integration project with PikeOS, so we always have the values on hand. It is quite an easy way to integrate the configuration into the test suite, but this was not our first attempt. The file is just a set of key-value pairs. Each pair has a name and some value that is parsed. The build process creates a header file with macros where the key is the macro name, and their values are the defined macro values.

The first attempt was to have the configuration as a standalone file. This file would be shipped with the test and after loaded as in our program. This approach was scrapped because the test suite moves all relevant files into new folders during the build process. We would have to account for different paths and possible mistakes with copying files. For this, we would need to have a parser for this file. Our test also has two parts running: The Slave and the Master. If we ever needed to have parameters in both the Master and the Slave, we would need to parse it twice. Also the file would need to be shipped within the PikeOS binary, which we found to be a bit challenging and that's why we decided to use the other approach.

When we decided on a way to pass values into the program, we needed to define what we are passing and how the configuration looks. We have configuration parameters. These are the parameters developers set in the configuration file. The Fuzzer takes these parameters and updates its configuration, which influences the generating of the schedule and ANIS configuration. The schedule and the ANIS configuration are technically just a set of predefined values that change each round. These values we call fuzzable parameters.

The value of every fuzzable parameter, either scenario or ANIS configuration parameter, can be influenced in a couple of ways. The first way is to restrict the range of values that can be fuzzed. So, we set upper and lower boundaries for every parameter. The second way is to specify probabilities of generating some specific value. So, we had a bounded interval of integers and needed to adapt probabilities of generating the value. We used probability functions. We just needed to represent them efficiently.

The first attempt to represent the distributions was to use multiple key-value pairs that would need to be set for each parameter. For example, we take the parameter for fuzzing the number of buffers. The first attempt was to create one C header specifying the upper bound and one specifying the lower bound, and then we needed to represent the distribution. We came up with the idea of splitting the function into multiple values. One value would tell us the type

of distribution: uniform, normal, or geometrical. The other part would then specify parameters for the normal distribution function. The problem with this solution is that all values for all the distributions need to be set because they are all C headers. This system creates a lot of unused values in the configuration. For example, if the type were uniform distribution, we would still need to define parameters for the case when it would be a normal distribution. We also need multiple parameters to adapt one distribution function, which was not desired. On top of everything, we couldn't manipulate the distributions completely. For example, we could only use geometric distribution for the entire range of values, but what if we wanted to have some values generated by geometric distribution and the rest to follow a uniform distribution? This design didn't support this case at all. Another problem was that the number of parameters we needed to set grew a lot, so we tried to get something more compact in terms of the number of parameters and more flexible to configure.

We decided to set up some form of grammar, which we can use to generate the description of the distribution functions that would suit SYSGO's needs. The description is a string that the fuzzer parses and updates its configuration according to that. With the grammar, developers can define distributions for all fuzzable parameters. Now, we describe how the description looks and formalize the grammar.

Grammar and function description

Every fuzzable parameter has a string specifying the distribution for it. The description splits the whole domain of the distribution function into smaller subdomains. With smaller parts, we can control how the values are generated. We can divide the entire range into multiple parts. Each part can be generated from a different distribution, e.g., values from interval $[1, 10]$ are generated by normal distribution, and the rest of the values are generated uniformly. We can use the following example of ANIS configuration value for the number of sockets to show the description.

```
fuzzer_num_of_sockets "{U(50\%, 4, 8); U(6\%, 1, 3); C(44\%, 50)}"
```

Every subdomain is then defined as a range of values or a single value. A semicolon separates subdomains. We use the name Ranges, and we call single value Elements. From the example in Section 5.8 a subdomain is `U(50\%, 4, 8)`. Every Range or Element defines the upper and lower bound of the interval and the distribution function for that interval. Our chosen subdomain is within interval $[4, 8]$. The value has 50% probability to be generated from this interval, and the U in the beginning symbolizes the uniform distribution. The configuration string consists of a combination of Ranges and Elements, which are separated by a semicolon. Even tho the implemented parser for the description isn't too strict, and a slight mistake won't necessarily break it, we recommend following key points that we describe in Section 5.8.

$$
\begin{aligned}
&\# \rightarrow \{s\} \\
&s \rightarrow s; s \mid t\,(p\%, b, e) \mid C\,(p\%, b) \\
&t \rightarrow U \mid G \mid N \\
&p \rightarrow x \in [1, 100] \\
&b \mid e \rightarrow x \in \mathbb{N}
\end{aligned}
$$

Figure 5.2: Grammar for fuzzer configuration

Grammar

Let's talk about the structure of the description. We have defined a grammar, which is generating such language. Following this grammar ensures a valid configuration of the fuzzer. Let's have a set of terminal symbols $T = \mathbb{N} \cup \{U, G, N, C\} \cup \{\%|;|, |\{|\}|\,(|)\}$, [1] non-terminal symbols $N = \{s, t, p, e, b\}$, starting as an empty string #. Figure 5.2 contains the rules of the grammar. To shorten the rule set, we use $non - terminal \rightarrow x \in [a, b]$ to symbolize, that the non-terminal can be extended to a single value x, that is, from an interval of whole numbers from a to b included.

Descriptions generated by this are sufficient to generate configuration for the fuzzer. The fuzzer has one more restriction on top of this grammar that we decided not to encapsulate in it. The sum of percentages of all subdomains has to be exactly 100. This rule is validated after the fuzzer loaded its configuration, and the developer is notified when the sums don't match. For better understanding, we go once more over what the grammar says.

The configuration value is a set of subdomains that create the distribution function's domain. Every subdomain consists of one from Ranges or Constants. Let's start with the Constant structure.

Every constant starts with the capital letter C, followed by an opening bracket. After the bracket, there have to be two valid integer values separated by a comma. The first number is expected to be a percentage value, and the second should be the value we want the Fuzzer to generate.

Ranges have a similar structure to Constants. It starts with capital letter U, G, or N, followed by an opening bracket. The starting letter is a symbol for some known probability distribution. In our case

U Uniform distribution

G Geometric distribution

N Normal distribution

This time, there should be three numbers split by commas. The first number is again the percentage value. The next two numbers specify the range from which the number should be generated, including those values.

[1] The symbols in last set are split by a | for better readability

Let's finish talking about grammar structure with an example of the probability distribution would look like if it were specified by our language like

```
fuzzer_num_of_sockets "{U(50\%, 4, 8); U(6\%, 1, 3); C(44\%, 50)}"
```

In the table 5.1 we can see the exact chances to generate the values by the fuzzer.

value	chance to generate
1	2%
2	2%
3	2%
4	10%
5	10%
6	10%
7	10%
8	10%
50	44%

Table 5.1: Probabilities of generating desired values

Parsers recommendations

We have implemented a parser for our grammar to store the configuration in our program. In this section we would like to point out some tips that developers should follow when writing the fuzzer configuration. First, we talk about what we shouldn't do when using our grammar because the implemented parser might be unable to parse it correctly, either because it doesn't follow the grammar, or the parameter won't be parsed correctly during the build time. The parser is fairly simple, which is another reason for the restrictions. Let's sum it up in a few points.

- There has to be at least one Range or Constant value in the string
- Opening bracket has to be immediately after a capital letter
- Sum of all percentage values must be equal to 100%
- Whole string has to be enclosed in a pair of '
- There can't be " in the string
- Multiple ranges or elements need to be split by a semicolon

Now, we know what to do, but there is no standard for the readability of the string. For that purpose, we have created a set of rules that, if not followed, will still be correctly parsed, but the string might be unreadable.

- Whole string should be enclosed in { brackets

- First value in both element and range should be followed by % symbol
- We should always use spaces after commas and semicolons

In the end, we would like to introduce the constant parameters. Sometimes, it is desired to generate the same value every time. This language allows this with a little hack when we want to set one parameter to a static value. In such case, we can use syntax `'{C(100%, X)}'`, where X is our desired value.

5.9 Fuzzer parameters

Here, we describe all parameters for the fuzzer and how the generated values influence the test. The tables on the following pages show all the parameters that can be set for the fuzzer. If necessary, the set of supported parameters might be extended with new parameters, especially for new scenarios. The set is described in the following tables. The parameters are usually structured in rows as the configuration parameter's name and what it influences, and optionally, the table contains a third column with additional information. Table 5.2 describes parameters passed to the ANIS partition and used for the ANIS configuration. In Table 5.3 are values directly linked to the schedule structure, e.g., how many scenarios will be run. Table 5.4 contains parameters that all scenario types share, e.g., when this scenario should start. In Table 5.5 are parameters that are linked to the specific scenario type, e.g., the number of unfinished packets to send when the unfinished scenario is running.

Parameter	**Meaning**	ANIS config name
$fuzzer\_num\_of\_sockets$	Number of sockets of ANIS that can be open at once	*numSockets*
$fuzzer\_buffers$	Number of buffers, that store incoming traffic	*numBuffers*
$fuzzer\_mtu$	Maximal transfer unit of the network	*mtu*
$fuzzer\_peer\_cache$	Restricts cache to track unique IP headers from different sources. Influences ability to identify duplicate fragments	*peerSize*
$fuzzer\_arp\_table\_size$	Number of entries, that can ARP table hold	*arpTableSize*
$fuzzer\_arp\_queue\_length$	Maximal number of waiting packets with unresolved ARP address	*arpQueue*
$fuzzer\_arp\_resolution\_timeout$	How long ANIS waits for address to be resolved to ARP	*arpResolvTimeout*

Table 5.2: Fuzzable parameters for ANIS configuration

Parameter	**Meaning**
$fuzzer\_schedule\_test\_length$	How long should 1 round of test last in ms
$fuzzer\_scenarios\_count$	How many scenarios will be run

Table 5.3: Fuzzable parameters for schedule

Parameter	**Meaning**
$fuzzer\_scenario\_type$	Type of generated scenario
$fuzzer\_scenario\_from$	When does the scenario start
$fuzzer\_scenario\_to$	How long does the scenario run
$fuzzer\_scenario\_master\_ports\_count$	How many ports should Master side use
$fuzzer\_scenario\_slave\_ports\_count$	Row many ports should Slave side use

Table 5.4: Fuzzable parameters influencing all scenarios

Parameter	Meaning	Scenario type
$fuzzer\_sc\_scheduled\_requests\_count$	How many requests for scheduled scenario will be generated	scheduled
$fuzzer\_sc\_scheduled\_requests\_size$	Size of UDP request. Max 65536 bytes	scheduled
$fuzzer\_sc\_scheduled\_requests\_direction$	Who sends this request, Master or Slave	scheduled
$fuzzer\_sc\_scheduled\_requests\_time$	When to send the request, within the time frame of the scenario	scheduled
$fuzzer\_sc\_unthrottled\_direction$	Which side generates stream of traffic	unthrottled
$fuzzer\_sc\_unthrottled\_blocking$	Should blocking mode be used in unthrottled scenario	unthrottled
$fuzzer\_sc\_unthrottled\_step$	How long should one step of sending data last in ms	unthrottled
$fuzzer\_sc\_unfinished\_packets$	Maximal amount of unfinished packets that will be sent	unfinished
$fuzzer\_sc\_unfinished\_time$	Maximal time in ms of sending packets during unfinished scenario	unfinished

Table 5.5: Fuzzer parameters for specific scenario types

5.10 Evolution of health check

Let's talk a bit about how we ended up with the Health Test component in a separate partition. We have already mentioned that we decided to go for the most simple check for the ANIS so the test doesn't report many false positives.

We originally designed the Health Test to be a part of the ANIS Client inside PikeOS. On the Master, it was just a function that sends one singular UDP packet to the Slave. However, we have encountered several issues. We have partially mentioned some issues before, but let's take everything from the beginning.

First, the UDP is not a reliable protocol and packets could get lost on the track or the endpoint buffers could be full when receiving it. The track for the packets is not complicated; it is just two devices connected through a switch, but the full buffers showed to be the real problem. We partially solved this by sending multiple packets to the Slave, which helped in lost packet situations. Unfortunately, this was not sufficient. We implemented similar behavior on the other side as well, because the Slave was also sending only one packet back to the Master. We have created a process of multiple packets being sent and received on both sides. Because our only concern was the stability of ANIS, we decided that one packet that comes through is enough for success. One successful packet shows that the stack is indeed functioning and didn't crash during the execution of the schedule. We were able to get more stable results, but the problem with the full buffers was still there.

The issue was that the ANIS held incoming traffic for quite some time. Because the ANIS releases buffers on closing the socket, we partially solved it by closing sockets after the scenarios finished using them. ANIS cleared its buffers for those sockets, and it helped with the false positive findings. Unfortunately, the test was still reporting some false findings, but at least at a much smaller rate.

Our final decision was to move the health check to a separate component. The reason for this was that we have discovered the scenarios that were using blocking mode were not finishing the threads because the thread was blocked in the receive or the send call, which is the desired behavior for blocking mode. Because the scenario never ends in this case, the sockets never close, and the buffers can still be full.

We needed some way to stop the whole process. We have decided to choose an option of moving the entire code to the separate partition. This partition has its resources completely separate from ANIS Client and can be controlled by it. But we still needed to ensure that the buffers were clear and ready to accept traffic for the health check. But because the code is now in a separate component, the Slave controller can shut it down. The sockets are automatically closed on shutdown, so the buffers are cleared.

With this step, we have created an isolated environment for our evaluation process, so it can't be influenced by any traffic that comes from the execution of the schedule and reliably validates the state of the ANIS. The unexpected shutdown had one unintended positive effect for us. By aggressively shutting down all sockets without any chance for ANIS to clean up or prepare for the shutdown, we extended the covered state space. The shutdown covers some resource management logic, which is welcomed extension of tested state space.

Special configurations

Earlier, we mentioned some specific conditions for which the Health Test reported false positives. The error was caused by some unfortunate, but never less valid, configurations of the ANIS fuzzer.

When the fuzzer was generating values for the ANIS configuration, one of the valid values, is a very low ARP resolution timeout. When the ANIS didn't have information about, which MAC address belongs to which IP address, it couldn't send out response packets for the health test. Not enough time to resolve IP led to false positive results. SYSGO developers didn't want to omit this state space, so we had to create a mechanism to work around it. So far, we haven't discovered any other special configurations, that would lead to false positive reports, but in case they are found, we recommend using a similar approach in case other edge cases are discovered in the future.

We solved this problem by watching for these special conditions and lowering the conditions for pass. In cooperation with SYSGO, we omitted the sending of the packet, and the only condition that validates the state of ANIS is opening a socket. If the socket opens successfully, ANIS still has to function. Otherwise, it returns an error. We then created a communication line from the Health Test component through the Slave controller to the Master Controller. Master gets the information if the socket was opened correctly. Then, it checks for the special configuration. In our specific case, when the ARP resolution timeout is less than five milliseconds. If the special configuration is set, the Master then evaluates the test only of the information that it got from the Health Check on the Slave side. The UDP packet test is completely ignored.

6. Technical implementation

Here, we look at the details of the test implementation. We comment on some challenges we faced during the development and explain some unconventional choices in the design and implementation. We were developing for PikeOS native personality, which has some differences from the common operating systems. We go through different memory management, getting the test to run on multiple threads and setting up the ANIS configuration inside PikeOS. Then, we will look at logs and how to read them. At the end of the chapter, we talk once again about the implemented scenarios and discuss the rest of the details about them.

6.1 Memory management in PikeOS

Usually, we want to use a heap for memory allocation. But this is not friendly to RTOSes. PikeOS native personality doesn't support heap allocation, and all memory has to be preallocated. The heap is not the most friendly structure for RTOSes. The allocated has a different latency every time. The heap gets fragmented, so there is no guarantee of enough space for the structure. Developers would also be sure that the heap is thread-safe. Without the heap, everything needs to be preallocated so the developers can be sure there is enough space for the data. And that's why we need to preallocate all the memory for our schedule in Slave during run time. No heap leaves us with a problem of how much memory the system needs, which is challenging due to the unpredictability of the fuzzer's randomness.

We would like to introduce a solution tied to the parametrization of the fuzzer. We took the fuzzer's configuration and updated the allocated memory's size. When we wrote the configuration, we defined the domain for all parameters. For example, we restrict the number of possible generated scenarios to 10, so we know that the program needs to store at most ten scenarios in its memory. Similarly, we can define how much memory we need for all the structures in code by estimating the worst-case scenario. This value can be computed if we take the biggest number that can be generated and multiply it by the size of the data structure for said parameter. We have created a set of parameters with their corresponding arrays in code that can be adjusted to fit the developer's needs. The whole list of parameters and their meaning is described in Section 5.8.

The schedule is not stored only in the Slave Controller but also in the ANIS client. We needed to allocate the same resources because they live in separate partitions that don't share memory. The definition of arrays with memory for the schedule are located in files */fuzzer/001/tc_fuzzer_001.slave* and *fuzzer/001/anisClient/anisClient.c*. Their sizes are configured in file */fuzzer/001/common.h*.

This style of pre-allocation of the memory can be found at all parts of the code base because we needed them for different reasons, e.g., storing the payload of UDP packets.

6.2 Parallelism

Our goal was to simulate multiple applications running on the network. Therefore, we had to use some level of concurrent execution of multiple threads, especially when our fuzzer created two scenarios that should run concurrently. Without multiple threads, the test could only run one scenario, and on top of that, such scenario could only send or receive data, never do both. That restricts how much state space we cover, which wouldn't be suitable for the test.

Role of threads

Let's first describe how the threads are used to run scenarios. When the time comes to run a scenario, ANIS Client creates a thread for it. This thread is responsible for correctly executing the scenario. But that's not everything. Earlier, we have mentioned scenarios running on multiple sockets. We also had to consider scenarios that want to simultaneously receive and send data over the network. The simultaneous execution created some requirements for our usage of threading. Every scenario creates more threads to accommodate its needs, depending on how many sockets it should listen on, and possibly also creates more sending threads.

Real-time thread scheduling

We have already mentioned hard deadlines for scheduled tasks. Every thread has its priority, which is used to schedule them on the CPU. On top of that, all threads live within a single partition. Partitions also have priorities between them, which cap the priorities of the threads that live inside them. The best option for us proved to set priorities to the same values. And not just between threads but also between partitions. We let the threads run with the same priorities, so no thread has an edge over others. To achieve better results, we tried to adjust the priorities of the threads. But it showed some problems. If the scenario threads had smaller priorities than the controller, it led to the controller running all the time, starving the scenario threads. The threads then ran at once at the end of the test. This behavior was not desired. When the tides were turned and the controller had lower priority, it blocked scenarios from running concurrently because the controller was waiting for the last executed thread to finish. So that's why we kept the priorities of the threads and the partitions equal.

Joining threads

When dealing with threads, we had one last problem to deal with. PikeOS native personality doesn't have the thread join implemented. The test needs that, for example, for waiting on the scenarios thread before cleaning up the ANIS client. Before implementing our joining mechanism, we tried to synchronize threads with sleep. The sleeping showed some promise, but it also proved to be unreliable.

Our implementation consists of an array of integers in the original thread. Every thread started from the origin can access one number in the array. Initially, all the numbers in the array are set to zero. When the thread is about to exit or done with its work, it sets the value in that array to one. The thread that

waits for the others to synchronize constantly oversees this array. The moment when all values are set to 1, the original thread can continue. This way, it is ensured that all working threads are done before the test continues to the next computation.

This implementation worked reasonably well, but one problem emerged with the blocking mode of sockets. The thread that wants to receive from such sockets stops on *receive_from* socket call and waits there infinitely. These threads will never finish. In this case, we have implemented method *wait_for_threads_to*, that not only checks if the threads are done but has also set an expiration timeout, after which it allows original thread to continue as all the threads are done executing. Other solution would be to kill the threads forcefully. The forceful killing sounds better than letting the threads be because it doesn't leave any never-stopping threads and our solution does, and that these threads can influence the next rounds of the test. That's true, but we want to restart the ANIS Client partition for each round. Restart reloads the whole running environment, including threads, so they can't influence other test rounds.

6.3 ANIS configuration

One of our tasks was to fuzz the configuration of the ANIS. The ANIS configuration was more of an interesting suggestion from SYSGO. We have decided to implement that because there are a lot of possible combinations of ANIS configuration that a developer can set. We needed to fuzz this as well. It is a big state space the developers would be missing when testing.

First, we needed to identify which ANIS configuration parameters would make sense to fuzz. The set is in Table 5.2. We needed to find a way to give this configuration to the ANIS. The application is shipped as an image with the initial ANIS configuration built in. The ANIS then takes the built-in configuration and sets itself up. The built-in configuration is the default configuration that ANIS uses. But before ANIS uses this configuration, it looks for a file in the shared memory. We haven't mentioned shared memory yet, but as the name suggests, it is a part of memory that is shared between all partitions. We can use this memory to transfer data between them, which is necessary here because ANIS lives in its partition, to which the Slave Controller needs to pass the configuration. If the file is found, ANIS takes the configuration from it rather than the default configuration. This file gives us a way to pass our fuzzed configuration to ANIS. After the file is created and filled with data, ANIS is then restarted by the Slave Controller. ANIS takes the new configuration file and starts working with parameters from said file.

6.4 Logs

The test wouldn't be beneficial if the developers could not understand what happened. We needed to create some readable and easily understandable log structure. This task is not trivial, mainly because the logs are used for diagnosing the round. Probably one of the most significant improvements was to reproduce the test rounds. This improvement was achieved by logging the seed used to fuzz the

data every round.

We had two options for how to track the progress of each test. We could have written all the messages down into a memory buffer and then transferred this buffer to some storage outside of the test as a file. But we also had an easier option to write logs to the console. TFW automatically picks them up and stores them into a file.

Now, we will describe the structure of logs produced by the test.

Before test

Before the first round is executed, the test loads configuration for the fuzzer. We are talking about all the fuzzer parameters and their distribution functions. This configuration is written down in the log as a list of parameters and their distributions. We can see an example in Figure 6.1

```
|tc_fuzzer_001|================ DISTRIBUTION FUNCTION for fuzzer_
   arp_table_size ================
|tc_fuzzer_001|Distribution parts 5
|tc_fuzzer_001|------------------------------------- PART
   -------------------------------------
|tc_fuzzer_001|percentage 5
|tc_fuzzer_001|type constant
|tc_fuzzer_001|value 10
|tc_fuzzer_001|------------------------------------- PART
   -------------------------------------
|tc_fuzzer_001|percentage 5
|tc_fuzzer_001|type constant
|tc_fuzzer_001|value 11
|tc_fuzzer_001|------------------------------------- PART
   -------------------------------------
|tc_fuzzer_001|percentage 10
|tc_fuzzer_001|type constant
|tc_fuzzer_001|value 20
|tc_fuzzer_001|------------------------------------- PART
   -------------------------------------
|tc_fuzzer_001|percentage 10
|tc_fuzzer_001|type constant
|tc_fuzzer_001|value 40
|tc_fuzzer_001|------------------------------------- PART
   -------------------------------------
|tc_fuzzer_001|percentage 70
|tc_fuzzer_001|type uniform
|tc_fuzzer_001|lower bound 21
|tc_fuzzer_001|upper bound 80
```

Figure 6.1: Example of log of fuzzer parameter size of ARP table

First, the parameter's name is written, followed by the defined parts of the distribution. Next, every part is written down with values like the percentage chance, the bounds for that part, and the type. This log serves as a chance for the developer to check if the fuzzer is configured correctly. From the log example in Figure 6.1, we can see that there is a 10% chance to generate value 10 or 11,

20% for 20 or 40 and 70% chance for the value to be uniformly generated from the range 21 to 80.

Every round has the fuzzed values of parameters and configuration written out. First, the configuration for the ANIS, followed by the schedule. Example ANIS configuration in Figure 6.2 is just a set of parameters and values. More information about parameters we have already described in Section 5.9. The configuration is very useful for second evaluating the test round. It can be used to discover false positives that the developers wouldn't want to waste their time with.

```
|tc_fuzzer_001|================================== ANIS CONFIG
   ==================================
|tc_fuzzer_001|ANIS config after fuzzing
|tc_fuzzer_001|ANIS number of sockets 141
|tc_fuzzer_001|ANIS number of buffers 1209
|tc_fuzzer_001|ANIS MTU 1500
|tc_fuzzer_001|ANIS peerchache 25
|tc_fuzzer_001|ANIS ARP table size 32
|tc_fuzzer_001|ANIS resolution timeout 325
|tc_fuzzer_001|ANIS queue length 10
```

Figure 6.2: Example of log of generated parameter for ANIS

Similarly, the schedule is logged in Figure 6.3. It starts with the total length of the test and the number of scenarios, followed by the list of scenarios and their parameters that can influence the test.

```
|tc_fuzzer_001|Total length of the test 9681 ms
|tc_fuzzer_001|SCHEDULE size 4
|tc_fuzzer_001|Scenarios in the schedule
|tc_fuzzer_001|================================== SCENARIO
   ==================================
|tc_fuzzer_001|ID 0
|tc_fuzzer_001|type Scheduled scenario
|tc_fuzzer_001|number of master ports 2
|tc_fuzzer_001|number of slave ports 2
|tc_fuzzer_001|FROM 1083ms
|tc_fuzzer_001|TO 5505ms
|tc_fuzzer_001|Number of requests 6
```

Figure 6.3: Example of log of the schedule with scenario

Test Execution

All scenarios are logging similar information. First, the test informs about the start of the scenario. For each thread the scenario runs, it logs the start of that thread. After the scenario finishes, it logs the end and summary of what happened. The summary differs for each type of scenario. The logging of the summary occurs on both sides, which means for the Master as well as for the Slave. Messages logged for each scenario can be recognized by its prefix, which uses the id of the scenario to distinguish them. See the Figure 6.4

```
|tc_fuzzer_001|ANIS CLIENT|SC ID 2|Receiving thread is stopping
```

Figure 6.4: Example of log during schedule execution

The first part tells us the identification for the whole test case, followed by on which side it is happening, the id of the scenario currently running, and finally, the actual log message.

Checking the results

After the execution, the test needs to log what happened with the ANIS during the health check. All the messages related to the Health Test have the prefix HEALTH TEST in place of the scenario ID during the execution logging. We show an example in Figure 6.5 Both Master and Slave log what packet they send or receive. On top of that, the Slave part prints out all errors that happen when opening sockets or providing other information. The special conditions, like the address resolution, is too low, are recognized and dealt with in the log. In such cases, the information about the result, PASS or FAIL, is followed by all the special conditions that could have influenced result of the health check.

```
|tc_fuzzer_001|============================== HEALTH TEST
   STARTING ==============================
|tc_fuzzer_001|MASTER|HEALTH TEST|Sending packet to slave try 1
|tc_fuzzer_001|MASTER|HEALTH TEST|Packet sent
|tc_fuzzer_001|MASTER|HEALTH TEST|Waiting for response
|tc_fuzzer_001|MASTER|HEALTH TEST|Timeout
|tc_fuzzer_001|MASTER|HEALTH TEST|Sending packet to slave try 2
|tc_fuzzer_001|MASTER|HEALTH TEST|Packet sent
|tc_fuzzer_001|MASTER|HEALTH TEST|Waiting for response
|tc_fuzzer_001|SLAVE CONTROLLER|starting partition
   anisHealthTestPart ...
|health test|HEALTH TEST|Starting
|health test|HEALTH TEST|Waiting for packet from master
|health test|HEALTH TEST|Received T
|health test|HEALTH TEST|Sending packet
|tc_fuzzer_001|MASTER|HEALTH TEST|Response received
|tc_fuzzer_001|MASTER|HEALTH TEST|Ends
| back to master try 1
|health test|HEALTH TEST|Ends
|tc_fuzzer_001|1|SURVIVED_SCHEDULE|S_PASS
```

Figure 6.5: Example of log during schedule execution

6.5 Fuzzer

The fuzzer has three usable entry points: load the configuration, set the current seed and fuzz the data. Before the fuzzer can be properly used, the configuration needs to be loaded. If the developer wants to have better control over the fuzzed data, the seed should be set. To correctly load the configuration, we needed to implement a parser for the configuration language.

Parser for fuzzer configuration

We also needed to implement a parser for configuration language described in Section 5.8. We aimed for a simple and robust parser, so if the developer makes a mistake, the code won't crash without providing the error for the developer. This parse might lead to some cases, when the configuration might not be parsed correctly, but it's up to the developer to check the logs and correct it to his liking.

Parser first splits the string into subparts by the semicolon ';'. Then, every part is checked for the leading letter, followed by the opening bracket. After that, the inner content is parsed. First split by normal colon ',' and parsed. All non-numeric symbols inside brackets are being ignored.

After everything is parsed, the fuzzer checks its configuration. If something is not in order, it informs the developer through logs and exits, leaving the developer to fix the mistake and re-run the test.

6.6 Scenarios

Now we look a little more under the hood of all test scenarios, how they are implemented and what we expect them to do.

Each scenario has an origin thread that is responsible for the execution of the scenario. Usually, the origin thread starts other threads that receive or send data to simulate traffic in the network. The scenario is considered done when the origin thread exits. For now, all scenarios need their behavior defined for both the Master and the Slave. The origin thread at the end waits for all the threads it has started and doesn't exit before that.

Scenario 1: *Scheduled*

The first scenario, called *Scheduled*, consists mainly of the list of UDP requests that the app is supposed to send. Each UDP request is specified to go either from the Master or the Slave. These requests are sent at a specific time. That might fail due to exhausted ANIS buffers or some other circumstances. The behavior is implemented in file *fuzzer/001/sc_scheduled_recv_master.h* for Master and in the file *fuzzer/001/sc_scheduled_recv_pike.h* for the Slave side.

Because the scenario has the same behavior on both sides, we can describe it just once. The origin thread first starts multiple receiving threads. Each thread listens on some port, waiting for input. These threads are there to monitor if ANIS is processing any packets. The number of listening threads differs depending on the fuzzed value for the number of sockets used for the scenario. The number of sockets can vary for the Master and the Slave. Receiving threads are running for some predefined period. The length of the scenario sets this period, but we have decided to extend it by a small padding of one second so the receiving thread has a chance to process more UDP packets that came a little late.

After the receiving threads are started, the origin thread starts another thread that is supposed to send data out. This thread goes through the list of UDP requests. The requests that belong to the other side are skipped, and the thread sends the rest out on time. The sending thread ends once it is done sending all the requests it was supposed to send.

A simple payload generator generates the packet payload. Upon receipt, the payload of all packets is checked to see if the data was received correctly. The packet's content is generated as a sequence of bytes, where each byte is dependent on its predecessor. We use multiplication and modulo operator combined with prime numbers to generate the same sequence again. This algorithm is used to check if the whole packet arrived correctly and no data were lost.

```
unsigned char gen_next_byte(unsigned char *seed) {
  int x = (int)*seed;

  /*Randomly chosen constants, their greatest common divisor is 1*/
  x = (x * 709) % 256;

  /*safe it as a seed for the next byte in sequence*/
  *seed = x;
  return (char)x;
}
```

Figure 6.6: Generating payload for the scheduled packets

Scenario 2: *Unthrottled*

The point of *Unthrottled* scenarios is to try to trigger the behavior of the ANIS under high pressure with many packets received or sent out. The unthrottled stream of UDP packets goes only one way, either from the Master or the Slave. Behavior on both sides is the same, so we call them the receiving and sending sides from now on. The origin thread starts some number of receiving or sending threads, depending on the direction parameter for the scenario. The behavior of receiving, or the sending, type is the same for the Master and the Slave. Once again, the number of running threads is determined by the number of sockets this scenario is supposed to use. The implementation can be again found in the files *fuzzer/001/sc_unthrottled_master.h* and *fuzzer/001/sc_unthrottled_pike.h* for the Master and the Slave side.

Unthrottled scenario is a little unique because it also fuzzes a parameter that determines the blocking mode of the socket. Blocking mode means that the thread stops its execution until the socket can perform its action. If the thread is trying to send something out, the thread is blocked until the socket is free to send out that packet. The receiving thread is blocked until a complete datagram can be read from the socket. Blocking gave us a little challenge with shutting down the receiving threads, which was already mentioned in Section 6.2.

Receiving threads just start and wait for some data. If non-blocking mode is set, it spins infinitely and waits for some packets. When the time comes, the thread shuts down. The problem comes with blocking mode. If there is no more incoming traffic and the thread stops on the call of the receive function, this thread will never exit. Because on the Master side, the socket is just a file, we can use file descriptors and Posix method *select* allows the code to check if the packet can be read from that file, in our case, socket. This method can be timed out, so our threads on the Master have a deadline for their exit. Unfortunately,

we can't use the same mechanism on the Slave side, because we don't have it, so we had to settle for the mechanism mentioned in Section 6.2.

Sending threads are used to send a stream of packets. The data are sent in smaller periods, alternating with periods when nothing happens. So, for example, the sending thread sends packets one by one for a hundred milliseconds, then waits for another 100 milliseconds and doesn't send anything. This loop is repeated until the scenario is supposed to end. These empty periods are supposed to give ANIS some time to handle incoming traffic. The fuzzed parameter determines the length of the periods. Sending threads end when they go through enough periods of sending or when the time for the scenario expires.

Scenario 3: *Unfinished packets*

Before we dive into what we have implemented, we describe more details about ANIS. To correctly assemble UDP datagrams from their fragments, IP packets, ANIS needs some inner memory to store their fragments. With this scenario, we aim to expose it to incomplete datagrams. These datagrams can't be assembled, so their fragments fill up the ANIS memory, and we cover more state space around it.

The scenario *Unfinished packets*, or shortly *Unfinished*, is meant to test the stability of the stack when the fragmented datagrams are not received completely. Our test simulates this behavior by fragmenting a UDP packet and not sending one of the IP fragments. We want this test to check ANIS for receiving this kind of data, we don't need to test the Master for receiving unfinished datagrams. That's why we only implemented sending behavior for the Master side and receiving behavior for the Slave side. Master serves as a sender and Slave as a receiver for those fragments.

Slave only waits for fragments to come to the ANIS so they can be read. But if the test runs correctly, the receiving thread should never read any packet. This thread is only used for keeping the socket open, so ANIS won't throw away all the incoming traffic for the test. For the *Unfinished* scenario to be effective, we must ensure enough fragments fill up the ANIS memory. The sending, the Master, side functions similarly to the *Unthrottled* scenarios when it sends a long stream of packets. This stream is influenced by parameters for this scenario, described in Table 5.5.

The Master side is more complicated than the Slave side because we have to create our headers for network packets and fragments. Parameters from the fuzzer define how many unfinished packets we should send and how often. The sending thread sends data either until the maximal time is over or enough packets have been sent. Let's look at the subtasks the Master needs to fulfill to send a fragmented datagram.

Raw sockets

Before we head into the subtasks, we need to realize that we are working on a lower layer than with UDP packets. Linux is usually trying to help and fill packet header data for us. We must use the raw sockets if we need to go around this. Raw socket allows us to specify different options that restrict what the Linux does with the packet headers. Specifically, we are looking at option $IP\_HDRINCL$,

which requires us to fill the headers on the IP layer and above. The operating system will still fill the lower layer header.

Datagram creation

First, the sending thread needs to create a UDP packet. SYSGO has provided us with a library that does this for us. We just had to first port this library to the UNIX environment because it was initially written for the PikeOS and then used it to create and fragment the datagrams. On the UDP layer, a port of the destination and origin needs to be specified. The thread has information about ports and IP addresses, and that is enough information to create a UDP packet by the provided library. We must ensure the UDP packet is larger than MTU for fragmentation. This requirement can be achieved by setting the size of the packet to the maximal UDP packet size, which is 65536 bytes. This packet has to be fragmented every single time. In Figure 6.7, we can see the call to the library that creates the needed datagram.

```
create_udp(udp, &len, dst.sin_addr, src.sin_addr, OPT_UDP_
    DESTINATION_PORT,
              ntohs(dst.sin_port), OPT_UDP_SOURCE_PORT, ntohs(
                  src.sin_port),
              OPT_UDP_PAYLOAD_LENGTH, payload_len, OPT_UDP_
                  SETLENGTH, -1,
              OPT_UDP_CHECKSUM_MODIFIER, 0x0000, OPT_UDP_PADDING
                  , 0,
              OPT_IPV4_VERSION, 4, OPT_IPV4_TOS, 0, OPT_IPV4_DF,
                   0x10000,
              OPT_IPV4_FLAG_RESERVED, 0, OPT_IPV4_PROTOCOL, 17,
              OPT_IPV4_HEADERLEN, 5, OPT_END);
```

Figure 6.7: Call to the library function to create a UDP datagram

Fragmentation

The sending thread must fragment the generated packet so its fragments can be sent on the lower layer. This time, the code adds the IP addresses of the source and destination to the headers of the fragments. Once again, we can use the provided library to fragment such packets. The fragment method uses callback function in Figure 6.8 to provide us with single fragments.

Now, the function has the fragments, but they can't just be sent out. That way, the thread would send the whole UDP packet, which is not desired because it wouldn't fill any of the ANIS memory. The callback function stores the fragments in a buffer. Because the callback function doesn't accept any outside parameters, we needed to use a global buffer and copy the data to a local buffer for the running thread. We copy it to the local buffer so the thread can proceed with sending the fragments, and some other thread can generate the next fragmented packet to the same buffer. This way, we can have more threads running concurrently, and they do not wait for each other so long. After this, we can do whatever we want with the fragments. It is enough to randomly select one fragment that won't be sent for our case. The ANIS should be able to deal with the incorrect order of

```
typedef struct fragment {
  unsigned char *buff;
  uint len;
  int used;
} fragment_t;

#define MAX_FRAGS (MAXIMUM_UDP_DATAGRAM_SIZE / 1500) + 1
fragment_t packet[MAX_FRAGS];
uint store_frag_id = 0;

void collect_fragment(unsigned char *buff, uint len) {
  fragment_t *frag = &packet[store_frag_id];
  frag->buff = malloc(len * sizeof(*buff));

  memcpy(frag->buff, buff + (14 * sizeof(*buff)), (len - 14) *
      sizeof(*buff));
  frag->len = len - 14;
  frag->used = 0;

  store_frag_id++;
}
```

Figure 6.8: Callback function collecting fragments

fragments, so it doesn't matter in which order the thread sends them out nor which fragment is left out. This behavior can trigger situations like overflowing buffers with incomplete packets that are not finished, wrong, and incomplete order of fragments that the ANIS might try to assemble wrongly.

6.7 Extending the long run test

We talked about more scenarios, with which we could improve our test, e.g., generating IGMP or ICMP traffic. That's why we put some focus on the extensibility of the test. The process is not easy, but it can be broken down into multiple steps, leading to successful test extensions. We assume that the developer has already decided what the scenario does and is sure about what parameters the scenario needs.

1. **Define data structures and fuzzer parameters**
   We take our desired parameters for the fuzzer and put them into the *config/anis.conf* file. For each parameter, we describe its distribution function using defined grammar from Section 5.8.

   Then, the developer has to define the data structure used for the scenario. All structures holding scenario data are in file *fuzzer/001/scenario.h*. This structure represents all fuzzed parameters for the scenario. The scenarios are associated with scenario ID, which the developer must extend for a new scenario.
   This file also contains some functions that help translate scenario data to human-readable format, which we recommend extending.

2. **Extend parameter loading**
   The fuzzer needs to be able to load the configuration properly. The developer needs to change *fuzzer/001/fuzzer_configuration_helper.h*. Specifically, the enum holds all fuzzer parameters and the array with the parameters parsed out of *anis.conf*. The fuzzer is now able to load the configuration.

3. **Extend fuzzer to generate new scenario**
   We will stay in the file $fuzzer.h$ from the last point. The loading of the new parameters is already set. Now, it is time to use them in fuzzing. To use them, extend the method $fuzz\_scenario\_specific$. The method contains a switch for the scenario IDs that must be extended. Then, the developer has to write a function that uses the random generation to fill the new scenario with its parameters. For generation, it is sufficient to call the provided function $random\_from\_dist$, which is defined in the file *fuzzer_random*.

4. **Pass parameters between components**
   The fuzzer generates the schedule only on the Master side. The data has to be passed to the Slave Controller and from there to the Anis Client.

   To pass data to the Slave Controller, the developer extends the *fuzzer/001/serialize_m_c.h* and *fuzzer/001/deserialize_m_c.h*. The used mechanism is the TFW serial line. Let's start with passing data to the Slave Controller. The Master needs to send all parameters one by one on the line; on the other end, the Slave needs to receive it. First, the file *fuzzer/001/serialize_m_c.h* contains function *send_schedule* with a switch that chooses serialization by the scenario ID. The developer extends this switch and writes a function that sends all the parameters in the new scenario structure. On the other side, in the file *fuzzer/001/deserialize_m_c.h*,

there is function *recv_schedule* with similar switch for which we will have to implement the receiving function.

The process of passing the data from the Slave Controller to the ANIS Client is the same. The developer extends the files *fuzzer/001/serialize_c_c.h* and *fuzzer/001/deserialize_c_c.h*. The only difference is that the Slave uses queuing ports API to communicate between the partitions.

5. **Scenario behavior**
   The next step is to code the behavior of the scenarios. The extension should be done in a new, separate file. Each scenario has its thread. The developer should create an entry method for the scenario. The ANIS and Network Clients call this method. The developer can then implement everything behind this method. While making the files for Master and Slave, the names for the file should follow the style of already implemented scenarios, *sc_{name}_pike.h* and *sc_{name}_master.h*.

6. **Extend controlling threads**
   The last step is to extend the controllers so they start running the scenarios through their entry functions.

   First, we extend the Network Controller of the Master. Before each round, a precomputation counts the number of each scenario type. The developer needs to extend the *precompute_round*() function in the file *fuzzer/001/tc_fuzzer_001.master.c* to count the new scenario. Then, in the file *network_traffic_controller.c*, there is a method *network_traffic_controller*. The developer must extend the switch to recognize a new scenario ID and run its entry method. The switch uses arrays to hold data of each scenario type in the structure *scen_thd_data_t*. That is all to extend the controller on the Master side.
   The developer needs to reproduce the same steps for the Slave side. This time, everything is in the file *fuzzer/001/clientApp/clientApp.c*. It also has a similar preprocessing as the Master. This time, it is in the function *preprocess_schedule*. The rest is in the function $execute_scenarios$. The developer needs to create an array holding the scenario data and extend the switch to execute the new scenarios entry function in a separate thread.

At this point, everything is correctly set for a new scenario. The test is extended with a new scenario and ready to cover even more state space of the ANIS.

The reader has probably already noticed that we have implemented everything in the header files, and we expect the developers to continue in this way. We found the build process of the test cases quite challenging to alter, and we have decided not to spend more time on it. Implementing everything in the header files is still functional way without many downfalls.

7. Use of the test

In this chapter, we look into how this test should be set up and how it can be used for report analysis of ANIS. We also discuss some properties of our test that could be improved.

7.1 Set up fuzzer in TFW

Before using the test, the developers will have to configure our fuzzer. In the attachment, we provide an example configuration that can be used as a reasonable starting point, but the developers can update it to their preferences. They will have to follow the syntax rules of our configuration. An important step is to set all the parameters of the fuzzer. If even one parameter is missing, the test will end with a compile-time error, and the parameter will have to be set for further use of the test.

Setting network interface

We have decided to set up a new network interface for this test. We chose this option because we wanted to isolate our test from any other traffic that could be going on our computer. This way, nothing interferes with our test and gives us an environment in which we can recreate the schedule for debugging purposes.

We provide a script $tap0\_set.bash$, that creates a new virtual network interface called tap0. If an interface with this name exists, changing the interface's name in the script is preferred. We want to avoid combining our traffic with traffic that can already exist in that interface. In the end, the script sets the environment variable $TFW\_QEMU\_OPTIONS$ to the value "$-nettap, ifname = tap0$", which uses the created tap0 interface. If the developers wish the test to use a different interface, only these variables need to be changed to the name of the desired interface.

7.2 Investigating findings of the fuzzer

When the test fails, developers want to examine where exactly the error appeared and why it happened. Our test allows us to set the specific seed for our fuzzer. This seed helps the developers rerun the same configuration quite easily just by specifying the correct seed.

To rerun the test round, the developer needs first to find a corresponding seed, that is written out in the log before every round of the test.

Then, before the main loop in file $tc\_fuzzer\_001.master.c$ needs to update variable *seed* to the desired value. Then, he can take that seed and set the *seed* variable in the Master Controller entry function to that value. After rerunning, the test value in *seed* will be used and if the fuzzer has the same settings, it will generate identical output.

It can be done after the schedule is generated if there is a need for manually modifying the schedule. In the same file with the Master Controller uncomment

line *TODO* with *modify_schedule* method. Implementation of this method is located in the same file. The developer must know what he wants to happen with the schedule and directly update the generated schedule.

The last debugging option is to uncomment the definition of macro VERBOSE in the file *common.h*. This macro enables more detailed output logs.

7.3 Effectiveness

We have a working prototype of the fuzzer that we know how to configure and use for debugging. At last, we talk about some downsides that could be improved in the future. Let's talk about the effectiveness of our test. We aim to test as many schedules as possible. Fuzzers usually try to achieve effectiveness by maximizing the throughput of inputs. The design of our test round has two major choke points that limit how many schedules it can run within some time. These choke points are the execution of the schedule and the Health Test.

Time spent testing

We first consider the length of the schedule. The time for which the schedule can generate scenarios, is managed by fuzzer parameter *fuzzer_schedule_test_length*. With this parameter, the developers can restrict the generated value to their liking. The value developers select depends on whether they want to see how AnisFP works under longer periods or wish to discover a short amount of network traffic that can break the stack. There is probably no correct answer on how to set this parameter and how much time should be spent on running the test. So the options are long-run mode when the ANIS runs for an extended amount of time, but the structure of the test rounds won't change that much, or very short rounds of testing, which generates a lot of different scenarios that try to break ANIS in a short period. Both these options test different state space, so the choice is really about what space the developers want to test.

The other choke point is the health check, with which we had issues. At this point, it is set to send multiple packets in the total period of 15 seconds. We consider only the worst-case scenario for our check. It can last shorter if the Health check returns the packet on the first try, but it doesn't always happen. If we consider schedules that run shorter, there is a significant overhead with Health Test, and spending twice as much time on checking the stack, then the testing is inefficient. With short schedules under 10 seconds, we always have a 15-second lasting health check. We can fit multiple scenarios into the 10-second schedule to cover some space. If we run the test with the 10-second schedules, we should be able to go through $3600/(10 + 15) = 144$ rounds per hour. If we consider some overhead, e.g., for the fuzzing or passing data, we were left with around 120 tested scenarios per hour. The Health Test could be more effective, but for SYSGO purposes, it showed to be sufficient.

We tried to make the health check run faster but at the point of writing this with no success. With shortening the time waiting and sending controlling packets to Slave, more false positives for bugs in AnisFP were found. So far, we have not fixed this problem, which is open for future improvement.

Repeating schedules

In the end, we wanted to measure the efficiency from a different angle. We wanted to discover how often schedules are repeatedly generated from the same or a different seed.

To generate schedules, we use a mechanism based on a seed. This seed determines the outcome of the generator. The generated content will be the same if the seeds are the same. When choosing our seed, we opted for the most straightforward choice, which is the machine's current time. Because the time is linear, we don't need to worry about the seed repeating.

We needed to consider one more thing: when the seed generates the same schedule. How much the generated schedule and ANIS configuration depend on the configuration for the fuzzer? If we set all the parameters to constant values, the schedules will be the same for each round. The same principle applies when we set the parameters as open as possible. The chances of generating the exact schedules drop. We wondered how often that happens for some minimalist configuration like we provide in the file *anis.conf*. Therefore, we provide a simple Python script *duplicates.py* that counts the exact schedules in one test run. The script uses the log to get the schedules and compare them. We have checked over 1500 different schedules generated by the provided configuration. That is around twelve hours of testing, and we still have not encountered duplicate schedules. Even with such a simple random generator, it is unlikely to generate the exact schedules, so we don't consider repeating schedules to impact the fuzzer's efficiency.

8. Conclusion

With this project, we have successfully extended a big test suite for the ANIS network stack, a safety-certifiable network stack for SYSGO's real-time operating system, PikeOS. Our test is fully integrated within SYSGO's test suite for ANIS. With the support of QEMU, the test can be run on any Linux workstation.

We have analyzed options of fuzzing techniques to generate and test different network stacks. We have used our findings to imitate real world traffic scenarios that could potentially trigger failures in ANIS network stack. Next to the techniques of randomly generating data for the test, we provided a way to configure how these parameters will be generated. To demonstrate the functionality of the test, we have implemented a couple of scenarios exploring some aspects of UDP traffic that might be received or sent by the network stack. We also described a detailed way to extend the current test for new scenarios, so even more areas of the network traffic can be explored. Next to fuzzing internet traffic, we also provide support for fuzzing the configuration for ANIS. Our goal was to test the stability of the stack, which we checked with a simple test to see if it survived. This check needs to recognize corner cases for ANIS configuration and adjust to them. The list of the corner cases might have to be extended in the future if discoveries about edge case configurations for the ANIS are made. We are talking about configurations like small ARP resolutions timeout we have mentioned before.

In the last chapter, we have seen some efficiency difficulties that are slowing the execution of the test. The developers need more time to process and validate the generated schedules. It would be welcomed if the health check runs faster but keeps its resilience towards false positive findings.

So far, the test has not discovered any bugs inside ANIS. There are multiple reasons why the fuzzer has yet to find bugs. The test currently executes a limited number of scenarios, and ANIS has already been verified by thousands of work hours. Therefore, ANIS should be bug-free.

Bibliography

[Andronidis and Cadar, 2022] Andronidis, A. and Cadar, C. (2022). Snapfuzz: High-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 340–351, New York, NY, USA. Association for Computing Machinery.

[Böhme et al., 2017] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. (2017). Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA. Association for Computing Machinery.

[Heuse et al.,] Heuse, M., Eißfeldt, H., Fioraldi, A., and Maier, D. Afl++ overview. Retrieved 2023-11-12.

[Miller et al., 1990] Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44.

[Pham et al., 2020] Pham, V.-T., Böhme, M., and Roychoudhury, A. (2020). Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465.

[Sutton et al., 2007] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[Zalewski,] Zalewski, M. American fuzzy loop. Retrieved 2023-11-12.

[Zeller et al., 2023] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. (2023). Fuzzing: Breaking things with random inputs. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. Retrieved 2023-01-07 14:00:06+01:00.

List of Figures

List of Tables

A. Attachments

A.1 Content of the file archive

The attachment contains source files for the fuzzer and configuration files. We are not able to run the fuzzer without the PikeOS and TFW supporting libraries which are not included in the attachments. The test is implemented only as a set of header files, because of the complicated nature of the TFW.

- The *duplicates* folder contains script *duplicates.py* for counting duplicate schedules as mentioned in Section 7.3
- The *config_files* folder contains parts of configuration files, that were writen or altered by us.
  - The file *anis.conf* contains configuration of the fuzzer
  - The file *int.conf* that sets partitions inside PikeOS
  - The file *snip.xml* alterst the partition configuration during build
  - The file *tfwtags.xml* updates TFW configuration
- The *fuzzer/001* folder contains the source code for the fuzzer
  - The file *anis_utils.h* implements setting up ANIS configuration
  - The file *common.h* defines common structures and header for the Master and the Slave.
  - The file *data_generator.h* implements UDP payload generator.
  - The file *deserialize_c_c.h* implements receiving communication between Slave partitions.
  - The file *deserialize_m_c.h* implements receiving communication between the Master and the Slave.
  - The file *fuzzer.h* implements the fuzzer.
  - The file *fuzzer_configuration_helper.h* implements helper structs for the fuzzer.
  - The file *fuzzer_parser.h* implements parsing of the grammar
  - The file *fuzzer_random.h* implements generating values from the probability distributions.
  - The file *health_check_master.h* implements the Health test on the Master side.
  - The file *info_data.h* contains logging or debugging messages.
  - The file *logger.h* implements the logger for the data.
  - The file *network_client_master.h* implements execution of the scenarios on the Master side.
  - The file *networking_pike.h* implements networking utils for the Slave.

- The file *sc_scheduled_recv_master.h* implements the scheduled scenario for the Master
- The file *sc_scheduled_recv_pike.h* implements the scheduled scenario for the Slave
- The file *sc_unfinished_master.h* implements the unfinished scenario for the Master.
- The file *sc_unfinished_pike.h* implements the unfinished scenario for the Slave.
- The file *sc_unthrottled_master.h* implements the unthrottled scenario for the Master.
- The file *sc_unthrottled_pike.h* implements the unthrottled scenario for the Slave.
- The file *scenario.h* defines structures for the scenarios.
- The file *serialize_c_c.h* implements sending communication between Slave partitions.
- The file *serialize_m_c.h* implements receiving communication between the Master and the Slave.
- The file *sockets_master.h* implements helper function for manipulating with sockets on the Master.
- The file *tc_fuzzer_001.master.c* is an entry point for the Master application.
- The file *tc_fuzzer_001.slave.c* is an entry point for the Slave application and implements Slave Controller.
- The file *threads_master.h* implements the utility for the threads in the Master.
- The file *threads_pike.h* implements the utility for the threads in the Slave.
- The file *time_utils.h* implements manipulation with time.
- The file *timing_qport_protocol.h* implements protocol for Slave controller to signal that it is time to execute next scenaro to ANIS Client.
- The file *timing_tfw_protocol.h* implements protocol for Slave controller to signal that it is time to execute next scenaro to Master.
- The folder *clientApp* contains the file *clientApp.c* implements ANIS Client.
- The folder *healtTest* contains the file *healthTest.c* implements Health check on the Slave side.