Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bringing fuzzing capabilities to the Genode Framework

Master Thesis

S. Meier

April 15, 2025

ETH-Advisors: Prof. Dr. Srdjan Čapkun, Dr. Moritz Schneider
Gapfruit-Advisors: Stefan Thöni, Sid Hussmann

Department of Computer Science, ETH Zürich

Abstract

The Genode OS framework represents a novel operating system architecture that has been developed to address the challenges posed by complexity. It is an open-source tool kit for building highly secure component-based operating systems and its functionality extends across a wide range of devices, from those intended for use in embedded systems to those designed for dynamic general-purpose computing. Despite Genode's self-characterisation as a security-oriented operating system, it is notable that there is an absence of support for fuzzing, a process which has proven its worth in discovering real-world software vulnerabilities. In this thesis we present the process of porting AFL++, a state-of-the-art fuzzer, to Genode, with the objective to address this lack of support for fuzzing. The experimental findings demonstrate that, despite the introduction of additional overhead by Genode, the ported fuzzer runs at 93% of the fuzzing speed compared to running AFL++ on a general-purpose operating system. The fuzzer was then employed to test a security-critical Genode component.

Acknowledgments

The completion of this thesis would not have been possible without the support and guidance of numerous individuals who have contributed in various ways. I would like to express my gratitude to each of them.

First and foremost, I would like to thank my supervisor, Dr. Moritz Schneider, for their invaluable guidance, and expertise throughout this project. His insights and encouragement have been instrumental in shaping the quality of this work. I would also like to thank Prof. Dr Srdjan Čapkun for making this thesis possible.

Special thanks go to my industry advisors, Sid Hussmann (Gapfruit) and Stefan Thöni (Gapfruit), who have shared their knowledge, experience and showed a lot of patience. I also want to highlight the help received from Pirmin Duss (Gapfruit), by quickly jumping into a call and debug an issue. I would also like to thank to my friends Roger Steinmann and Julia Meier for proofreading this thesis.

Lastly, I am deeply grateful to my friends and family for their support and encouragement throughout this journey.

Contents

Chapter 1

Introduction

In the rapidly evolving landscape of cybersecurity, ensuring the reliability and security of software systems has become paramount. As software applications become increasingly complex and interconnected, the potential for vulnerabilities and bugs that can be exploited by malicious actors also grows. One of the most effective techniques for identifying these vulnerabilities is fuzzing, a dynamic analysis method that involves inputting invalid, unexpected, or random data into a computer program to uncover bugs and other weaknesses [1].

Fuzzing has emerged as a critical component in the software development lifecycle, particularly in the context of security testing. By systematically exploring a wide range of input scenarios, fuzzing can uncover flaws that might otherwise go undetected through traditional testing methods.

The significance of fuzzing is underscored by its widespread adoption in both industry [2, 3, 4, 5] and academia [6, 7, 8]. Major technology companies and open-source projects alike have integrated fuzzing into their development processes, recognizing its value in identifying and addressing vulnerabilities.

Especially in safety-critical systems, such as industrial control systems or critical network infrastructure, undetected vulnerabilities can potentially cause a lot of harm. It is a common practice for such systems to be equipped with an operating system that provide the necessary functionality. General-purpose operating systems, such as Linux or Windows, are a popular choice as they can provide a multitude of functionality, drivers and system services to their applications.

However, this approach inevitably results in a substantial and complex system, which is unavoidable due to the dynamic nature of workloads and the high functional demands placed upon them. This increased complexity invariably leads to a greater probability of vulnerabilities arising [9, 10].

Therefore this complexity needs to be addressed somehow. One way to organize it is by applying a strict organizational structure to all software components, as proposed by 'Genode, the operating system framework' [11]. Each component is explicitly defined in a tree structure, receives a share of the available hardware resources and, more importantly, is compartmentalised [12]. With this approach vulnerabilities can still occur, but with a reduced impact.

This thesis describes how we added fuzzing capabilities to the Genode framework. We provide an additional security tool by porting a state-of-the-art fuzzer AFL++ to Genode. A distinguishing feature of our fuzzer is its ability to test the communication interfaces of Genode's components and its core functionalities, which are crucial to every Genode system. Furthermore, by porting an existing fuzzer, we can utilise years of research that have been incorporated into these fuzzing tools.

By combining the vulnerability detection capability of fuzzers with the organizational structure of Genode, a novel method of addressing the complexity issue in security-critical systems will be presented. The ported fuzzer is based on AFL++ and achieves an execution speed of around 93% of that of AFL++. This disparity can be attributed to the architectural design of Genode, which prioritises security over performance.

The structure of the thesis is as follows: Initially, a comprehensive overview of the background of Genode, fuzzing and its related topics is provided. Thereafter, the problem statement is presented. The chapter after that demonstrates the concrete implementation of the ported fuzzer through a top-down approach. The subsequent chapters will then evaluate the fuzzer, review the related work, followed by a concise discussion and conclusion.

Chapter 2

Background

2.1 The Genode Framework

Genode [11] is an open-source operating system framework designed for high security and robustness. It enables the construction of highly secure systems by building on a microkernel architecture, which separates applications and functionality into distinct, isolated domains known as Protection domain (PD). These protection domains provide compartmentalization and operate independently, ensuring that even if one fails or is compromised, the impact on the overall system is minimized. Genode is often used for embedded systems, IoT devices, and security-critical environments due to its flexibility and fine-grained control over system resources.

At its core, Genode runs on top of a microkernel, which is responsible for essential tasks like memory management, thread scheduling, and Inter Process Communication (IPC). Unlike traditional monolithic operating systems, Genode organizes applications and system services into small, modular components that communicate with each other through Remote Procedure Call (RPC)s. This architecture allows for fine-grained control over permissions, memory, and resources, as each component is confined within its PD.

2.1.1 Components

Genode's architecture is organized around modular components, each serving a distinct role in the system and running within its own PD. This modularity allows the system to be highly customizable, enabling developers to include only the components required for a specific application. And as mentioned, each component communicates with others through the microkernel's RPC mechanisms, governed by capabilities to ensure strict access control.

Components are organized in a hierarchical and recursive system structure, as illustrated in Figure 2.1. The first user-level component is called `core` and represents the root of the component tree. It is a central part of the Genode architecture, providing essential services such as memory management, capability distribution, and system resource allocation [12]. Further, it directly communicates with the kernel and manages the creation and organization of other components while enforcing system-wide policies.

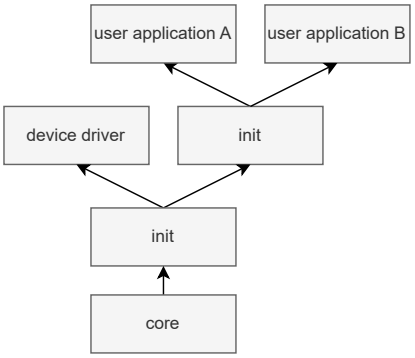

Figure 2.1: Example of a Genode component tree. It highlights the hierarchical and recursive system structure: the arrows depict the parent-child relationship and `init` can start another `init` component.

Next an `init` component is started. Among other functions, this component creates subsystems according to a user defined configuration called a "run-script". This script contains all information necessary for it to run, such as the components used, how many resources each component receives and how or when the script finishes. The tree structure exhibits an inherent parent-child relationship, whereby each component is initiated by its parent as a child. Exceptions to this rule are represented by the root of the tree.

This parent-child relationship allows components to manage resources and delegate specific capabilities to their children, providing a flexible way to construct and manage subsystems with isolated privileges. A child can then announce its service to the parent, allowing other components to detect its service.

To go beyond the `core`'s functionality, device drivers in Genode are also modular components that interact with hardware through strictly controlled interfaces. System services, such as networking, file storage, and graphics, are likewise encapsulated in separate components. Thanks to the protection domain, failures or security vulnerabilities in one driver or service component do not affect others.

2.1.2 Protection domains

Protection domains in Genode are isolated environments for applications and services. Each domain has its own set of resources and permissions, and interactions between domains are carefully controlled. This isolation mechanism is fundamental for security, as it prevents direct access to resources or data in other domains. If a domain needs to communicate or access resources outside its boundary, it must do so through defined, secure interfaces, managed by the employed microkernel.

The primary means of communication between protection domains in Genode are RPCs, which enable a domain to request services or resources from another domain in a secure, structured manner. When a component (the client) wants to use a service provided by another component (the server), it sends an

RPCs request. The microkernel intercepts and mediates this communication, providing security checks and enforcing access control policies to ensure that only authorized components can interact.

2.1.3 Sessions

As previously stated, a child announces its service to the parent. Other components can then access this announced service by setting up a session through the use of capabilities. Services that one might expect from a POSIX system, such as logging information to the console or accessing a file-system, are only available through the use of theses sessions, because there is no global namespace.

These sessions define clear interfaces through which a service can be accessed. Each component can announce its own custom session type, that defines how their service is accessed. However, it is preferable for only a small number of custom sessions to be in place, in order to maximise the composability of components. There are many common session interfaces that are used throughout Genode, which are introduced in the foundation book [11].

2.2 Code Coverage

Code coverage is a metric used in software testing to determine the extent to which the source code of a program has been executed and tested. High code coverage is often associated with thorough and effective testing. The reasoning being, that a software bug is only detected if the flawed code is executed. However, it is important to note that achieving high coverage does not guarantee the absence of bugs.

Different kind of coverage metrics exist with different tradeoffs, such as statement coverage, edge coverage and path coverage [13].

2.2.1 Statement Coverage

Statement coverage verification is conducted for each instruction in the code to ascertain whether it has been executed. The implementation of the tracking involves the use of a simple bit-array, called trace bits, in which a 1 indicates the execution of a statement and 0 indicates the absence of execution. In order to reduce the size of the trace bit array, the basic block[1] can be tracked instead of each statement on its own.

However, this approach does not guarantee the capture of all edges[2]. For instance, consider the simple function `foo()` with its corresponding Control Flow Graph (CFG) depicted in Figure 2.2. If all basic blocks A, B and C are covered, we know that the edge A $\rightarrow$ B, and B $\rightarrow$ C are covered, but it is not clear whether the edge from A $\rightarrow$ C was executed. It is for this reason that more precise coverage information is typically desired.

[1] Basic blocks are straight-line code sequences with a single entry point and a single exit point, containing no branches except at the end.

[2] Here edges refer to the edges from a basic block to another in the CFG.

```
1 void foo(char *input) {
2
3   if (input)
4     *input = 'a';
5
6   exit(0);
7 }
```

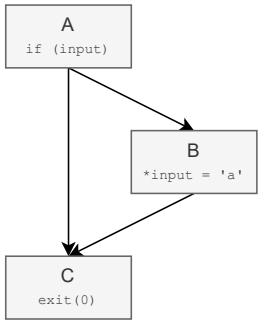

Figure 2.2: On the left: Example `foo()` function. On the right: CFG of the `foo()` function. Each box is a basic block and the arrows signifying the edges.

2.2.2 Edge Coverage

Edge coverage, as the name suggests, tracks the edges instead of the statements. This approach effectively addresses the limitations of statement coverage previously highlighted. As with statement coverage, trace bits are used to mark the basic block and the edge to this specific block. Consequently, full edge coverage implies full statement coverage. While this metric is more comprehensive, there still exist situations where full edge coverage has been achieved, but not every possible execution path has been taken. This situation is exemplified in Figure 2.3. If the function `bar()` is run once with `a = 2, b = 3` and once with `a = 3, b = 2`, we achieve full edge coverage, but we miss the out-of-bound access when calling the function with `a = 3, b = 3`.

```
1  char array[5] = { ... };
2
3  char bar(int a, int b) {
4      int i = 2;
5      if (a < 3) i -= 1;
6      else       i += 2;
7      if (b < 3) i -= 2;
8      else       i += 1;
9      return array[i];
10 }
```

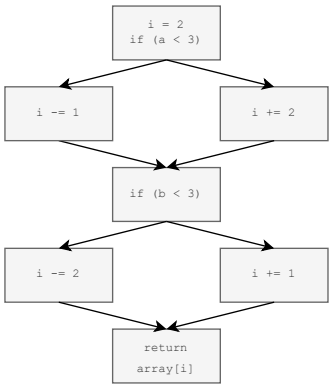

Figure 2.3: On the left: Vulnerable `bar()` function that contains a potential out-of-bound access. Inspired by [13]. On the right: CFG of the `bar()` function.

2.2.3 Path Coverage

Path coverage takes the tracking one step further and tracks every possible path to address the short comings of edge coverage. This, however, poses another problem, as each branch doubles the number of evaluated paths. This quickly becomes expensive and is known as path explosion [14]. This problem is exacerbated by loops where the number of iterations is not known. The implementation also becomes more complex: Given current memory constraints, programs with 40 branches or more cannot be mapped into a flat array in memory, so enumerating all paths becomes impossible. One option is to apply runtime monitoring of the execution and the paths [13].

2.3 Fuzzing

OWASP defines fuzzing in the following way:

> "Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion." (OWASP[15]; 2024)

A fuzzer is then defined as the application or a system that applies fuzzing on a system. This system is generalized as the so called System under Test (SUT) and refers to an application or subsystem that we want to test. It can be as simple as a ELF-binary or a complex multi-threaded server infrastructure. The process of using a fuzzer on a SUT is then known as a fuzzing campaign.

Figure 2.4 illustrates how a fuzzer typically operates. Here, the fuzzer generates a random input and then executes the SUT with this input data. Subsequently, the fuzzer receives execution feedback, which may include information on whether the SUT crashed or experienced a timeout. This feedback is then used to generate new input data.

In the event that solely random data is generated, the efficacy of the fuzzer is equivalent to that of a brute force approach. It is therefore crucial that relevant input data is generated. This is typically accomplished by incorporating coverage data during the generation of new input data. The coverage information is used by the fuzzer to identify inputs of interest, given that these inputs have higher coverage. The fuzzer then uses these inputs as a basis to generate further inputs. This enhances the fuzzer's efficiency, as it leads to the identification of more vulnerabilities with less fuzzing iterations.

An alternative approach would be to utilise dictionaries, which explicitly define the structure of the generated input data. To illustrate this, one may consider a dictionary designed for a web server, which would generate valid HTTP requests, with the HTTP headers and files undergoing constant modification.

Once relevant input is generated, the main goal becomes to increase the performance, that is to say, the number of executions per second. It is important that each execution is as fast as possible, thereby enabling the identification of more potential vulnerabilities within the specified time frame. Thus, performance is a central concern when using fuzzers.

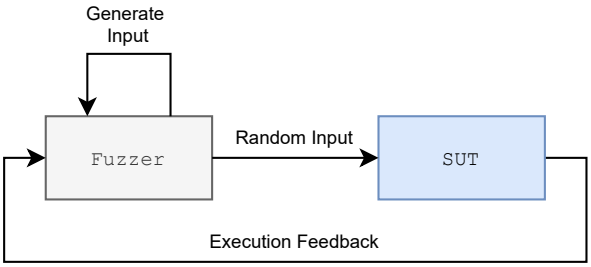

Figure 2.4: Generalized overview on how fuzzing operate.

2.3.1 Fuzzing Types

The effectiveness of fuzzing depends significantly on the approach used, with the three main categories being black-box, grey-box, and white-box fuzzing. Each offers distinct benefits and trade-offs based on the level of program visibility, execution context and instrumentation available.

Black-Box Fuzzing. Black-box fuzzing is the simplest and most general approach, where the tester has no access to or knowledge of the internal workings of the SUT. It treats the software as a "black box", relying solely on its input-output behavior. Inputs are generated randomly or using predefined patterns without feedback from the program's execution. The lack of instrumentation makes black-box fuzzing easy to implement and broadly applicable, particularly in environments where source code or internal details are unavailable. However, this simplicity often results in poor efficiency in finding deep or complex bugs, as the process may fail to systematically explore diverse or critical code paths.

White-Box Fuzzing. In contrast, white-box fuzzing provides complete access to the target program's internal logic and structure, utilizing techniques like symbolic execution or constraint solving to systematically explore all possible execution paths. This allows for precise identification of vulnerabilities and exhaustive testing, particularly in safety-critical applications. However, the computational overhead of white-box fuzzing can be substantial, and its scalability is limited when dealing with programs with a large state space or complex dependencies [16].

Grey-Box Fuzzing. Grey-box fuzzing bridges the gap between black-box and white-box approaches by providing partial visibility into the program's internal state through lightweight instrumentation. This balance allows testers to gain execution feedback without the computational costs of full symbolic analysis. Tools like American Fuzzy Lop (AFL) exemplify grey-box fuzzing by using code coverage metrics to guide input generation, prioritizing unexplored paths and avoiding redundant tests. Grey-box fuzzing has become the preferred method for many practical applications, offering an excellent trade-off between efficiency and depth of vulnerability discovery [17].

2.3.2 Coverage-Guided Fuzzing

Coverage-guided fuzzing is a specialized form of grey-box fuzzing that uses execution feedback, typically code coverage data, to inform and optimize the fuzzing process. By focusing on inputs that trigger previously unexplored paths, it maximizes testing efficiency and effectiveness. Popular tools like AFL, LibFuzzer, and Honggfuzz [18, 19, 2] utilize this approach, which has proven successful in uncovering numerous high-profile vulnerabilities. The adaptive nature of coverage-guided fuzzing makes it particularly suitable for large and complex software systems, where random input generation would likely fail to achieve sufficient coverage.

Fuzzing, by its very design, is only capable of detecting bugs in code that has been executed. In order to ensure that every part of the code has been executed, it is considered best practice to achieve high code coverage. In the context of fuzzing, two approaches are usually employed together: mutation-based fuzzing and source code instrumentation.

Mutation-based Fuzzing. Mutation-based fuzzing works by taking known inputs and subjecting them to random mutations. This approach is also attractive because it requires only a small number of sample inputs (seeds) for the target program. Once a seed has been mutated, it is added to the collection of inputs and used in a later execution.

Source Code Instrumentation. Source code instrumentation can facilitate the identification of new code paths or blocks that have been triggered. Its primary goal is to track code coverage, monitor execution paths in order to guide fuzzers to generate more effective inputs. Techniques include compile-time, runtime, and binary instrumentation, where extra instructions are added at specific points (for example at the beginning of a basic block) in the code.

While instrumentation improves fuzzing by providing detailed execution insights, it introduces performance overhead and code complexity. But this is a favourable tradeoff as exmplified:

The code in the function `process_data()` defined in Listing 2.1 causes a stack overflow that can be triggered when executing line 6. In order to successfully identify the first three characters, a black-box fuzzer would require a correct guess from among $2^{8\times3} = 2^{24}$ possible combinations. By using a coverage-guided fuzzer with source code instrumentation, each comparison step can be progressed step separately, which leads to $2^8 \times 3$ possible combinations, massively increasing the overall probability of generating an input that triggers the stack overflow during fuzzing [20].

```
1 void process_data(char *input, int len) {
2   char buffer [10];
3   if (len > 0 && input[0] == 'b')
4     if (len > 1 && input[1] == 'u')
5       if (len > 2 && input[2] == 'g')
6         memcpy(buffer, input, len);
7 }
```

Listing 2.1: Vulnerable C function, inspired by [21].

2.4 AFL++

AFL++ [22] is an edge-coverage-guided mutation-based fuzzer, approximating path-coverage by tracking the last two executed basic blocks [13], which boasts a feature-rich toolbox and wide soft- and hardware support. It is the successor of the original project AFL [23] and has the objective to consolidate all features and enhancements of AFL into a unified project. This project was initiated in response to the discontinuation of AFL development in mid-2017 and is now being run by its community. Since then, AFL++ has incorporated a multitude of features and reasearch results. It has extended its functionality to encompass diverse architectures and fuzzing modes, like fuzzing binaries, Android apps, fuzzing on MacOS or binary rewriters.

AFL++ provides a range of tools, including `afl-min`, which minimises the initial set of seeds, and `afl-showmap`, which provides information about the attained code coverage. The utilisation of these tools has the potential to increase the execution process and to facilitate the triaging of the fuzzing campaign and potential crashes.

Further, AFL++ supports a vast amount of command line arguments as well as environment variables, which provide a way to control how the mutator of the fuzzer works, allow to bind the execution to a CPU, specify a timeout parameter and more. AFL++ supports different compilers, Clang or GCC, and instrumentation modes. The introduction of further concepts is made when there is a need to do so. For a comprehensive list and detailed explanation of the features, please refer to the official documentation of AFL++ [24].

2.4.1 Fuzzing Overview

The first step when developing a fuzzing campaign using AFL++ is to define the `SUT`. In an ideal scenario, the SUT should comprise solely the functionality that is to be tested, with the objective of enhancing performance. The subsequent step is to instrument the target using one of the available modes. Then the initial inputs, called seeds, need to be defined and the fuzzing campaign can be started. This simply involves starting a binary called `afl-fuzz` with the correct command line arguments and the fuzzing starts.

Figure 2.5 illustrates a simplified overview of the manner in which the fuzzer works. When `afl-fuzz` is running, it uses the system calls `fork()` and `execv()` to create a new process and start the `SUT` respectively. Different techniques are used in order to increase the execution speed of AFL++. One such improvement is the so-called `forkserver`. The `forkserver` increases the execution speed by forking just before the code we want to test is executed, effectively removing the need to call `execv()` in each iteration. The communications between `afl-fuzz` and the `SUT` is then handled by the `forkserver`.

Figure 2.5: Simplified AFL++ fuzzing setup.

Once the system is operational, `afl-fuzz` provides the new test case via shared memory or a shared file and transmits a signal to the `forkserver`. A new process is forked, the test-case is read and used in the current execution. A coverage map (the trace bits) is employed for the purpose of indicating each used edge. Upon completion of the execution, whether successfully or due to a crash, the `forkserver` receives the process' status and relays it to `afl-fuzz`. The test-case is then subjected to mutation based on the status and the coverage map, and if it resulted in a crash, it is saved. Subsequently, a new run is initiated.

2.4.2 Mechanisms and Architecture

Figure 2.6 shows a detailed sequence diagram, that depicts all communications between the processes when running a fuzzing campaign. Note that many details of `afl-fuzz` were left out, as they are optional or not deemed relevant to the port. However, the interactions between the components have been accurately represented.

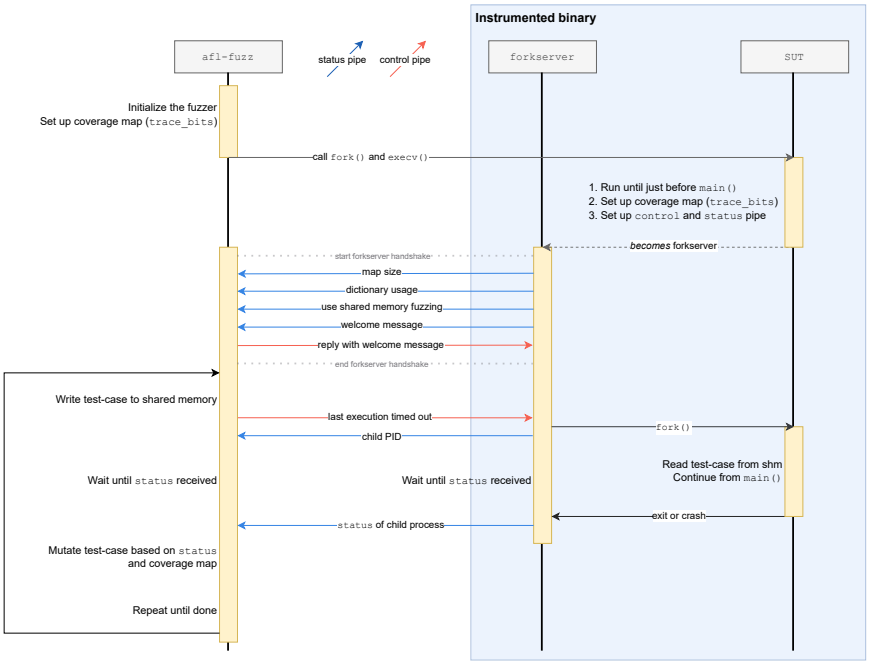

Figure 2.6: Sequence diagram of the communication between `afl-fuzz`, `forkserver` and the `SUT` running in default fuzzing mode and using shared memory fuzzing. Note that `execv()` is only executed once, while `fork()` is called in each iteration. Test-cases are communicated using shared memory, which is not depicted in the diagram. The status and control pipes refer to UNIX pipes, i.e. unidirectional data channels used for interprocess communication.

afl-fuzz

This component hosts the entry point, a `main()` function, of the fuzzer. Typically, it is initiated as a command-line executable, accepting a multitude of

flags and configurations. These allow the user to modify various aspects of the execution, mutator settings, fuzzing behaviour, test settings, and several other options. The execution process can be further influenced by the configuration of environment variables.

In order to execute `afl-fuzz`, it is required to specify an input and output directory. The input directory is used to provide one or several seeds to the fuzzer. The execution of `afl-fuzz` requires that at least one of the seeds does not result in a crash or timeout. The output directory contains every artifact created by the fuzzer, including program inputs that result in a crash and the specified configuration.

Based on the desired configuration, `afl-fuzz` will continue its initialization, for example setting up the shared memory segment and checking if the `SUT` has been instrumented or not. Then, if everything is ready to execute, this process will fork itself and start the `SUT` binary. See the `fork()` and `execv()` call in the sequence diagram.

Following the execution of the handshake with the `forkserver`, `afl-fuzz` initiates a dry run to evaluate the configuration. Subsequently, the program enters a loop where it takes the current test-case from a queue, writes it to shared memory, notifies the `forkserver` and awaits the response of the execution.

The execution falls into one of four options. It can run successfully, which is the expected option. If not, it will either be marked a crash, a hang or a timeout. It is labeled as a crash if the `SUT` returned a non-zero status code. The distinction between a hang and a timeout is that a hang will complete if the allocated time is doubled, whereas a timeout will not. So if the execution does not finish before the user defined doubled timeout, it is marked as a timeout.

In combination with the execution status and the trace bits, the test-case is evaluated and checked if it is deemed to be interesting. In such cases, the test case is saved to disk. Then, the test-case is subjected to multiple mutations, appended to the queue, and the current iteration is finished.

forkserver

Following the initialization of the `SUT`, the control flow of the program is transitioned to the `forkserver`, thereby the `SUT` *becomes* the `forkserver`.

Next, the `forkserver` does the handshake, exchanging a few more configuration details with `afl-fuzz`. Upon receipt of the notification from `afl-fuzz` that states whether the last execution resulted in a timeout, the server is then prepared to initiate a `fork()` and pass control flow back to the `SUT`. It waits until it has received execution feedback from the `SUT` and relays this information to `afl-fuzz`.

SUT

The `SUT` contains the actual code that is subject to testing. However, in order to ensure compatibility with AFL++, it is necessary to instrument its source

code. The documentation of AFL++ enumerates three instrumentation modes: `LTO` (link-time optimisation) mode, `LLVM` mode and `GCC_PLUGIN` mode. As will be argued subsequently in section 4.3.1, the utilisation of both `LTO` and `LLVM` mode is not a viable option to use with Genode. Therefore, the focus will be exclusively on the `GCC_PLUGIN` mode.

As the name implies, the `GCC_PLUGIN` mode uses a GCC-plugin [25]. By attaching itself to GCC, compilation information about the CFG is made available. Using this information, the `GCC_PLUGIN` can add additional compiler passes where code instructions, called instrumentation, are added to the binary.

The instrumented `SUT` can then be used and started by `afl-fuzz`. The `SUT`, similar to `afl-fuzz`, retrieves its configuration from environment variables. It initialises the shared memory segment and just before executing the `main()` function it becomes the `forkserver`. This ensures that when the `forkserver` forks itself, the execution will continue at the `main()` function, which is exactly where the code subject to testing starts.

By default, SUT reads the test case through standard input (stdin). In certain instances, this may suffice for the purpose of testing. However, in most cases, the user is required to write a harness. The purpose of the harness is to provide a translation layer between the test-case and the tested functionality. This translation layer is able to interpret the test-case and translate it into a comprehensible form. This can be illustrated by the ability to load the test-case into a specified data structure or to modify it to conform to a specific interface. This is where dictionaries can help: by defining a specific format to which the input must conform, the test case can be translated more easily.

Chapter 3

Problem Statement

Safety-critical devices play an increasingly vital role in modern society, forming the backbone of systems where failure can result in catastrophic consequences. These systems include medical devices such as pacemakers and infusion pumps, avionics systems in aircraft, industrial control systems, and components in the automotive sector like autonomous driving modules or braking systems. Given their impact, ensuring the reliability and security of the software that powers them is of utmost importance.

The software running on safety-critical devices is intentionally designed to be small and lean. This minimalistic approach reduces complexity, making it easier to verify and validate the software. Smaller codebases are generally easier to understand, maintain, and test, which is crucial for identifying and mitigating potential risks. They reduce the attack surface and provide a more deterministic execution model—features that are highly desirable in security-critical environments.

The "Genode Operating System Framework" is a prominent example of such an approach. It is a microkernel-oriented, capability-based framework designed to build secure operating systems. Its architectural principles emphasize modularity, strict separation of components through compartmentalization, and the principle of least privilege, all of which contribute to its suitability for safety-critical deployments.

Despite its strengths, Genode currently lacks support for one of the most effective bug-finding techniques in software development called fuzzing. Fuzzing, involves the automatic generation of large volumes of randomized or semi-randomized inputs to test programs in an attempt to uncover unexpected behavior, crashes, or security vulnerabilities. It has proven to be highly successful in many mainstream software projects such as OSS-Fuzz, a fuzzing initiative by Google, the integration of fuzzers into browsers like Chrome and Firefox, and extensive use within operating system kernels and libraries.

There is an apparent disconnect between the safety-critical systems built on Genode and the current state-of-the-art in software testing. While Genode emphasizes minimalism and formal reasoning, modern fuzzing techniques offer complementary benefits by uncovering bugs that might escape traditional review or formal analysis.

This thesis aims to bridge this gap by bringing fuzzing capabilities to the Genode framework. By doing so, we seek to enhance Genode's ability to identify and mitigate software vulnerabilities, thereby improving the safety and reliability of systems built on this framework. The introduction of fuzzing will enable developers to proactively discover and address bugs, ultimately contributing to the creation of more robust and secure safety-critical devices.

Chapter 4

System

A top-down approach was used while designing the port. The structure of this chapter mirrors this methodological approach. The initial section of this chapter offers a high-level overview through the utilisation of a component tree diagram. The subsequent section delves into the intricacies of component communication during startup and a single fuzzing iteration. The final implementation section provides a comprehensive overview of the port's functionality and the rationale behind the design decisions.

All code is publicly available on Github [26].

4.1 Overview

The design of the port was based on the overview in Figure 2.5. All three parts, `afl-fuzz`, `forkserver` and `SUT` were mapped to a set of Genode components. The outcome of this process is illustrated in the component tree depicted in Figure 4.1.

The six components that are grouped as "Fuzzing Engine" provide the functionality of both `afl-fuzz` and the `forkserver`. Note that the `forkserver` has

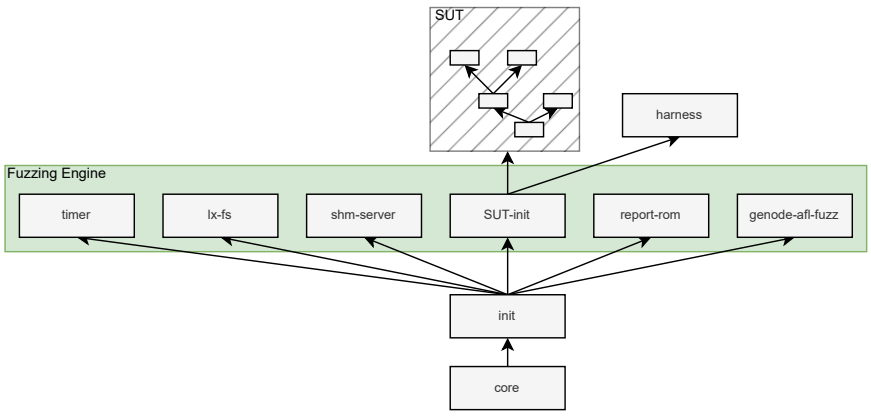

Figure 4.1: Overview of the component tree required to execute `afl-fuzz`.

been merged with `afl-fuzz` into a new component called `genode-afl-fuzz`.

Each component provides a unique but required service:

(1) The `timer` component provides timing information to check whether the current execution has exeperienced a timeout.

(2) The `lx-fs` component is needed to save the test-cases that AFL++ deemed interesting to disk. Here the "lx" refers to linux and "fs" to filesystem. So this needs to be updated according to the kernel on which the fuzzer is executed on.

(3) The `shm-server` component provides an translation layer between POSIX shared memory system calls[1] and Genode's shared memory implementation.

(4) The `SUT-init` component is responsible for starting the `SUT`.

(5) The `report-rom` component signals `SUT-init` about new component trees.

(6) The `genode-afl-fuzz` component contains all ported logic from AFL++ and all additional logic in order to enable the use of the other components making up the fuzzing engine.

The `SUT` is a component tree of any size and its functionality is invoked by the `harness`. In contrast to AFL++, the `harness` is always required, as it is responsible for translating the test-cases to the required format.

4.2 Execution Details

The interactions shown in the AFL++ sequence diagram in Figure 2.6 define an interface that we adhere to. Based on this interface, the required interactions between the Genode components were defined, as illustrated in Figure 4.2.

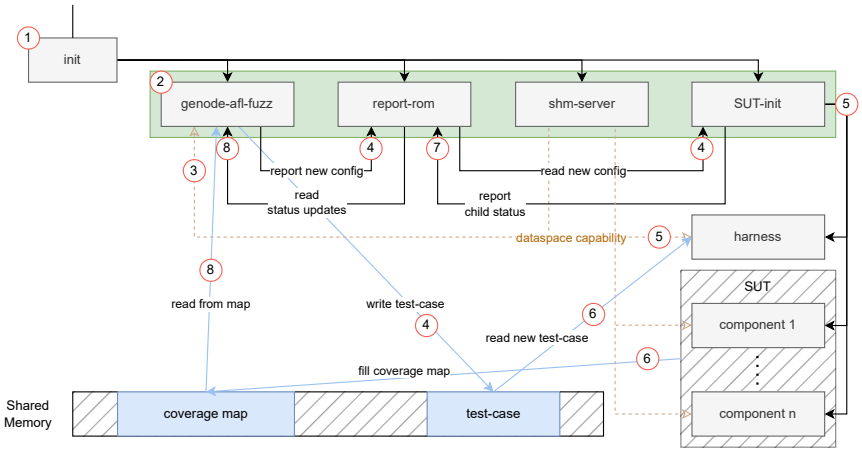

Figure 4.2: Diagram of the inter component communications taking place during start up and a single fuzzing iteration. The numerical values represent the sequence of communication steps.

[1] For example `shmget()` or `shmat()`.

The operational mechanics of such a fuzzing campaign are then as follows:

(1) The `init` component starts `genode-afl-fuzz`, `report-rom`, `shm-server` and `SUT-init`. Additionally, the `timer` and a `lx-fs` components are started. Despite the fact that both components contribute to the functionality of the port, they have been omitted from the diagram for the purposes of simplicity and conciseness.

(2) Once initialized, all components except `genode-afl-fuzz` wait and listen for Genode signals or RPC calls. Meanwhile, the `genode-afl-fuzz` program initiates the fuzzing process via `afl-fuzz`' program entry point. It reads the user-provided configuration and starts setting up all necessary data structures until it needs to attach a shared memory segment.

(3) Then, `genode-afl-fuzz` signals the `shm-server` through a custom session type to receive two shared memory segments: one for the coverage map and one for the test-cases. It will then write the first test-case into its newly attached shared memory segment.

(4) Next, a new config is generated and reported to the `report-rom`. This config contains a detailed description of the set up of the `harness` and all components that are part of the `SUT`. The `SUT-init` component then receives a signal that a new config is available and handles the request by starting all components specified in the config as children.

(5) Once the `harness` is started, the `shm-server` is again used to setup the same shared memory segments mentioned in step (3).

(6) The `harness` reads the test-case and forwards it to the `SUT`. Each newly covered basic-block during the execution of `SUT` is marked in the coverage map. This continues until the `SUT` exits, crashed or experiences a timeout.

(7) In addition to starting the config as a new child component, `SUT-init` monitors all specified children in regular intervals. This monitored state is then forwarded to the `report-rom` and further to `genode-afl-fuzz`.

(8) Upon indication from the state that the current execution of `SUT` has been completed, `genode-afl-fuzz` reads the content of the coverage map, mutates the test-case and resets the state to prepare for the next fuzzing iteration. A single test-case has now been processed and the execution loops back to step (4).

4.3 Implementation Details

The process of porting AFL++ to the Genode framework necessitated a multitude of alterations in order to ensure its compatibility. A notable example of this is the requirement to support system calls in order to execute `afl-fuzz`. Most system calls, however, are not natively supported by Genode. Fortunately, this is not the first project to encounter such requirements: Genode provides a libc plugin that supports the majority of system calls required to run AFL++, though not all of them. The missing system calls were substituted with native Genode functionality, which integrates more effectively within the framework and provides better compatibility for the `sut`.

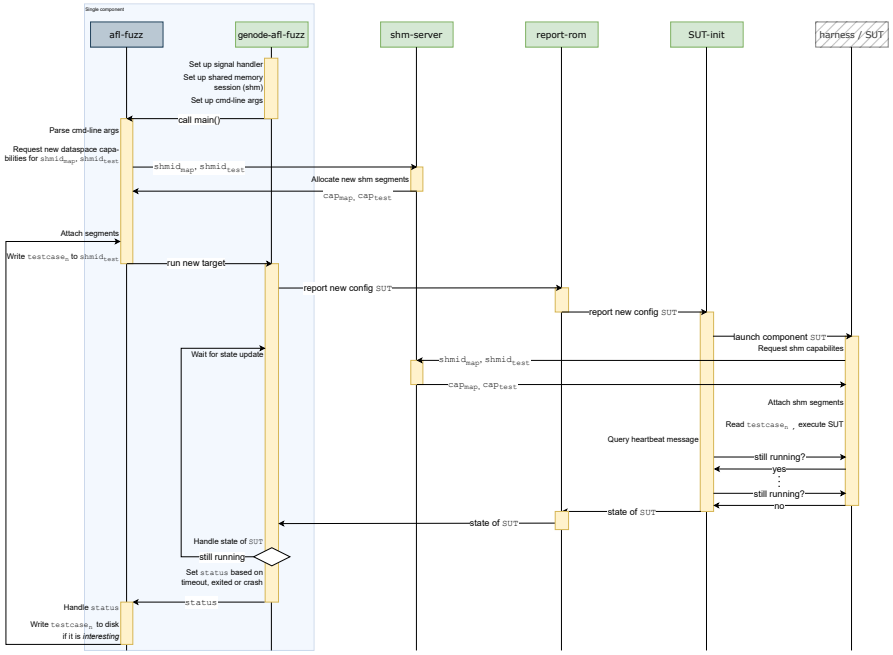

Figure 4.3: Sequence diagram of the AFL++ Genode port in default mode, so not depicting the more performant persistent mode. It expands the diagram in Figure 4.2 and shows every interaction between the components. Note that the `timer` and `lx-fs` are not shown for simplicity.

Figure 4.3 depicts the sequence diagram of the system's components and the communication protocols between them. The diagram is largely self-explanatory; however, the interesting design decisions are explained in greater detail in the following sections.

4.3.1 Instrumenting Genode Components

Genode employs a sophisticated tool chain that utilises Makefile and expect[2], that facilitates the compilation, linkage and execution of the project. Compiling and instrumenting the source code of the `SUT` using this tool chain is required in order to adhere to Genode's special requirements. This tool chain is designed to support custom rules and flags, thereby offering a high degree of customisability. Genode also provides binaries that, among others, facilitate the processes of compilation and linkage of the code, both of which are based on GCC.

Following this, a custom Genode library, named "`afl++`", was developed to facilitate source code instrumentation. This library fulfills two purposes: firstly it attaches the `GCC_PLUGIN` from AFL++ during compilation, and secondly, after a binary has been compiled using the `GCC_PLUGIN`, it provides the required variable and function definition during linking.

The utilisation of a Genode library seamlessly integrates the source code instrumentation mechanism with Genode's tool chain, necessitating only

[2]The script language based on Tcl.

minimal alterations. Furthermore, all customisation options available in AFL++ remain and require no additional change. This enables, fo example, the use of an allow list, thereby ensuring that only the files specified in the allow list are instrumented.

4.3.2 System under Test (SUT)

In the context of this port, the SUT is a Genode component tree. In order to fuzz a particular functionality or session, the user is tasked with defining this component tree and must adhere to the following points:

Firstly, a harness, i.e. a Genode component, needs to be defined. The harness is responsible for translating the test-case, which is only provided as simple a void pointer through shared memory, such that the session can be tested. This harness is analogous to that employed in AFL++.

Secondly, each component that executes instrumented code needs to attach the shared memory dataspace, which is required to fill the coverage map.

Thirdly, in the event of fuzzing core functionality – for example, logging or the PD – it will be necessary to build and link these separately. When the `core` and `init` components are initiated, they expect uninstrumented code and may consequently crash during the startup process. It is possible to fuzz an uninstrumented target, though this approach invariably leads to a reduction in efficiency.

4.3.3 Replacing the Forkserver

In each iteration of the fuzzing process in AFL++, the `forkserver` invokes the `fork()` system call, as illustrated in section 2.4.2. Support for this `fork()` system call exists in Genode's libc plugin and would therefore be available. Though utilising the `fork()` system call imposes limitations on the SUT that can be created. The `forkserver` anticipates executing a `main()` function subsequent to the fork call, which limits the SUT in Genode too much

The possibility of rewriting the ported `forkserver` in such a manner that it could utilise `fork()` was considered. However, this approach was ultimately dismissed due to the anticipation of further challenges. For instance, it was unclear how several components could be forked simultaneously or how the sessions behave if they are forked.

Therefore, the main purpose of the `forkserver` had become superfluous, and turned into an overhead. Consequently, the `forkserver` was merged with `afl-fuzz` into a unified component called `genode-afl-fuzz`. The communication steps between the `forkserver` and `afl-fuzz` were completely patched out. The `fork()` call was replaced with Genode-specific functionality, namely a `report-rom` and `init` component, which are invoked by the new `genode-afl-fuzz` component.

4.3.4 Monitoring the SUT

As illustrated in Figure 2.6, `afl-fuzz` utilises a system call to ascertain whether the current iteration has finished or until a timeout is reached. In this case

it uses the synchronous system call `select()`, that monitors the status pipe for updates or until a timeout is attained. This pipe is updated when the system call `wait()` from the `forkserver` finishes. Both these mechanisms are unavailable when the `forkserver` is patched out; the communication pipes are completely removed and `wait()` is never called.

Instead, the functionality is approximated by a built-in Genode implementation. The mechanism for monitoring the status of a component is termed the "heartbeat". This status comprises information such as the assigned RAM, assigned capabilities and, most importantly, the exit code, which is indicative of the end of the execution. Within a predetermined interval, the parent `init` component transmits a signal to each child component of the `SUT`, and a response is expected in return. This response is then transmitted to the `report-rom`, which is subsequently reported to `genode-afl-fuzz`.

In the event of a child component failing to respond to a heartbeat signal, the number of skipped heartbeat is also reported. The Genode documentation acknowledges that this mechanism does not guarantee 100% accuracy, and as such, a single skipped heartbeat is not necessarily indicative of a system crash. In view of the inaccurate information, it is necessary to map the result of the test-case to the four possible outcomes: success, crash, hang or timeout.

The successful execution of a process can be readily identified by the exit status code contained within the heartbeat message. In the event of an exit code being available and indicating a 0, the execution is to be considered successful. The distinction between a hang and a timeout is addressed by AFL++, rendering them indistinguishable for the purposes of this argument. Consequently, it is imperative to establish a method that can differentiate between a crash and a timeout.

There is not always a clear distinction between a timeout or a crash. As it turns out, this results in a race condition, as illustrated in Figure 4.4. In the upper example, `genode-afl-fuzz` sees the missed heartbeat messages, after the `SUT` has crashed, and thus, this execution is considered a crash. In the lower example, the SUT does not crash until after `genode-afl-fuzz` checks the second time for heartbeat messages, but before `genode-afl-fuzz` is able to detect the missed heartbeat messages. This particular execution of the `SUT` is therefore counted as a timeout and not as a crash.

The user is presented with a series of configuration options, the purpose of which is to provide a certain degree of control over the crash detection mechanism. The user can set (1) the interval of heartbeat messages. By default, every 5 ms a request is sent. (2) the number of skipped heartbeat messages. By default, if 50 skipped messages were reported, the execution is considered to have crashed. (3) the timeout. By default, this value is 200 ms. It is longer than the timeout from AFL++ due to the additional overhead of running Genode.

So it is imperative to set a suitable timeout value. Too great and the overall execution time is impacted, too low and a crash might go undetected, if it cannot be considered as a hang

To summarise, the combination of this heartbeat mechanism with a blocking call, designed to await incoming signals, results in functionality analogous

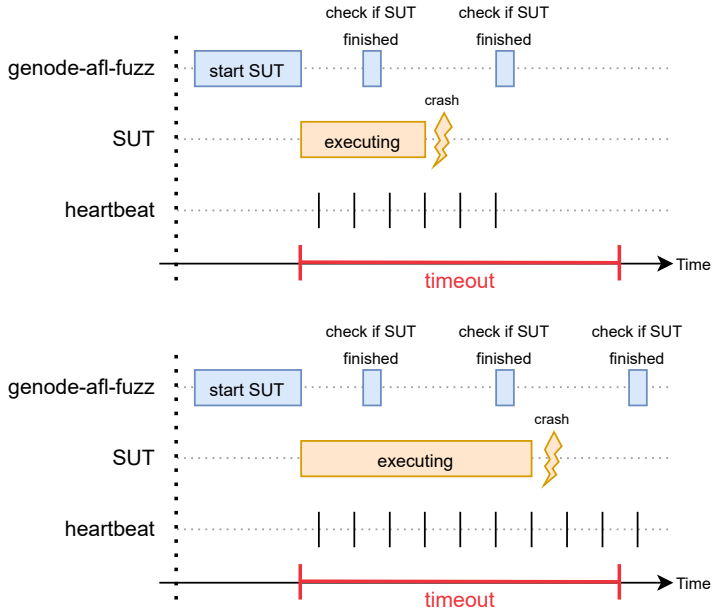

Figure 4.4: Example executions that illustrate the race condition of the monitoring functionality.

to that of the `select()` call. By using this approach, `genode-afl-fuzz` uses a purely Genode native solution to monitor the `SUT`.

However, it should be noted that this approach of waiting for incoming signals drastically slows down the fuzzing execution speed. It was discovered that the waiting mechanism was accountable for a decline in execution speed of 20 to 25 times. This issue is addressed in the following section.

4.3.5 Persistent Mode

AFL++ provides an option to execute the fuzzing campaign using the so called 'persistent mode', which allows for a much higher execution speed. The magnitude of this speed increase is reported to be in the range of 10 to 20 times faster than the original speed. The `genode-afl-fuzz` component supports this mode as well and it is enabled by default.

The intuition is, that only a small number of executions result in a system crash or timeout. Consequently, the overhead can be significantly reduced by reusing the existing, already running components in each iteration. This approach necessitates that the user resets the state in each iteration, ensures that multiple calls can be executed without resulting in resource leaks and that earlier iterations do not affect subsequent ones.

Following, a new approach for monitoring the `SUT` is required, given that the `SUT` does not exit on successful executions and responds to all heartbeat messages. In case of a successful execution, the reports from the `report-rom` lack the necessary information. To address this, a shared byte is used to

transfer the status between `genode-afl-fuzz` and the `SUT`. The value is polled on both sides to detect changes and updated accordingly.

The merit of this monitoring approach is twofold. Firstly, it eliminates the overhead of reporting and creating a new component tree in each iteration, and secondly, it removes the inefficient waiting for signal functionality. This optimization resulted in a execution speed increase of up to 40 times!

Chapter 5

Evaluation

5.1 Experimental Setup

Two different testing setups were designed, in order to evaluate effectiveness and efficiency of the AFL++ port. The development and execution of the Genode port was conducted within a Virtual Machine (VM), so to ensure a level playing field all tests were also executed within the same VM. The virtual machine is a 'Debian GNU/Linux 12 (bookworm)' system, executed on a 'Lenovo ThinkPad P14s Gen 3' with a '12th Gen Intel Core i7-1260P $\times$ 10' and using 6 GB of memory. The version of AFL++ utilised for the purposes of testing is designated 'afl-fuzz++4.21c'.

5.1.1 Performance and Correctness

Execution speed is a common metric by which the effectiveness of a fuzzer is measured and compared. Because there is no other fuzzer available for Genode, the performance was compared between AFL++ running on Linux and running on Genode.

A rudimentary executable was constructed for Genode and again, using the same functionality, for Linux. The executable contains certain bugs, which were deliberately introduced to be detected by the fuzzer. The bugs included are an infinite loop, a floating-point division by zero and a segmentation fault. The duration of both fuzzing campaigns was 6 hours. This duration is generally considered to be relatively brief; however, it was sufficient to obtain the desired results.

No special flags or environment variables, that interfere with the execution speed, were set and both setups use the `GCC_PLUGIN` for instrumentation.

5.1.2 Init Component

The `init` component is widely used component in Genode, rendering it a primary target for fuzzing. Figure 5.1 shows the exact component tree used as the setup: the SUT is constituted of three components, namely, a `report-rom`, an `init` component and a component called `print-component`. The `init` component is duplicated an renamed to "`init-instrumented`".

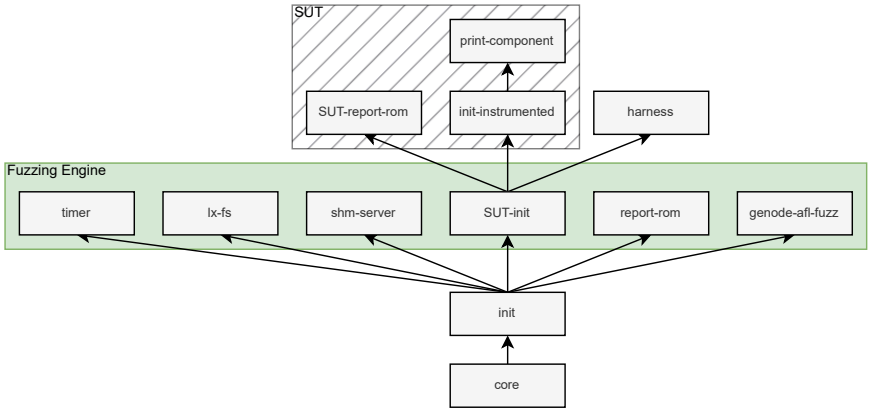

Figure 5.1: Exact depiction of the component tree that was used to fuzz the `init` component.

This means that its source code is instrumented, as the name suggests. A harness then writes a new report based on the fuzzing test-cases and through the `SUT-report-rom`. The initial seed reports the valid `print-component`, a component that no real purpose beside printing a single line.

No special flags or environment variables have been set for testing.

5.2 Results

5.2.1 Performance and Correctness

For comparing we focus on three key metrics, namely total executions, total crashes and total timeouts. The results are listed in Table 5.1. The vulnerabilities are easily reachable and were therefore triggered many times.

Within a time frame of 6 hours, AFL++ executed the program 75.2M times, which equates to approximately 3'481 executions per second. The Genode port attained a total of 70.4M executions, amounting to 3'259 executions per second. This places the Genode port at approximately 93% of the execution speed of AFL++.

	Total Executions	Total Crashes	Total Timeouts
AFL++	75.2M	2	90.9k
Genode	70.4M	37.6k	31.4k

Table 5.1: Table depicting the results of fuzzing an example program containing several intentional vulnerabilities during 6 hours.

Next, AFL++ reported only two total crashes, whereas the port detected over 37 thousand crashes. When examining the total number of reported timeouts, AFL++ recorded over 90 thousand timeouts in comparison to the 31 thousand recorded for the Genode port. This discrepancy shows the limitations of the signaling handling used for monitoring the status, a topic that is explained in section 4.3.4. Ultimately, both fuzzers successfully identified and reported the introduced vulnerabilities, showing the correctness of the ported fuzzer.

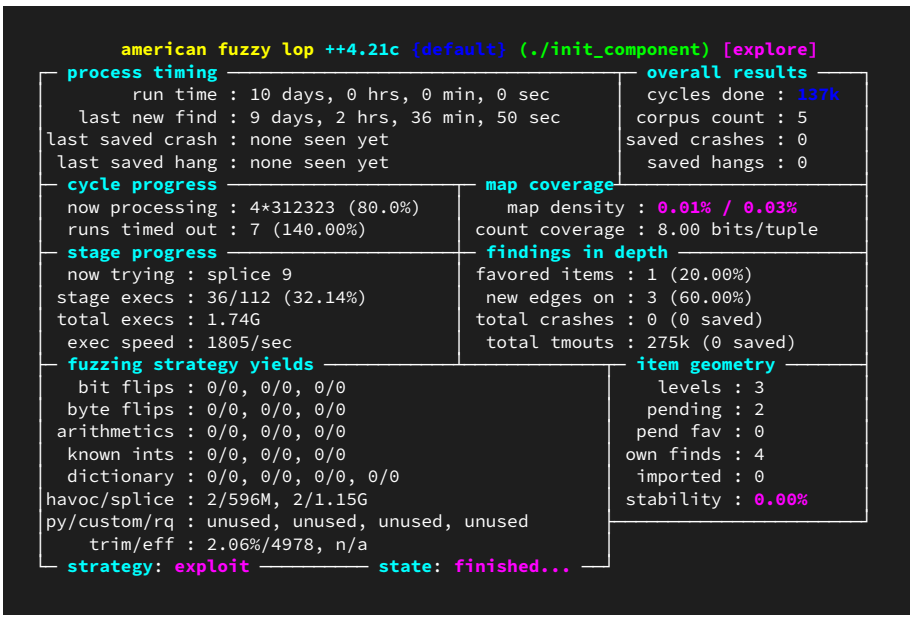

Figure 5.2: Final status screen of the init component fuzz run. A detailed explanation of all labels is provided in the official documentation [27]. Most labels give an optional peek at what the fuzzer is actually doing, such as the "stage progress", "findings in depth" and "fuzzing strategy yields" labels, and therefore not of interest.

5.2.2 Init Component

In this fuzzing campaign the fuzzer executed over 1.74 billion testcases within a time frame of 10 days, so around 2'013 executions per second. An excerpt from AFL++ status page at the end of fuzzing is shown in Figure 5.2.

While no crashes and no hangs were detected, this extended execution of the fuzzer proved the stability of the fuzzer and shows that the source code instrumentation is working as expected. In "process timing" the label "last new find" indicates that a new path has been found, thereby indicating that the coverage map operates as expected.

Chapter 6

Related Work

Since the inception of fuzzing in 1990 [28], a plethora of fuzzers, along with their associated forks and mutators, have been developed, and their numbers continue to increase.

A paper from 2019 provided an overview of around 50 fuzzers, starting in 2001 and up to 2019 [6] and at the time of writing, Github lists over 8 thousand public repository related to fuzzing [29].

In a SoK paper from 2024 [30], the authors conducted a comprehensive study of 289 fuzzing papers from 2018 to 2023, from which 150 were randomly selected for manual analysis. Among the findings, it was determined that 33% of papers are concerned with the development of a new fuzzing tool, suggesting that approximately 15 fuzzers are released per year. These fuzzers have been observed to specialise in a particular system or environment.

There are no fuzzers that can run on top of Genode and fuzz Genode sessions. So our research includes a range of fuzzers that cover general purpose fuzzers, fuzzing libraries, and fuzzers that work in constrained systems, such as embedded systems.

6.1 Microkernel Operating Systems

6.1.1 Qubes OS

Qubes OS [31] is an open-source operating system designed with a strong focus on security and privacy. It leverages Xen-based virtualization to create isolated environments, known as “qubes”, each running in its own VM. This architecture minimizes the risk of security breaches by compartmentalizing different applications groups, for instance work, personal, and online activities, into separate qubes.

The core principle behind Qubes OS is “security by isolation.” By isolating different tasks and applications into separate VMs, Qubes OS ensures that a compromise in one qube does not automatically affect others. This isolation extends to hardware components, network connections, and storage devices.

On a fundamental level, there exists a strong resemblance between Qubes OS and Genode. Though in contrast, Genode employs a fundamental different

approach. Genode constructs a system from the ground up, characterised by its fine-grained multi-component design, while Qubes uses a sandboxing mechanism, which utilises Xen-based virtual machines. It is conceivable that Qubes could be executed within the Genode Framework, thereby integrating the two approaches.

With regard to fuzzing, Qubes OS tests a selected set of functionality and utilises OSS-Fuzz[1] and, more specifically, libFuzzer. This constitutes a fundamental difference in the approach adopted, as the objective of this thesis is to fuzz Genode sessions rather than a function. Qubes OS' approach is more akin to "normal" fuzzing on Linux.

6.1.2 Fuchsia

Fuchsia is an operating system being developed by Google [32]. It is built from the ground up with a microkernel architecture, known as Zircon, which is composed of a kernel as well as a set of userspace services, drivers, and libraries. The system's core platform facilitates the initiation of the boot sequence, communication with hardware components, the loading of user-space processes, and their subsequent execution. Furthermore, the implementation of the majority of system components in user space and isolation serves to reinforce the principle of least privilege.

Fuchsia has a similar goal to Genode in that it tries to provide a simple and secure operating system, intended to reduce the amount of trusted code running in the system. A further similarity is the utilisation of a component as an abstraction for a sandbox, where process are being run in.

In contrast to Genode, Fuchsia incorporates built-in support for fuzzing within its tool chain. In this case, the software utilises libFuzzer, thereby necessitating the use of LLVM. In order to facilitate the development of a fuzzer, Fuchsia provides a set of helper functions that aid splitting fuzz input into multiple parts of various types. Furthermore, custom mutators and dictionaries are supported. Fuchsia offers a wrapper around their fuzzing functionality which can be accessed through a custom fuzz shell [33].

Despite the similarities in the overall microkernel architecture employed by both Genode and Fuchsia, their tool chain fuzzing approach is on the wrong layer. This approach can offer a valuable tool for testing functionality that does not rely on the IPC, for example a parser.

6.2 Fuzzers and Fuzzing Libraries

6.2.1 Honggfuzz

Honggfuzz is an advanced and efficient feedback-driven fuzzing tool [2]. It employs feedback mechanisms such as code coverage and hardware-based tracing (using Intel's Processor Trace or perf) to intelligently guide the fuzzing process towards unexplored execution paths, increasing the likelihood of discovering critical bugs.

[1]This project, a collaborative effort between Google and OpenSSF, tries to increase the security and stability of common open-source software [4].

Key features of Honggfuzz include its high performance, ease of integration, and support for multiple target platforms, including Linux, macOS, Android, and more. It also offers advanced capabilities like persistent fuzzing, thread safety, and detailed crash analysis, making it a popular choice for security researchers and developers seeking to enhance software robustness.

Similar to AFL++, Honggfuzz is a mature fuzzer that is extensively utilised for fuzzing and as a foundation for constructing additional fuzzers. It possesses a substantial number of features comparable to those of AFL++, at least in the context of this project. It also supports instrumentation using GCC, thus rendering it a prime candidate for porting to Genode.

In a 2022 paper [34], several fuzzers were subjected to a comparative analysis, including Honggfuzz and AFL++. The fuzzers were run for several hundred days on a manually curated benchmark suite consisting of 12 real-world software systems. In terms of their effectiveness in finding bugs Honggfuzz, compared to AFL++, ranked equally when using Friedman Rankings or an adjusted ranking approach, and ranked slightly higher when using Linear Score or Effect Size Rankings Quadratic Score. In conclusion, the summary asserts the following:

> "If the most important metric is effectiveness in finding bugs, honggfuzz would be the best choice as it ranked first in almost all of our rankings. [...] Finally, as a compromise between these two aspects (i.e., effectiveness and efficiency), we would recommend the use of AFL++." (Asprone et al.; 2022; p. 9)

Magma [35], a ground-truth fuzzing benchmark to compare different fuzzers, also compared Honggfuzz and AFL++. This benchmark employs the discovery of unique bugs as the primary metric for assessing fuzzer performance. Their findings indicate that Honggfuzz and MOpt-AFL significantly outperformed other fuzzers. Specifically, Honggfuzz excelled in 4 benchmarks, while MOpt-AFL performed best in 3. Additionally, they observed that AFL, AFLFast, and AFL++ showed comparable performance across most targets, but these fuzzer scored lower than Honggfuzz.

Another benchmark, FuzzBench [3], yielded contradictory results in comparison to Magma. The functionality of FuzzBench is analogous to that of Magma, but additionally uses code coverage as part of the metric. In this study, AFL++ was found to be the most efficient and effective fuzzer in five out of seven different experiments.

To summarise, there is no clear consensus regarding the relative efficiency of Honggfuzz and AFL++ as fuzzers. Consequently, relevant to this thesis was on the effort required in porting this fuzzer to the Genode framework, which was not clear beforehand. So in the end it came down to personal preference and how many resources such as documentation are available. In this regard, AFL++ emerged as a superior option, by offering a wealth of information and resources that facilitated a more profound comprehension.

6.2.2 libFuzzer

LibFuzzer [19] is an in-process, coverage-guided, evolutionary fuzzing engine. It is a popular choice for fuzzing due to its integration with LLVM and its ability to seamlessly interface with sanitizer tool chains, which include for instance the AddressSanitizer and UndefinedBehaviorSanitizer. The effectiveness and ease of use of libFuzzer have made it a preferred choice for many projects.

The primary disadvantage associated with libFuzzer is its reliance on LLVM, a tool which is incompatible with Genode. However, it would have been possible to write a program that instruments the source code without relying on LLVM. But given the abundance of alternative fuzzers available, we swiftly eliminated libFuzzer as a viable option and did not pursue further research. Moreover, the original authors of libFuzzer have ceased active work on the project; therefore, no new features will be added.

6.2.3 libAfl

LibAFL [36] is a versatile and extensible fuzzing library designed to facilitate the development and experimentation of custom fuzzing solutions. By providing a modular architecture, libAFL enables to seamlessly integrate various fuzzing techniques and algorithms. This flexibility allows for the creation of tailored fuzzers that can address specific testing requirements or explore novel fuzzing strategies in special environments.

This library appears to be a particularly effective approach, as it was designed to function in unconventional environments, such as Genode. Further, libAFL does not necessitate the utilisation of any core libraries such as libc. It has been constructed with the programming language Rust, which support was integrated into Genode in 2023 [37]. Another advantageous feature would have been the support of customised instrumentation back-ends.

However, one of the external dependencies is Clang, the C/C++ compiler. At present, the Rust programming language does not provide support for all of the necessary compiler features, including weak linking and LLVM builtins linking. This necessitates the use of Clang.

Despite the confirmation by the libAFL maintainers of the theoretical possibility of using GCC instead of Clang, they explicitly advised against its use. Citing the significant effort required and the fact that the GCC Rust compiler is still in its early stages and not yet suitable for compiling real Rust programs. Consequently, this library was not given further consideration.

6.2.4 Fuzzing Embedded Systems

The application of fuzz testing to embedded systems has gained traction over the last couple of years, as evidenced by the development of numerous fuzzers that target different communication interfaces, including side-channels, peripheral drivers or DMA [38, 39, 40, 41].

The execution of software in an embedded system context is typically subject to certain constraints, including limitations in terms of resources or communi-

cation channels. It was thought that this restricted environment would be a promising approach, as Genode can be used in this context and is generally more restrictive compared to general purpose OSes.

The proposed fuzzers usually focus on fuzzing through unconventional communication methods, for instance through an UART debugging interface, which limited their overall usefulness. Nevertheless, they did provide some novel ideas on how to generally port AFL++ to Genode.

For example FIRM-AFL [41] proposes an interesting way to fuzz firmware in an emulated setting. FIRM-AFL is configured to start `afl-fuzz` in user-mode QEMU, and to instrument the branch transitions of the target program. The idea then is to be able to reuse as much workflow from AFL++ as possible. So each time the `forkserver` and the `SUT` are started, user-mode QEMU is replaced with an augmented process emulation. Meaning inside this process emulation the `fork()` call is removed, and instead a new snapshot of an already running VM is created and then started. After the execution there is switch back to the user-mode QEMU.

The objective is to increase the fuzzing performance within the QEMU environment by enabling the target user process to execute within the more performant user-mode for as long as is possible before switching to the more expensive augmented process emulation. This approach minimises the overhead associated with the translation process and leaves much of AFL++' workflow as is.

Whilst the aforementioned approach is not directly applicable to our system, it has provided the foundation for the development. The inspiration for this approach stems from the utilization of AFL++' fuzzing mechanism and interface. By adhering to the defined communication steps between `afl-fuzz` and the `forkserver`, most of AFL++' workflow can be used.

6.3 Rationale for Selecting AFL++

In order to achieve fast execution times, many coverage-guided fuzzers [24, 2, 42] are deeply intertwined with the operation system they are running on. Given that we are operating within the Genode Framework, a number of traditional operating system features will not be directly available. As a result, the selection of an appropriate fuzzer was contingent upon the identification of a tool with a minimal reliance on the aforementioned system features.

The next constraint is that Genode and its components can only be compiled using GCC, meaning Clang/LLVM cannot be used. This limits our options as many modern fuzzers use the built-in LLVM support to instrument the source code in order to track the coverage.

Following a thorough consideration of the available options, it was determined that AFL++ would be the most suitable option in this instance, due to several reasons:

Firstly, AFL++ provides explicit support for source code instrumentation using GCC. Secondly, AFL++ is regarded as a state-of-the-art fuzzer due to its incorporation of current fuzzing research into its tools, such as AFLFast [43]

and MOpt-AFL [44]. The argument is further compounded by the observation that a significant proportion of custom fuzzers are built on top of AFL++. Thirdly, the tool is a well-established fuzzer with many features and customisation options, which allows fine-tuning of execution of the fuzzer. Fourthly, given the popularity of AFL++, a many resources are available to facilitate a more profound comprehension of the fuzzer, like the official documentation, academic papers and community blog posts from security engineers and enthusiasts [45]. Lastly, existing familiarity with the AFL facilitated the decision-making process.

Chapter 7

Discussion

7.1 Limitations

As demonstrated in section 4.3.4, the mechanism to detect the current status of the SUT is not equivalent to the one employed by AFL++. Depending on the used flags and configuration, this can lead to drastically different results between AFL++ and the port, as shown in section 5.2.1. It is acknowledged that there are further mechanisms which could assist in reducing the discrepancy; for instance, tracking the CPU or memory of the SUT to ascertain whether an error has been experienced. This approach would facilitate the determination of whether a system crash occurred, thereby reducing reliance on heartbeat messages. But due to time constraints this approach is not implemented in this version of the port.

A further limitation is the way source code instrumentation works. A significant proportion of responsibility is ascribed to the user. Initially, the user has to ascertain that the instrumentation does not hinder the Genode start-up sequence. At start up, Genode might crash if some of the required binaries are instrumented. In such cases, the code responsible for the session must be duplicated, compiled, instrumented and then linked correctly. Thus, the instrumentation of core functionality requires a substantial engineering effort.

Next, as shown in section 2.4, AFL++ provides a lot of features that can provide performance or developing benefits. While it is conceivable that these features could function within Genode without additional effort, it should be noted that no resources have been used ensuring their operability. The majority of these features have not been tested. But, it would have been especially interesting to investigate the performance-enhancing features of this system. These could include multi-core/thread execution, sanitizers that help to detect errors earlier, or custom dictionaries to generate more suitable seeds. However, due to time constraints, the incorporation of such features was not a viable option.

7.2 Future Work

The next step regarding this port is to address the limitation of feedback collection. Obtaining more precise feedback would improve the port in several

areas. Firstly, the number of reported false positive crashes would be reduced, thereby minimising the amount of work required to debug a reported crash. Secondly, by reducing reliance on heartbeat messages for crash detection, the impact of the race condition would be limited. Thirdly, the overall execution speed would increase, as a crash could be detected more rapidly.

Next, fuzzing the init component, as outlined in the evaluation, should be conducted over an extended period. Typically, such a fuzzing campaign would necessitate several weeks, or even months, of dedicated effort. However, due to the temporal limitations imposed on this thesis, it was not possible to conduct fuzzing over such an extended period.

When a fuzzing campaign is started, improving the overall performance should be considered. We attribute the speed discrepancy of 7% between AFL++ and the port to the overhead introduced from the communication using signals as well as the reporting mechanism that is used. But no proper profiling has been conducted to potentially detect other bottlenecks or slowdowns introduced by the port.

Multi-thread or multi-core execution would be an ideal use case for fuzzing, as each iteration is independent and can therefore be parallelised. AFL++ facilitates the synchronisation of the outcomes of each fuzzing process at regular intervals. Another improvement could be to incorporate the functionality of `init` in the `afl-fuzz` component. This would remove the needs to use `report-rom` component and instead allow `genode-afl-fuzz` to directly launch new components.

In order to further test the robustness and functionality of the port, an additional thorough fuzzing example should be implemented, with the NIC-session being a potential candidate for testing. This would serve to highlight any potential shortcomings in the user experience when fuzzing on Genode, or any potential undetected discrepancies between AFL++ and the port.

Lastly, it became apparent that AFL++ requires many changes and fixes to ensure its functionality. While this was anticipated to a certain extent, in retrospect, libAFL might have also provided a robust method for developing a fuzzer. With libAFL demonstrating its particular strengths within an environment such as Genode, it appears that this would represent an optimal approach. However, we hypothesised that the technical challenges associated with the development and execution of this library would not outweigh the advantages, particularly in comparison to AFL++. Consequently, the potential of employing libAFL was not fully explored.

Chapter 8

Conclusion

This thesis presented the Genode OS framework, a unique and innovative approach to operating system architecture, focusing on security and modularity. However, the absence of fuzzing support has been a notable limitation, given the effectiveness of fuzzing in identifying software vulnerabilities.

This thesis has addressed this gap in the Genode OS framework by adding fuzzing capabilities through the successful porting of AFL++, a state-of-the-art fuzzer. Through careful adaptation and integration, we demonstrated that AFL++ can operate within the constraints of the Genode environment, achieving 93% of the fuzzing speed observed on a general-purpose operating system. This performance serves to illustrate the feasibility and practicality of integrating fuzzing into Genode.

Furthermore, the fuzzer was applied to a security-critical Genode component, illustrating its value in uncovering potential defects in high-assurance software. By bridging the gap between Genode's security-centric design and modern dynamic testing methods, this work contributes to enhancing the reliability and trustworthiness of systems built on Genode. It also lays the groundwork for further exploration of automated testing techniques in safety-critical, minimal operating system environments.

Bibliography

[1] P. Oehlert, "Violating Assumptions with Fuzzing," en, *IEEE Security and Privacy Magazine*, vol. 3, no. 2, pp. 58–62, Mar. 2005, ISSN: 1540-7993. DOI: 10.1109/MSP.2005.55. [Online]. Available: http://ieeexplore.ieee.org/document/1423963/ (visited on 03/28/2025).

[2] R. Swiecki, *Honggfuzz*, en-US, 2024. [Online]. Available: https://honggfuzz.dev/ (visited on 01/06/2025).

[3] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2021.

[4] Google, *OSS-Fuzz*, en-US, Technical Documentation, 2025. [Online]. Available: https://google.github.io/oss-fuzz/ (visited on 03/25/2025).

[5] Microsoft, *GitHub - microsoft/onefuzz: A self-hosted Fuzzing-As-A-Service platform*, en, 2020. [Online]. Available: https://github.com/microsoft/onefuzz (visited on 03/27/2025).

[6] V. J. M. Manes *et al.*, *The Art, Science, and Engineering of Fuzzing: A Survey*, 2018. DOI: 10.48550/ARXIV.1812.00140. [Online]. Available: https://arxiv.org/abs/1812.00140 (visited on 03/27/2025).

[7] Y. Li *et al.*, "G-Fuzz: A Directed Fuzzing Framework for gVisor," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 1, pp. 168–185, Jan. 2024, ISSN: 1545-5971, 1941-0018, 2160-9209. DOI: 10.1109/TDSC.2023.3244825. [Online]. Available: https://ieeexplore.ieee.org/document/10049484/ (visited on 03/27/2025).

[8] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, "RPG: Rust Library Fuzzing with Pool-based Fuzz Target Generation and Generic Support," en, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Apr. 2024, pp. 1–13, ISBN: 9798400702174. DOI: 10.1145/3597503.3639102. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639102 (visited on 03/27/2025).

[9] B. Schneier, *Beyond fear: thinking sensibly about security in an uncertain world*, eng, Repr. New York, NY: Copernicus Books, 2006, ISBN: 9780387026206.

[10] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, Nov. 2011, ISSN: 0098-5589. DOI: 10.1109/TSE.2010.81. [Online]. Available: http://ieeexplore.ieee.org/document/5560680/ (visited on 03/28/2025).

[11] G. Labs, *Genode - Genode Operating System Framework*, English, Technical Documentation. [Online]. Available: https://genode.org/index (visited on 11/04/2024).

[12] N. Freske, *Introducing Genode*, en, Brussels, Feb. 2012. [Online]. Available: https://genode-labs.com/publications/nfeske-genode-fosdem-2012-02.pdf (visited on 03/17/2025).

[13] M. Payer, *Software Security: Principles, Policies, and Protection*, 0.37. HexHive Books, Jul. 2021. [Online]. Available: http://nebelwelt.net/SS3P/.

[14] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "PathAFL: Path-Coverage Assisted Fuzzing," en, in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, Taipei Taiwan: ACM, Oct. 2020, pp. 598–609, ISBN: 9781450367509. DOI: 10.1145/3320269.3384736. [Online]. Available: https://dl.acm.org/doi/10.1145/3320269.3384736 (visited on 03/18/2025).

[15] owasp, *Fuzzing — OWASP Foundation*, en, Dec. 2024. [Online]. Available: https://owasp.org/www-community/Fuzzing (visited on 12/13/2024).

[16] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," en, *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, Jun. 2005, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1064978.1065036. [Online]. Available: https://dl.acm.org/doi/10.1145/1064978.1065036 (visited on 01/06/2025).

[17] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, *Greybox Fuzzing of Distributed Systems*, 2023. DOI: 10.48550/ARXIV.2305.02601. [Online]. Available: https://arxiv.org/abs/2305.02601 (visited on 01/06/2025).

[18] M. Zalewski, *American fuzzy lop*, en, 2017. [Online]. Available: https://lcamtuf.coredump.cx/afl/ (visited on 03/31/2025).

[19] *libFuzzer – a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation*. [Online]. Available: https://llvm.org/docs/LibFuzzer.html (visited on 03/21/2025).

[20] M. Eisele, D. Ebert, C. Huth, and A. Zeller, "Fuzzing Embedded Systems Using Debug Interfaces," en, in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).*, Seattle, USA, May 2023. [Online]. Available: https://publications.cispa.saarland/3950/ (visited on 10/30/2024).

[21] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," en, *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, Mar. 2012, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2093548.2093564. [Online]. Available: https://dl.acm.org/doi/10.1145/2093548.2093564 (visited on 03/17/2025).

[22] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[23] M. Zalewski, *American fuzzy lop*, en, Jun. 2020. [Online]. Available: https://lcamtuf.coredump.cx/afl/ (visited on 01/21/2025).

[24] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, *AFL++ - Repository*, original-date: 2019-05-28T14:29:06Z, Jan. 2022. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus (visited on 03/20/2025).

[25] GNU, *Plugins (GNU Compiler Collection (GCC) Internals)*, en, Technical Documentation. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Plugins.html (visited on 03/20/2025).

[26] S. Meier, *Washipp/genode*, original-date: 2024-10-07T08:40:26Z, Oct. 2024. [Online]. Available: https://github.com/Washipp/genode (visited on 04/15/2025).

[27] AFL++, *AFLplusplus/docs/afl-fuzz_approach.md at stable · AFLplusplus/AFLplusplus*, en, 2021. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md#understanding-the-status-screen (visited on 04/13/2025).

[28] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," en, *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/96267.96279. [Online]. Available: https://dl.acm.org/doi/10.1145/96267.96279 (visited on 03/27/2025).

[29] Github, *Build software better, together*, en, Code Repository, 2025. [Online]. Available: https://github.com (visited on 03/31/2025).

[30] M. Schloegel *et al.*, "SoK: Prudent Evaluation Practices for Fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2024, pp. 1974–1993, ISBN: 9798350331301. DOI: 10.1109/SP54263.2024.00137. [Online]. Available: https://ieeexplore.ieee.org/document/10646824/ (visited on 03/27/2025).

[31] M. Marczykowski-Górecki, *Qubes OS: A reasonably secure operating system*, en, 2025. [Online]. Available: https://www.qubes-os.org/ (visited on 03/25/2025).

[32] Google, *Fuchsia*, en, Technical Documentation, 2021. [Online]. Available: https://fuchsia.dev/ (visited on 04/11/2025).

[33] Google, *Run a fuzzer*, en, Technical Documentation, 2021. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/testing/fuzzing/run-a-fuzzer (visited on 04/13/2025).

[34] D. Asprone, J. Metzman, A. Arya, G. Guizzo, and F. Sarro, "Comparing Fuzzers on a Level Playing Field with FuzzBench," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Valencia, Spain: IEEE, Apr. 2022, pp. 302–311, ISBN: 9781665466790. DOI: 10.1109/ICST53961.2022.00039. [Online]. Available: https://ieeexplore.ieee.org/document/9787836/ (visited on 03/21/2025).

[35] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. DOI: 10.1145/3428334. [Online]. Available: https://doi.org/10.1145/3428334.

[36] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22, event-place: Los Angeles, U.S.A., ACM, Nov. 2022.

[37] B. Lamowski, *Bringing Rust back to Genode*, en, Blog, May 2023. [Online]. Available: https://genodians.org/atopia/2023-05-30-bringing-rust-back-to-genode (visited on 03/24/2025).

[38] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella, "Embedded fuzzing: A review of challenges, tools, and solutions," en, *Cybersecurity*, vol. 5, no. 1, p. 18, Sep. 2022, ISSN: 2523-3246. DOI: 10.1186/s42400-022-00123-y. [Online]. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-022-00123-y (visited on 10/01/2024).

[39] X. Feng *et al.*, *Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference*, 2021. DOI: 10.48550/ARXIV.2105.05445. [Online]. Available: https://arxiv.org/abs/2105.05445 (visited on 03/27/2025).

[40] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic Firmware Emulation through Invalidity-guided Knowledge Inference," en, 2021, pp. 2007–2024, ISBN: 9781939133243. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou (visited on 03/27/2025).

[41] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1099–1114, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

[42] T. C. Team, *SanitizerCoverage — Clang 21.0.0git documentation*, Technical Documentation, 2025. [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html (visited on 03/21/2025).

[43] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," en, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria: ACM, Oct. 2016, pp. 1032–1043, ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978428 (visited on 03/24/2025).

[44] C. Lyu *et al.*, "MOPT: Optimized Mutation Scheduling for Fuzzers," en, 2019, pp. 1949–1966, ISBN: 9781939133069. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu (visited on 03/24/2025).

[45] S. Khanna, *A Look at AFL++ Under The Hood*, en, Apr. 2023. [Online]. Available: https://blog.ritsec.club/posts/afl-under-hood/ (visited on 03/21/2025).