

Bhabani Sankar Das

Securing the Case Company’s Embedded Systems: Practical Assessment and Integration of Fuzz Testing Tools

Metropolia University of Applied Sciences Master of Engineering

Information Technology Master’s Thesis

30 September 2023

**Abstract**

Author: Bhabani Sankar Das

Title: Securing the Case Company’s Embedded Systems: Practical Assessment and Integration of Fuzz Testing Tools

Number of Pages: 87 pages + 2 appendices Date: 30 September 2023

Degree: Master of Engineering Degree Programme: Information Technology Professional Major: Networking and Services

Supervisors: First name Last name, Title (e.g.: Project Manager)

First name Last name, Title (Sami Sainio: Principal Lecturer)

In the era of pervasive digital interconnectedness, the robustness and security resilience of embedded software systems have become paramount. These systems are integral to various industries, such as automotive, aviation, and consumer electronics. As they grow in complexity, they simultaneously present an expanding cybersecurity threat landscape. This thesis focuses on enhancing a case company’s embedded software systems testing methods through fuzz testing-an automated technique that uncovers vulnerabilities by inputting invalid or unexpected data.

The motivation for this study stems from the escalating need for stronger security measures within embedded systems to counteract growing cybersecurity threats. A significant challenge is selecting an appropriate fuzzing tool from the multitude of available options. This study, therefore, adopts a practical, hands-on approach, conducted within the unique environment of a case company.

Following a literature review of various fuzzing tools, two popular fuzzing engines-American Fuzzy Lop (AFL++) and libFuzzer, were selected for the

proof-of-concept. These tools were chosen based on their reputation, widespread use, and community support. The study examines critical factors such as ease of integration, efficiency, speed, code coverage, and ability to identify vulnerabilities. This provides an impartial perspective to assist the decision-making process.

In conclusion, this study offers valuable insights into the intricacies of fuzzing, demonstrating the capabilities of tools such as AFL++ and libFuzzer. Future research could delve deeper into integration of these tools, exploring their scalability and effectiveness in CI/CD, and possibly identifying areas where new fuzzing techniques or tools might be developed for the case company.

Keywords: fuzzing, embedded system, AFL++, libFuzzer

**Preface**

A Preface is optional but highly recommended. In this page you may use informal language such as “This study was a long and hard journey for me but it is done now … finally!” Remember that after this preface you are not allowed to use informal language and do not use “I” or “You” etc. in the coming sections (see below).

“I this thesis I studied …” should be something like “This thesis includes a study …”

A Preface, which is typically about half a page long, includes: - challenges you faced while writing the thesis - things you learned - acknowledgments for those who helped you to do your thesis (colleagues, instructor, family, friends, pets ….)

Location, Date Your Name

**Contents**

[List of Figures](#_bookmark0) [List of Tables](#_bookmark1)

[List of Abbreviations](#_bookmark2) [Glossary](#_bookmark20)

1. [Introduction](#_bookmark27) 1
   1. [Background](#_bookmark29) 5
   2. [Objectives, Purpose and Scope](#_bookmark30) 6

|  |  |  |
| --- | --- | --- |
| [1.2.1](#_bookmark31) | [Objectives](#_bookmark31) | 7 |
| [1.2.2](#_bookmark32) | [Purpose](#_bookmark32) | 7 |
| [1.2.3](#_bookmark33) | [Scope](#_bookmark33) | 7 |

* 1. [Structure of the Thesis](#_bookmark34) 8

1. [Theoretical background: Software And Software Testing](#_bookmark36) 9
   1. [Computer Software](#_bookmark37) 9
   2. [Software Testing](#_bookmark39) 12
      1. [Software Testing Quality Attributes](#_bookmark40) 12
2. [Literature Review: Fuzzing as a Testing Technique](#_bookmark43) 15
   1. [Fuzzing: An Introduction, History, Successes, and Components](#_bookmark44) 15
      1. [History and Evolution of Fuzzy Testing](#_bookmark46) 17
      2. [Successes Of Fuzzing](#_bookmark47) 18
      3. [Why, How and What to Fuzz](#_bookmark49) 19
   2. [Fuzzing Process, Components, and Various Approaches](#_bookmark52) 22
      1. [Test Case Generation: From Basic to Advanced Techniques](#_bookmark53) 22
      2. [Execution and Monitoring: Coverage-Guided Fuzzing and](#_bookmark54) [Sanitizers](#_bookmark54) 23
      3. [Analysis: Discovering and Filtering Bugs](#_bookmark56) 24
      4. [Visibility in Fuzzing](#_bookmark57) 24
   3. [Implementing the Art of Fuzzing](#_bookmark59) 26
      1. [Generation and Selection of Seed](#_bookmark60) 26
      2. [A Good Fuzz Target and Validation of Inputs](#_bookmark61) 27
3. [Embedded System, Challenges, Tools And Fuzzers](#_bookmark62) 30
   1. [Embedded System](#_bookmark63) 30
      1. [Types of Embedded Devices](#_bookmark65) 31
      2. [Comparison Between Traditional and Embedded Fuzzing](#_bookmark67) 33
      3. [Challenges in Embedded Fuzzing](#_bookmark69) 34
   2. [Fuzzing Tools and Fuzzers](#_bookmark70) 36
   3. [Open-source Projects and Ongoing Research](#_bookmark72) 39
4. [Analysis: Current vs. Proposed](#_bookmark73) 42
   1. [Embedded System With Root Of Trust Architecture](#_bookmark74) 42
   2. [CI/CD for Embedded Software System](#_bookmark76) 45
   3. [Proposed: CI/CD Pipeline with Fuzzing](#_bookmark78) 48
      1. [Integration into Existing Pipeline](#_bookmark79) 48
5. [Design, Implementations and Results](#_bookmark81) 51
   1. [Implementations Using AFL++](#_bookmark82) 51
   2. [Implementations Using libFuzzer](#_bookmark86) 58
6. [Discussions and Conclusions](#_bookmark88) 64
   1. [Discussions](#_bookmark89) 64
   2. [Conclusions](#_bookmark90) 65

[References](#_bookmark91) 67

[Appendices](#_bookmark342)

[Appendix 1 AFL++ Target Program](#_bookmark343) [Appendix 2 libFuzzer Target Program](#_bookmark344)

**List of Figures**

1. [Thesis Flow Chart](#_bookmark35) 8
2. [Software Quality Perspective](#_bookmark38) 11
3. [Testing Effort vs](#_bookmark41) [Quality[93]](#_bookmark184) 13
4. [Fuzzing](#_bookmark45) [Stages[98][99]](#_bookmark190) 17
5. [Feedback Based](#_bookmark55) [Fuzzing[162][163]](#_bookmark254) 24
6. [General Process](#_bookmark58) [Fuzzing[164]](#_bookmark255) 25
7. [Embedded](#_bookmark64) [System[182]](#_bookmark273) 31
8. [Types of Embedded Devices](#_bookmark66) 32
9. [Embedded Root of Trust](#_bookmark75) [Architecture[182]](#_bookmark273) 43
10. [Embedded CI/CD flow](#_bookmark77) 46
11. [CI/CD flow With Fuzz Testing](#_bookmark80) 50
12. [AFL++ Execution Screen](#_bookmark83) 56
13. [LCOV Coverage](#_bookmark84) [Report[243]](#_bookmark334) 57
14. [afl-plot Report](#_bookmark85) 57
15. [AFL++ Execution Screen](#_bookmark87) 61

**List of Tables**

1. [Data Breaches in the 21st Century](#_bookmark28) 3
2. [Software Quality Attributes](#_bookmark42) 14
3. [Examples of Vulnerabilities Discovered by Fuzzing Tools](#_bookmark48) 19
4. [Overview of Bugs Found by Fuzzing Tools in Various Projects](#_bookmark50) 20
5. [Fuzzing Success in Embedded Projects](#_bookmark51) 21
6. [Comparison of Traditional Fuzzing and Embedded Fuzzing](#_bookmark68) 34
7. [Different Types of Fuzzers](#_bookmark71) 36

**List of Abbreviations**

**AFL:** American Fuzzy Lop.

**CI/CD:** Continuous Integration and Continuous Deployment.

**CPU:** Central Processing Unit.

**CSA:** Current State Analysis.

**DDoS:** Distributed Denial of Service. **DPA:** Differential Power Analysis. **GCC:** GNU Compiler Collection.

**HSM:** hardware security module.

**IoT:** Internet of Things.

**IP:** Intellectual Property.

**ISO:** International Organization of Standardization.

**POS:** Point Of Sale. **QEMU:** Quick Emulator. **ROT:** Root of Trust.

**RTOS:** Real Time Operating System.

**SDLC:** Software Development Lifecycle.

**SUT:** System Under Test.

**Glossary**

**edge-cases:** uncommon inputs that explore the software’s behavior at the outer limits or boundaries of what is considered normal or typical..

**Fuzz Target:** A specific function or area of a software application or system

that is intentionally subjected to fuzz testing.

**Fuzzers:** Fuzzers are software testing tools used to find security vulnerabilities and software bugs by submitting random or semi-random inputs to a program.

**Fuzzing Engine:** A tool that finds and executes inputs for a fuzz target.

**Ingenjörsfirma Anders Rundgren:** IAR Systems is a Swedish computer software company that offers development tools for embedded systems.

**Quick Emulator:** Free and open-source software product that is used to emulate a variety of computer system.

1. **Introduction**

Embedded systems are specialized computer systems that are designed to perform specific tasks or functions within a larger [system[1].They](#_bookmark92) are integral to a wide variety of devices and applications, enabling seamless functionality and efficiency in our daily lives. These systems tightly integrate hardware and software components to optimize performance, power consumption, and system [size[2].](#_bookmark93)

The importance of embedded systems lies in their ubiquity and versatility. They can be found in numerous devices and industries, including consumer [electronics[3],](#_bookmark94) [telecommunications[4],](#_bookmark95) [automotive[5],](#_bookmark96) [aerospace[6],](#_bookmark97) [medical[7],](#_bookmark98) and industrial [automation[8].](#_bookmark99) Examples of embedded systems include [microcontrollers[9]](#_bookmark100) in smart home [appliances[10],](#_bookmark101) digital signal processors in wireless communication devices[11], control systems in autonomous vehicles[11], and sensor networks for monitoring and [automation[12].](#_bookmark103)

As the world becomes increasingly interconnected and reliant on technology, the role of embedded systems continues to expand. They are essential for enabling the [Internet of Things (IoT),](#_bookmark11) as well as advancements in artificial intelligence, robotics, medical instruments, and smart [cities[13].](#_bookmark104) As of February 2023, there are an estimated 21.5 billion interconnected devices in the world, according to [Statistas[14][15].](#_bookmark106) This number is expected to continue to increase as internet consumption rises and new gadgets and machinery are introduced to the market and by 2025, it is projected that 75.44 billion [IoT](#_bookmark11) devices will be installed [worldwide[15].](#_bookmark106)

The number of embedded devices per person is significantly higher in developed countries than in developing nations. For example, in the United States, there were approximately 13 [IoT](#_bookmark11) devices per person in [2020[16].](#_bookmark107) In comparison, the global average was around 3.5 [IoT](#_bookmark11) devices per person during the same [year[17].](#_bookmark108) In developed countries, prevalent devices typically include intelligent home [appliances[18],](#_bookmark109) wearable [technology[19],](#_bookmark110) and interconnected [vehicles[20][21].](#_bookmark112)

Whereas, a greater concentration of devices aimed at supporting [agriculture[22],](#_bookmark113)

[healthcare[23],](#_bookmark114) and monitoring critical infrastructure in developing [countries[24].](#_bookmark115)

The exponential growth of embedded systems necessitates robust security, reliability, and efficiency measures to ensure the safety and functionality of the technology we rely on [daily[1].](#_bookmark92) Mirai malware attacks have demonstrated their potential for large-scale disruption, targeting devices like routers, cameras, and DVRs to execute [Distributed Denial of Service (DDoS)](#_bookmark7) [attacks[25][26].](#_bookmark117) In 2016, a [DDoS](#_bookmark7) attack on [Dyn[27],](#_bookmark118) a DNS provider, significantly hindered user access to prominent services such as Twitter, Amazon, Tumblr, Reddit, Spotify, and [Netflix[1][28].](#_bookmark119) *Ripple20*, a set of 19 critical vulnerabilities in Treck TCP/IP [library[29],](#_bookmark120) impacted IoT devices used in industrial control systems, healthcare devices, and home appliances. URGENT/1[1[30]](#_bookmark121) and [BadAlloc[31],](#_bookmark122) discovered in the VxWorks [Real Time Operating System (RT](#_bookmark17)[OS)[32],](#_bookmark123) affected millions of embedded devices, including medical devices, industrial control systems, and [routers[33].](#_bookmark124) These vulnerabilities originate from memory allocation functions and can lead to remote code execution, allowing attackers to seize control of compromised [devices[34].](#_bookmark125)

As the data breaches have become a prevalent issue in the digital age, impacting individuals, businesses, and even [governments[35].](#_bookmark126) The following t[able:1](#_bookmark28) highlights some of the most significant data breaches that have occurred in the 21st [century[36][35].](#_bookmark126) These incidents serve as a reminder of the importance of robust cybersecurity measures and the ongoing need for vigilance in protecting sensitive information.

Software [testing[37]](#_bookmark128) is a critical process in the development of embedded systems, especially as the number of embedded and software-controlled devices continues to increase. With this expansion comes a growing potential platform for illicit [activities[38].](#_bookmark129) To mitigate the risks associated with these devices, it is imperative to improve the quality and security of the software produced. While traditional testing methods, such as [manual[39]](#_bookmark130) and automated [testing[40],](#_bookmark131) have been widely used, they have limitations in uncovering all potential vulnerabilities

Table 1: Data Breaches in the 21st Century

|  |  |
| --- | --- |
| **Company** | **Description** |
| Yahoo | In 2013 and 2014, Yahoo experienced two massive data breaches that affected approxi- mately 3 billion user accounts. The breaches exposed users’ personal information, including names, email addresses, and hashed pass-  words. |
| Equifax | In 2017, Equifax, one of the largest credit re- porting agencies, suffered a breach that com- promised the personal data of approximately 147 million Americans. The breach exposed sensitive information such as Social Security  numbers, birth dates, and addresses. |
| Marriott International | In 2018, Marriott disclosed a data breach af- fecting approximately 500 million customers. The breach involved unauthorized access to the Starwood guest reservation database, ex- posing personal details, passport numbers,  and payment card information. |
| Capital One | In 2019, a hacker gained access to Capi- tal One’s systems, resulting in the exposure of personal information of approximately 106 million individuals in the United States and Canada. The breach included names, ad- dresses, credit scores, and Social Security  numbers. |
| Facebook/Cambridge Analytica | In 2018, it was revealed that the data analytics firm Cambridge Analytica harvested personal data from millions of Facebook users without their consent. The incident raised concerns about privacy and data protection on social  media platforms. |
| Target | In 2013, Target experienced a breach that af- fected approximately 41 million customer pay- ment card accounts. The attack exploited vul- nerabilities in the company’s payment system, resulting in the theft of credit and debit card in-  formation. |

and edge cases, particularly in the context of software [security[41].](#_bookmark132)

The [edge-cases](#_bookmark21) refer to scenarios or inputs that are at or beyond the outer limits or boundaries of what is considered normal or typical. These edge cases explore the software’s behavior when it encounters extreme or uncommon conditions. By

testing software with edge cases, including inputs near the upper and lower limits, unusual values, or unexpected scenarios, developers can uncover potential issues or unexpected behaviors that may not be evident during regular [testing[42][43][44].](#_bookmark135)

Fuzz testing, also known as fuzzing involves dynamically generating input data variations based on valid inputs, enabling the exploration of edge cases and the detection of bugs that are not directly related to functional [requirements[45][46][42][2].](#_bookmark93) A dynamic testing technique which feeds the random data to a program till it [crashes[47][48].](#_bookmark139) Despite its success in general software testing, fuzz testing has not been implemented in the case in company as part of their development lifecycle.

This study embarks on a journey to examine the potential benefits and implications associated with the incorporation of fuzz testing techniques into the development lifecycle of a specific case company. Operating in the semiconductor sector, this company specializes in embedded software development. Although it currently utilizes traditional testing methods, such as manual and automated testing, it has yet to explore and adopt fuzz testing within its development processes. This gap in testing practices, amidst rising cyber threats, fuels the motivation for this study.

This study aims to explore the potential benefits and implications of introducing fuzz testing into this company’s development lifecycle. The pivotal research question is: “Can the implementation of fuzz testing techniques significantly strengthen the security and quality of embedded systems developed by the case company?” The importance of this question cannot be overstated given the role of the company’s systems across numerous critical applications and industries.

To answer this research question, the study will begin with a [Current State](#_bookmark6) [Analysis (CSA)](#_bookmark6) of the company’s existing software testing practices. This [CSA](#_bookmark6) will provide a comprehensive understanding of the current testing environment and highlight potential areas that could benefit from fuzz testing.

Upon completion of the [CSA,](#_bookmark6) the study will proceed to a proof-of-concept case study using popular fuzz testing tools [AFL++[49]](#_bookmark140) and [LibFuzzer[50].](#_bookmark141) This step will

assess the effectiveness and feasibility of fuzz testing within the company’s specific development environment.

The ultimate objective of this study is to underline the potential benefits of integrating fuzz testing into the embedded software development lifecycle, thus providing valuable insights and a clear strategy for how fuzz testing can be utilized to enhance software quality and security. This study aims to serve as a practical guideline for the case company and equip them with actionable insights for incorporating fuzz testing into their software development process.

* 1. Background

The focus of this study is a company located in Helsinki, Finland that specializes in designing and developing embedded secure silicon [Intellectual Property](#_bookmark12) [solutions[51].](#_bookmark142) These solutions find extensive application across a variety of domains, including:

* + - Key Management [Solutions[52]](#_bookmark143) for securing key provisioning [management[53]](#_bookmark144) services
    - Security Protocol [Solutions[54]](#_bookmark145) for safeguarding networking [services[55]](#_bookmark146)
    - [Differential Power Analysis (DPA)](#_bookmark8) [Solutions[56][57]](#_bookmark148) for protecting against side-channel [attacks[57][58]](#_bookmark149)
    - Anti-Counterfeiting [Products[59]](#_bookmark150) and [Root of Trust](#_bookmark14) [(ROT)[60]](#_bookmark151) products, which provide highly secure [hardware[61],](#_bookmark152) software, and [firmware[62]](#_bookmark153) components

The company holds an [International Organization of Standardization](#_bookmark13) 9001 [standard[63]](#_bookmark154) certification. This certification, internationally recognized for quality management, underlines the company’s dedication to continuous improvement, a customer-centric approach, and top-level management engagement. In the context of the company’s practices, this translates to a commitment to enhancing their system testing processes and standards, which in turn, boosts the security and quality of its products.

In terms of software testing, the company utilizes an array of methodologies

including manual, integration, automation, and exploratory [testing[64].](#_bookmark155) A significant emphasis is placed on security [testing[65]](#_bookmark156) to proactively identify and mitigate potential vulnerabilities and risks. Furthermore, the company implements [Continuous Integration and Continuous Deployment (CI/CD)](#_bookmark4) practices, facilitating incremental development, integration, and testing. This approach allows for early error detection, efficient location of issues, and ultimately enhances testing ef[ficiency[66].](#_bookmark157)

In the evolving landscape of automated security testing, fuzzing has garnered substantial attention for its potential. With this in mind, the company is evaluating various fuzzing tools and techniques for its embedded products. The objective of this evaluation is to investigate the potential enhancements in the quality and coverage of their existing testing processes.

* 1. Objectives, Purpose and Scope

In the context of the rapidly growing technology landscape, the importance of robust and efficient software testing methods cannot be overstated. Embedded systems, being integral to countless applications and industries, demand meticulous testing to ensure their reliability, functionality, and most importantly, security. Traditional software testing methods, while effective to an extent, may not be sufficient to uncover all vulnerabilities in these complex systems. Therein lies the necessity for advanced testing techniques such as fuzz testing, which are designed to thoroughly probe software and uncover issues that could potentially be exploited, leading to system breakdowns or breaches.

Fuzz testing, or fuzzing, has shown remarkable potential in enhancing software quality and security, and as a result, it has garnered significant interest in the software testing field. Despite this, its application, particularly in embedded systems, remains somewhat limited, largely due to the lack of understanding and awareness about the technique.

* + 1. Objectives
       - Conduct a thorough analysis of the potential benefits and practical applications of fuzzing as an advanced software testing technique in the context of embedded systems.
       - Perform a comparative analysis of the leading fuzzing tools-AFL++ and libFuzzer, to determine the tool best suited to the company’s specific needs.
       - Carry out a proof-of-concept case study using the chosen fuzzing tool to evaluate its ability to uncover software vulnerabilities and improve system security and quality.
       - Prepare guidelines for future integration of the chosen fuzzing tool into the company’s existing [CI/CD](#_bookmark4) practices.
    2. Purpose
       - Highlight the limitations of traditional software testing methods for embedded systems and emphasize the necessity for more advanced techniques like fuzzing.
       - Demonstrate how fuzzing can augment software quality and security by uncovering vulnerabilities that may be missed by conventional testing methods.
       - Support the case company’s goal of enhancing its software testing practices by introducing a well-suited fuzzing tool into its development cycle.
    3. Scope

This study will focus on the evaluation and application of fuzzing as a software testing technique for embedded systems, within the specific context of the case company. The research will involve a comparative analysis of three leading fuzzing tools—AFL++, libFuzzer, and Atheris—and will include a proof-of-concept case study. Additionally, while this study will provide guidelines for integrating the chosen fuzzing tool into CI/CD practices, it will not cover the actual implementation or assessment of these practices.

* 1. Structure of the Thesis

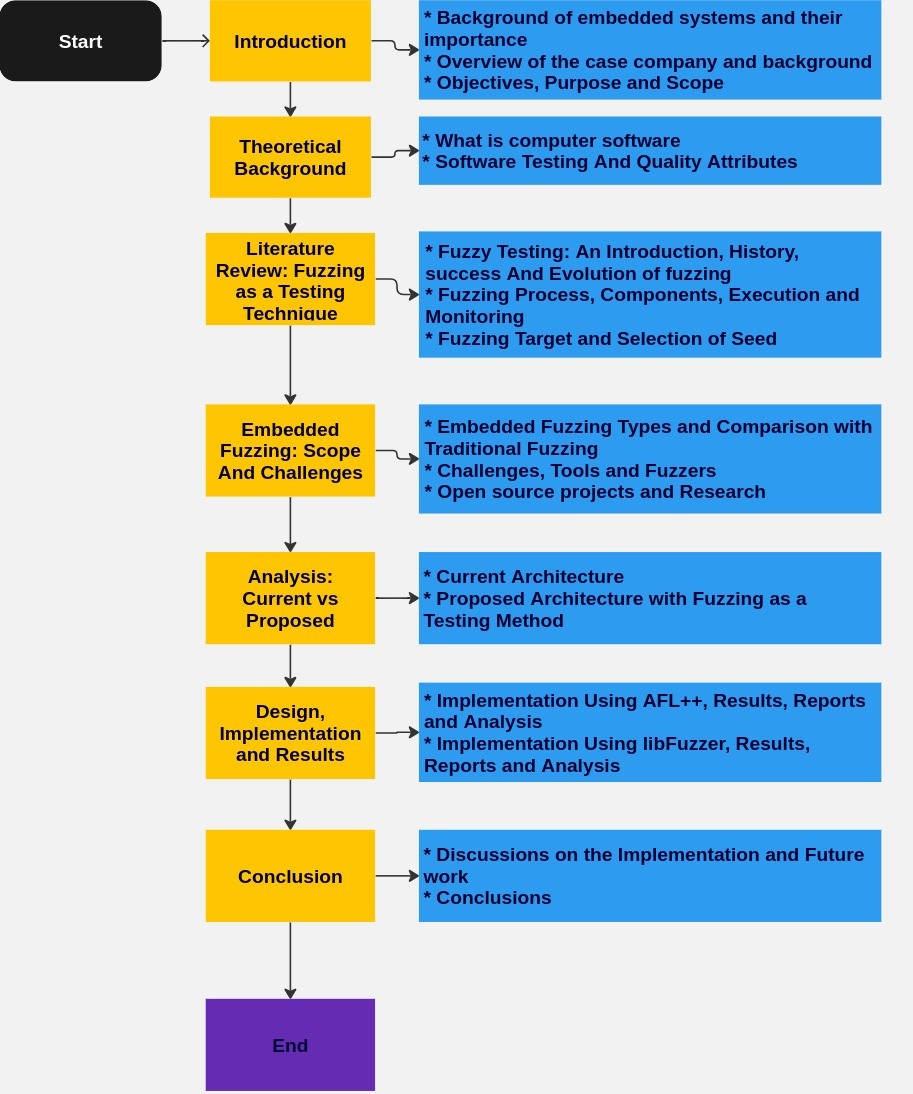


Figure 1: Thesis Flow Chart

# Theoretical background: Software And Software Testing

This chapter explores the concept of computer software and its foundational theory as it applies to software testing. The discussion commences with a comprehensive overview of computer software and its historical progression. Subsequently, the focus shifts to the crucial aspects of software testing. The discourse culminates in an exploration of software quality from the perspective of different quality attributes, highlighting their significance in the development and maintenance of high-quality software systems.

## Computer Software

Computer software, also known as programs or applications, encapsulates a set of instructions that enable interaction with a computer or digital device to accomplish a broad array of [tasks[67].](#_bookmark158) This could span operating [systems[68],](#_bookmark159) application [software[69]](#_bookmark160) such as word processors or graphic design tools, and more complex systems like databases or scientific [simulations[69].](#_bookmark160)

Software can be classified into various categories, with one crucial distinction being between system software, which governs the fundamental functions of a computer or network, and application software, which executes tasks for the [user[70][69].](#_bookmark160) Additionally, software can be classified as local, denoting that it operates on the user’s device, or web-based, meaning it functions on remote servers and is accessed via the internet. A growing amount of software is crafted for handheld devices such as smartphones and tablets, commonly referred to as mobile applications or [‘apps’[71].](#_bookmark162)

The evolution of software development dates back to the 1950s and 60s, low-level assembly languages were [prevalent[72].](#_bookmark163) However, from the 1970s onward,

high-level languages providing a greater degree of abstraction from the [hardware[61]](#_bookmark152) have come to be the norm. For instance, the C programming language, developed at Bell Labs in the early 1970s, has been extensively used

for system programming and embedded [systems[73].](#_bookmark164) Python, introduced in 1991, has gained popularity in fields like web development, data analysis, and artificial intelligence due to its emphasis on code readability and a comprehensive standard [library[74].](#_bookmark165) Programming Languages like [Swift[75]](#_bookmark166) and [Kotlin[76],](#_bookmark167) introduced in 2014 and 2011 respectively, have been officially adopted by Apple and Google for [iOS[77]](#_bookmark168) and Android mobile app [development[78].](#_bookmark169) These languages, while newer, have quickly gained traction due to their ease of use, modern features, and support from their respective tech giants.

The emergence of these languages and many others have contributed to the rich, diverse landscape of software development. The evolution and selection of programming languages continue to be driven by factors such as project requirements, efficiency, ease of use, and the ongoing growth and transformation of [technology[79].](#_bookmark170)

The integration of software into almost every facet of modern life and business operations has amplified the importance of delivering high-quality, error-free software [applications[80].](#_bookmark171) This emphasis is driven by a multitude of factors. User expectations, for instance, have become increasingly high, requiring software to not only meet functional needs but also offer seamless and intuitive [interfaces[81].](#_bookmark172) System [reliability[82]](#_bookmark173) is essential as software failures can have severe ramifications, including financial losses, compromised safety, or negative impacts on company reputation. Additionally, regulatory and compliance requirements mandate certain standards of performance and reliability, particularly in industries such as healthcare, finance, and [aviation[83].](#_bookmark174) Furthermore, in a competitive business environment, the quality of software can be a key differentiator that influences customer loyalty and market [share[84].](#_bookmark175) Despite the obvious significance of these factors, numerous software applications still experience operational failures due to inadequate adherence to rigorous software engineering processes, including critical stages such as requirements elicitation, system design, implementation, testing, and subsequent maintenance. Consequently, ensuring robust software engineering methodologies is of paramount importance in order to mitigate potential system failures and to foster the development of

robust, reliable software systems.

Software quality is a multifaceted construct encompassing various attributes that determine the degree to which a software product meets the needs and expectations of its [users[85].](#_bookmark176) These attributes can be broadly categorized into functional and non-functional aspects. Functional quality refers to how well the software performs its intended tasks, including factors such as accuracy, reliability, and correctness of [output[86].](#_bookmark177) Non-functional quality, on the other hand, relates to the overall performance and usability of the software, including aspects like efficiency, scalability, maintainability, usability, security, and [portability[87].](#_bookmark178)

Evaluating software quality is a complex task that requires comprehensive testing and analysis methodologies. It is integral to the entire [SDLC,](#_bookmark18) from initial requirements gathering to system design, implementation, and maintenance stages. Ensuring high software quality is critical, as it directly impacts user satisfaction, system reliability, and the ultimate success of the software in fulfilling its intended purpose. As such, investing in quality assurance and testing procedures is an essential aspect of software engineering, aimed at identifying and rectifying defects, improving the user experience, and assuring the delivery of robust, high-performing software [products[88].](#_bookmark179)

The [figure:2](#_bookmark38) describes different software quality perspectives and its dependencies.

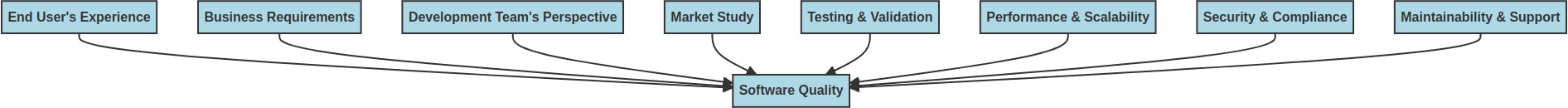


Figure 2: Software Quality Perspective

High-quality software leads to customer satisfaction, retention, and ultimately, increased profits. Jain; Sharma & Ahuja argue that “a process with high quality standards yields products with high quality standards”. To ensure that software is of high quality, testing is an essential step in the software development [process[89].](#_bookmark180)

## Software Testing

According to Whittaker, the software testing process is often misunderstood within the realm of software [development[90].](#_bookmark181) It serves as a means of evaluating and verifying a software program to determine its alignment with technical and business requirements, as well as user expectations and overall [functionality[91].](#_bookmark182) Additionally, as noted by Jamil et al., software testing is a risk-based activity that aims to uncover bugs, errors, unmet requirements, and vulnerabilities. Through this process, confidence in the behavior and requirements of the program can be [gained[92].](#_bookmark183)

Software testing is a costly endeavor, and it is impossible for any software program to be completely devoid of bugs or errors. As depicted in [figure:3](#_bookmark41) there exists a correlation between the quality of software and the effort invested in testing. Specifically, as the cost of testing increases, the quality of the software improves and the number of bugs decreases in proportion to the amount of testing conducted. The criteria for the termination of testing is contingent upon various factors, including time constraints, budget limitations, and the scope of testing.

The [figure:3](#_bookmark41) shows quality vs testing.

### 2.2.1 Software Testing Quality Attributes

Software testing methods and attributes play a crucial and integral role within the [Software Development Lifecycle,](#_bookmark18) contributing to the enhancement of quality processes and techniques. One key component of software testing is static analysis techniques, which encompass inspections and reviews, and are essential attributes in ensuring the quality of the software. These techniques serve as a complement to traditional software testing methods.

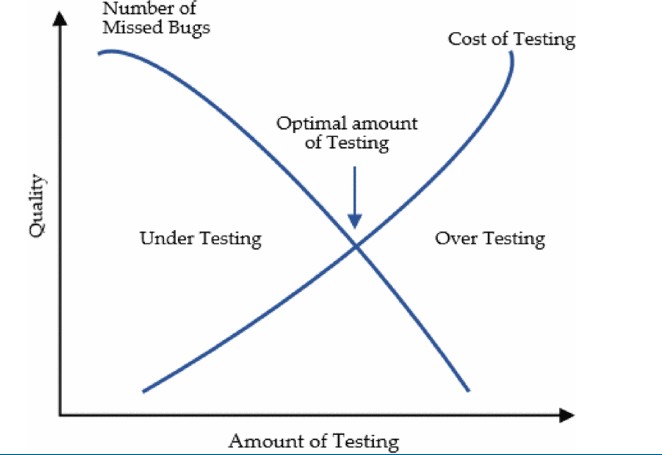


Figure 3: Testing Effort vs [Quality[93]](#_bookmark184)

##### Software Attributes

Software attributes are characteristics of software testing that assist in describing the overall quality of a software program.

Table [2](#_bookmark42) shows the attributes of software [quality[94].](#_bookmark185)

Table 2: Software Quality Attributes

##### Attribute Description

Interoperability The ability of the software to operate seamlessly in different operating systems and machines.

Simplicity The ease of use and avoidance of misuse, as well as interac- tions with other systems.

Ease of Learning The ability of the software to adapt and improve through sys- tem interactions and output logs.

Customizability The ability of the software to be configured or adjusted to meet the specific needs of a particular user or organization.

Calibrability The ability of the software to be fine-tuned for optimal perfor- mance and accuracy.

Reliability The ability of the software to perform its intended functions without failure or external influence.

Functionality The features and capabilities of the software, which are orga- nized in a modular manner to conceal information.

Portability The ability of the software to run on different platforms and hardware configurations.

Reusability The ease with which the software can be modified, adapted, and reused.

All of these attributes play a crucial role in the software development process to ensure that the software meets the desired quality standards and that it’s

user-friendly.

# Literature Review: Fuzzing as a Testing Technique

In this chapter, the focus shifts to fuzz testing as a specialized technique within the realm of software testing. The chapter starts with an introduction to Fuzzy testing. The next section describes about the history and evolution of fuzz testing, providing a solid foundation for understanding the development of this technique over time. The next section delves into the reasons behind the utilization of fuzz testing, the process of how it is performed, and the aspects of software that can be targeted for fuzzing. Following that, the working process and components of fuzzing are explored, offering a detailed explanation of the various elements involved in fuzz testing, such as the [Fuzzers,](#_bookmark23) the test harness, and the monitoring tools. Finally, the last section of this chapter concludes with success of fuzzy testing including embedded projects.

## Fuzzing: An Introduction, History, Successes, and Components

“Fuzzy testing, also known as fuzzing”, has emerged as a vital technique in discovering vulnerabilities during software testing. This approach involves the iterative and random generation of test inputs to a target program, with the objective of uncovering software bugs. During the [System Under Test (SUT)](#_bookmark19) phase, tools known as “fuzzers” continuously produce substantial quantities of valid and invalid data. These data are then fed into the target application, while the [Fuzzers](#_bookmark23) monitor and report any crashes that occur during the testing [process[95][96].](#_bookmark187)

Fuzzing has gained traction due to its capability to unearth previously unknown vulnerabilities in software that traditional testing methods might overlook.

Moreover, fuzzing can be utilized to test software across a variety of platforms and configurations, a feature especially beneficial for complex applications. The random and iterative nature of fuzzing renders it a potent tool for testing software exhibiting complex, non-linear [behavior[95].](#_bookmark186)

It is critical to understand that although fuzzing is widely recognized in vulnerability discovery, it does not replace other testing methods. Rather, it often works alongside other techniques to offer a comprehensive approach to software testing. The relevance of fuzzing is escalating as software systems grow in complexity and diversity, making it an indispensable technique to ensure the security and reliability of modern software [systems[97].](#_bookmark188)

The fuzz testing method, a systematic approach for detecting software vulnerabilities, consists of several distinct stages. First, the target application is identified, which could range from the entire application to a specific file or library within [it[98].](#_bookmark189) Second, the input data transmitted from the client to the target must be determined.

The third stage involves test case generation, which entails creating various combinations of valid and malformed data in formats such as binary or files. The fourth stage sees the fuzzers executing the target program with the inputs generated previously, stopping execution after a pre-defined timeout period. In this phase, fuzzers document any crashes or unexpected behavior.

The fifth stage, known as the analysis phase, requires the fuzzers to examine or monitor the results from the previous stage. The final stage involves determining whether any observed crashes or unexpected behavior during the target program’s execution indicate potential vulnerabilities. This stage involves verifying whether the irregular behavior stems from a legitimate security issue or a benign error. The f[igure:4](#_bookmark45) describes different stages of fuzzing.

Overall, the use of fuzzy testing provides a valuable approach to identifying potential vulnerabilities in software systems. By generating many test cases with both valid and malformed data, fuzzing techniques can expose weaknesses in software that may not be detected through traditional testing methods.

Additionally, the analysis of the results of the fuzzing process enables software developers to identify and address potential security issues before they can be exploited by malicious actors.

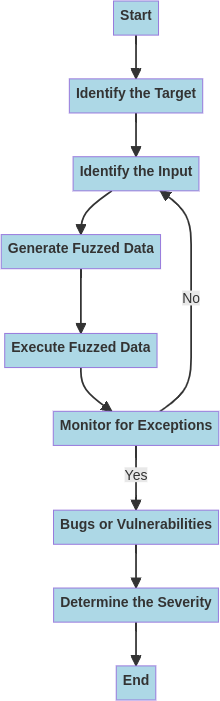


Figure 4: Fuzzing [Stages[98][99]](#_bookmark190)

### History and Evolution of Fuzzy Testing

The term “fuzzing” or “fuzz testing”,although a relatively recent addition to the lexicon of automation techniques, was first introduced by Barton Miller in 1988 as part of a class [project[100].](#_bookmark191) Initially, fuzzing was perceived as an ad-hoc or random testing method used within mission-critical [applications[101].](#_bookmark192) Over time, however, it has developed into a specialized technique for the automated generation and testing of extensive sets of input values for software [systems[102].](#_bookmark193)

The advent of the first open-source fuzz testing framework in [2007[100]](#_bookmark191) marked a significant evolution in formalizing fuzzing as an approach to software testing. This led to the creation of a myriad of additional frameworks and tools, expanding the application of fuzz testing across a diverse set of software systems.

Since its inception, fuzzing has considerably broadened its capacities. Initially, it was utilized mainly to uncover memory corruption bugs. However, as discussed

by Böhme; Cadar & Roychoudhury, its functionality has evolved over time to include the identification of a diverse range of software [vulnerabilities[102].](#_bookmark193) Takanen and Vidas; Franco; Heffner, et al. explain that the flexibility and customization capabilities of modern fuzzing frameworks enable their application across various software layers, aking fuzzing a potent tool for identifying potential weak spots in software syst[ems[100].](#_bookmark191)

Fuzzing techniques have evolved beyond mere bug detection, becoming an essential tool for vulnerability discovery and, potentially, exploitation. As indicated by Beaman et al., fuzzing can be systematically applied to uncover software vulnerabilities, which may subsequently become exploitable avenues for security [attacks[104].](#_bookmark195) The subsequent section, referenced as [Section-3.1.2,](#_bookmark47) describes the accomplishments and vulnerabilities revealed by fuzzing techniques over time.

### Successes Of Fuzzing

Fuzzy testing has been extremely effective technique since its inception in the late 1980s in discovering long-standing bugs, vulnerabilities and improving the quality and security of software. Over the years, wide range of applications shown in [Section-3.1.3,](#_bookmark49) have been successfully fuzzed and critical security vulnerabilities have been found that could have been exploited by malicious actors.

The [table:3](#_bookmark48) showcases examples of vulnerabilities discovered by fuzzing tools and affected components.

The [table:4](#_bookmark50) provides a broader overview of the number of bugs found by different fuzzing tools in various projects.

The [table:5](#_bookmark51) includes fuzzing success in embedded systems and IoT devices specifying whether the fuzzing tools are open-source or closed-source:

##### Fuzzing Tool

##### CVE ID Description and Affected Component

AFL CVE-2014-0160 Heartbleed vulnerability in OpenSSL, affecting the

TLS heartbeat [extension[105]](#_bookmark196)

AFL CVE-2014-6271 Shellshock vulnerability in the Bash shell, allowing

remote code [execution[106]](#_bookmark197)

libFuzzer CVE-2016-1839 Heap buffer overflow in ImageIO, affecting image

decoding in macOS and iOS

syzkaller CVE-2017-2636 Double fetch vulnerability in the Linux kernel, al-

lowing privilege [escalation[107]](#_bookmark198)

go-fuzz CVE-2016-3959 Denial of service vulnerability in Go’s standard li-

brary, affecting the HTTP/2 implementation

Honggfuzz CVE-2017-

15650

Heap buffer overflow in Poppler, affecting PDF ren- [dering[108]](#_bookmark199)

Table 3: Examples of Vulnerabilities Discovered by Fuzzing Tools

### Why, How and What to Fuzz

Fuzzing provides significant insights into the security, robustness, and resilience of software systems, especially those that interact with inputs from untrusted sources. While it does not make any guarantees about the reliability of inputs or transforms an untrusted source into a trusted one, it helps identify problematic inputs that could cause the software system to crash or behave [unexpectedly[102][104][103][124].](#_bookmark215)

It’s crucial to emphasize that fuzz testing doesn’t necessarily affirm the correctness of complex software programs. Instead, it contributes to the understanding of the system’s behavior under diverse and unexpected input conditions. Particularly for software playing a critical role in larger systems or applications, fuzz testing can highlight unforeseen [issues[125][126][124].](#_bookmark215)

Moreover, fuzz testing aids in assessing the software system’s stability under stress. By exposing the software to high volumes of varied inputs, this technique can identify potential performance or stability issues, arising from heavy usage or stress [conditions[126].](#_bookmark217)

##### Year Organiza-

##### tion/Re-

##### searchers

##### Fuzzing Tool/Project

##### Achievements

1999 OUSPG PROTOS Found security vulnerabilities in Voice over

IP (VoIP) [implementations[109]](#_bookmark200)

2007 Microsoft SAGE Found critical vulnerabilities in Windows

and Office products and had a remarkable impact at Microsoft[1[10]](#_bookmark201)

2010 Adobe NA Vulnerabilities uncovered by fuzzing in

Adobe Flash, Reader, and Acrobat[1[11]](#_bookmark202)

2016 ForAllSe-

cure

Mayhem Competed in DARPA’s Cyber Grand Chal-

lenge, identified, patched vulnerabilities in real-time[1[12]](#_bookmark203)

2019 Microsoft Project

OneFuzz

Used for fuzzing Azure Cloud[1[13]](#_bookmark204)

2023 Cross platform

2023 Cross-

platform

2023 Linux kernel de- velopers

2023 Go

community

2023 Cross-

platform

AFL More than 2000 bugs found in open source projects, including OpenSSL, PHP, Python, and others[1[14]](#_bookmark205)

libFuzzer More than 1000 bugs found in Chromium,

OpenSSL, LibreOffice, and other [projects[50]](#_bookmark141)

syzkaller More than 800 bugs found in the Linux ker-

nel[1[15]](#_bookmark206)

go-fuzz 200+ bugs found in Go programming lan-

guage and related projects[1[16]](#_bookmark207)

Honggfuzz Bugs found in various projects, including

Google’s Android and Chrome, and oth- ers[1[17]](#_bookmark208)

2023 Facebook Facebook’s

Sapienz

Automated testing of Android apps, leading to crash fixes and improvements[1[18]](#_bookmark209)

2023 Google ClusterFuzz More than 25000 bugs found in Google and

Chrome[1[19]](#_bookmark210)

2023 Google OSS-Fuzz 36,000+ bugs in over 550 open source

projects[1[19]](#_bookmark210)

Table 4: Overview of Bugs Found by Fuzzing Tools in Various Projects

However, it’s important to understand that fuzzing is not a panacea for all software flaws. It’s a specialized tool aimed at discovering errors and vulnerabilities that might not directly relate to the software’s requirements or intended functionalities.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Fuzzing Tool** | **Target Systems** | **Achievements** | **Source** |
| 1 | AFLNet | FTP, HTTP, and SMTP  implementations in IoT devices | Uncovered vulnerabilities and im- proved security in IoT [devices[120]](#_bookmark211) | Open- source |
| 2 | IoTcube | IoT devices including smart home appliances, medical devices, and industrial IoT devices | Identified and addressed security issues in IoT [devices[121]](#_bookmark212) | Closed- source |
| 3 | Firm- Fuzz | Firmware in IoT devices including IP cameras and routers | Discovered 7 previously undis- closed vulnerabilities across 6 dif- ferent [devices[122]](#_bookmark213) | Open- source |
| 4 | FUZZY | Zigbee implementations in IoT devices | Identified new security vulnerabili- ties in Zigbee, a widely used wire- less communication protocol for IoT [devices[103]](#_bookmark194) | Closed- source |
| 5 | Atheris | Python programs, including embedded systems using MicroPython | Detected various types of vulner- abilities, such as buffer overflows and memory leaks, in Python- based embedded [systems[123]](#_bookmark214) | Open- source |

Table 5: Fuzzing Success in Embedded Projects

While it’s effective at uncovering specific issues like memory [leaks[127],](#_bookmark218) address [corruptions[1],](#_bookmark92) and buffer [overflows[128],](#_bookmark219) it should complement other types of testing such as functional, unit, integration, and system testing, rather than replace [them[129][130][124].](#_bookmark215)

To leverage fuzzing effectively, careful planning is essential to identify specific targets for testing. Notably, identifying and defining entry points for fuzzing in large and complex applications can be challenging but essential. These entry points, including file inputs, network inputs, and command line inputs, help fuzz testers concentrate their efforts on the areas most susceptible to potential [issues[131].](#_bookmark222)

Thus, the meticulous selection of fuzz testing targets is instrumental to the test’s success, underpinning thorough testing of the most vulnerable software components.

Below given examples applications targets where fuzzing has been [successful[132].](#_bookmark223)

* Databases: [SQLite[133]](#_bookmark224)
* Text editors: [VIM[134],](#_bookmark225) OpenOffice[1[35]](#_bookmark226)
* Media codecs: audio, video, raster and vector [images[136][137]](#_bookmark228)
* Linux OS kernels, [drivers[138][139]](#_bookmark230)
* Parsers: xml, [pdf[140]](#_bookmark231)
* Crypto libraries: [OpenSSL[141],](#_bookmark232) [LibreSSL[142]](#_bookmark233)
* Browsers: [Chrome[143],](#_bookmark234) [Firefox[144]](#_bookmark235)
* Network [protocols[145],](#_bookmark236) [scanners[146]](#_bookmark237)

## Fuzzing Process, Components, and Various Approaches

Fuzzing, a dynamic software testing technique, plays a crucial role in enhancing software reliability and security by detecting concealed bugs and vulnerabilities. Its importance has grown in tandem with the increasing complexity and pervasiveness of software systems, making it a prominent field of academic and practical interest. This section provides an exhaustive exploration of the fuzzing process, including its various stages, the evolution of its techniques, and the methodologies employed.

### Test Case Generation: From Basic to Advanced Techniques

The process of fuzzing commences with the generation of numerous inputs designed to provoke the program’s failure. These inputs can take various forms, including different file formats, binary executables, or network commands [[147][102][148].](#_bookmark239) Generating broken inputs that can trigger a program to fail presents a significant challenge, which is tackled through

[mutation-based[149][150]](#_bookmark241) and generation-based [generators[151].](#_bookmark242)

Mutation-based fuzzing represents an evolution from primitive fuzzing techniques that relied heavily on generating inputs through random bytes. In contrast, mutation-based fuzzing utilizes existing input data, also known as a test corpus,

and tweaks it to create new test [data[149].](#_bookmark240) These mutation algorithms can vary significantly, with some employing stochastic modeling to concentrate mutations on specific parts of the [input[150][149].](#_bookmark240)

Generation-based generators, on the other hand, fabricate entirely new inputs for fuzzers, making them an essential tool in the fuzzing [process[96][149][140].](#_bookmark231)

### Execution and Monitoring: Coverage-Guided Fuzzing and Sanitizers

Upon generating the inputs, the test cases are fed to the program, and the fuzzing process persists until the occurrence of a program crash or hang. This stage involves monitoring crashes and exceptions using tools like [sanitizers[152][153],](#_bookmark244) which can detect specific system signals, crashes, and other violations.

##### Sanitizers

Sanitizers are integral components in the process of fuzzing. They assist in locating bugs which do not necessarily lead to a crash. However, they are resource-intensive, increasing both [CPU[154]](#_bookmark245) and [RAM[155]](#_bookmark246) usage. Therefore, usage of sanitizers should be carefully managed to avoid excessive resource utilization.

* + - * *Address Sanitizers* [*(ASAN)[156]:*](#_bookmark247) These tools are crucial in identifying vulnerabilities related to memory corruption such as use-after-free, NULL pointer dereferences, and buffer [overflows[108].](#_bookmark199)
      * *Memory Sanitizers* [*(MSAN)[157]:*](#_bookmark248) They serve to detect unauthorized read accesses to uninitialized memory, such as when a local variable is defined and read before being [set[158].](#_bookmark249)
      * *Undefined Behavior Sanitizers* [*(UBSAN)[159]:*](#_bookmark250) These sanitizers help in identifying instances of undefined behavior as per the C and C++ standards, such as when the result of an operation exceeds the permissible limit of a data type.

Coverage-guided [fuzzing[160]](#_bookmark251) is another crucial part of the execution and monitoring stage. This method involves tracking how much and which sections of the [SUT](#_bookmark19) are covered by the inputs throughout the fuzzing campaign. Compiler instrumentation, utilized by fuzzers such as [AFL/AFL++[49]](#_bookmark140) or [libFuzzer[50],](#_bookmark141) is

often employed to gather this data. External function calls are injected into specific points during the compilation process to relay data to the fuzzer each time an edge or block is [encountered[50][161][49].](#_bookmark140)

[Figure5](#_bookmark55) depicts a simple visual representation of the feedback based fuzzing.

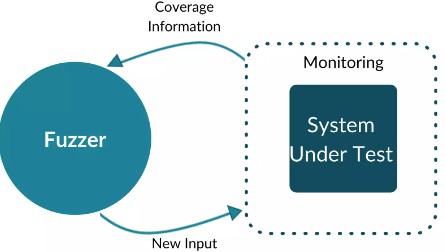


Figure 5: Feedback Based [Fuzzing[162][163]](#_bookmark254)

### Analysis: Discovering and Filtering Bugs

The analysis stage concentrates on identifying the root cause of issues detected during the monitoring phase. This stage involves both the bug [detector[164][165],](#_bookmark256) a key component in fuzzers designed to identify potential bugs, and the bug [filter[166][165][167],](#_bookmark258) which works to filter out non-security related bugs from the overall reported bugs. The bug detector collects and analyzes the stack traces from the crashes and errors that occur as a result of the test inputs, while the bug filter helps sift through these findings to prioritize security-related bugs.

### Visibility in Fuzzing

The term “Visibility” in fuzzing refers to the degree of information about the [SUT](#_bookmark19) exposed to the fuzzer’s runtime. There are three primary visibility levels in fuzzing:

[Blackbox[168][148],](#_bookmark239) [Graybox[169][96],](#_bookmark187) and [Whitebox[170][168][171]](#_bookmark262) fuzzing, each offering different degrees of access to the source code of the [SUT.](#_bookmark19)

Through the synthesis of these [techniques-mutation-based[150][149]](#_bookmark240) test case generation, [coverage-guided[160]](#_bookmark251) execution and monitoring, comprehensive bug analysis, and the consideration of visibility levels-fuzzing becomes a powerful tool for enhancing the overall software quality and security. By identifying and addressing the root cause of software bugs and vulnerabilities, developers can prevent these issues from being exploited by attackers.

The [figure-6](#_bookmark58) offers a visual representation of the fuzzing process and its dependencies.

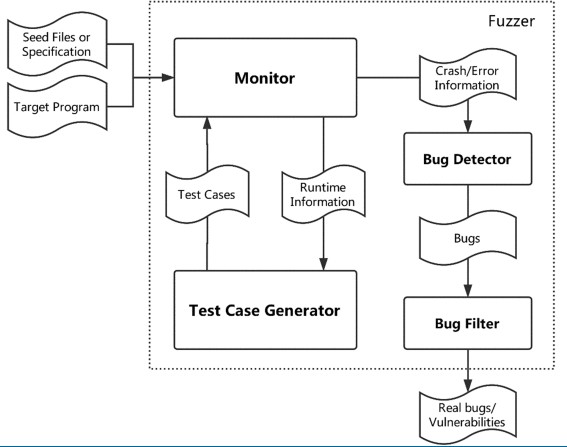


Figure 6: General Process [Fuzzing[164]](#_bookmark255)

As the target of a fuzzing test, the [SUT,](#_bookmark19) typically a binary program, is meticulously scrutinized by fuzzers. The generated inputs for the fuzzing process, derived from either mutation-based or generation-based methods, comprise a mixture of valid

and invalid inputs that pass initial validation but may still trigger bugs. The process is dynamic and continuously evolving, with ongoing research aimed at improving and automating the various stages, from input generation to bug detection and filtering.

## Implementing the Art of Fuzzing

Based on the classification of fuzzing methods delineated in section [Classification](#_bookmark52) [of Fuzzing Methods,](#_bookmark52) several questions arise when constructing and implementing fuzzing with [fuzzers[164]:](#_bookmark255)

* + - How to select and generate seeds?
    - What characteristics define a good [Fuzz Target,](#_bookmark22) and how can one validate the inputs?
    - What considerations should be taken into account when dealing with test cases causing crashes?
    - How to effectively utilize information during runtime?
    - How to achieve scalability improvements?

The responses to these queries determine the effectiveness of fuzzing in revealing hidden vulnerabilities, thereby enhancing the security and reliability of the target system.

### Generation and Selection of Seed

Seed corpus in fuzz testing refers to a collection of representative input files that the target program is designed to process. The seed corpus selection can significantly influence the efficacy and efficiency of the fuzzing process, emphasizing its importance for maximizing test [coverage[172].](#_bookmark263)

In an ideal scenario, a fuzzer aims for 100% coverage, seeking to activate and examine as many code paths as possible within the target program. However, this is rarely achieved due to the complex and large-scale nature of contemporary software. Nonetheless, striving for high coverage significantly improves the chances of uncovering hidden bugs and vulnerabilities, enhancing the evaluation

of system reliability and security[110].

The size of the seed corpus must be chosen with careful consideration of its potential effect on coverage and computational resources. Larger seed files may provide more coverage but consume more resources and time. If smaller seed files can provide similar coverage to larger ones, they are typically preferred for efficient [testing[164][173].](#_bookmark264)

The initial seed corpus is particularly crucial when using a mutation-based fuzzer, which requires an initial set of inputs or seeds. The quality of these seeds substantially influences the fuzzer’s capacity to explore the program’s state space and detect [bugs[149].](#_bookmark240) An efficient way of enhancing code coverage, proposed by Kim; Choi & Lee, involves a detailed analysis of binary file fields during fuzzing.

This approach includes tracking and evaluating stack frames, assembly codes, and registers to unearth potential new inputs that could augment [coverage[97].](#_bookmark188)

### A Good Fuzz Target and Validation of Inputs

In the context of fuzz testing, the ‘fuzz target’ is the item being tested. This can be anything from a command line tool to a hardware device. The quality of the fuzz target is crucial as it should accurately represent the behavior of the real-world system or application being [tested[174].](#_bookmark265)

##### A Good Fuzz Target

Here is an example of a fuzz target written in C with LLVM [libFuzzer[50]:](#_bookmark141)

1

2 **extern** "C" **int** LLVMFuzzerTestOneInput (**const uint8\_t** \*Data, **size\_t**

Size) {

3 DoSomethingInterestingWithMyAPI(Data, Size);

#### 4 **return** 0; // Values other than 0 and -1 are reserved for future use.

5 }

6

In this case, the function *LLVMFuzzerTestOneInput* takes two arguments, the address of the data to store and its size. The API of the fuzz target is called in subsequent steps of fuzzing. When creating a fuzz target, the following aspects should be taken into [account[50][49]:](#_bookmark140)

* The [Fuzzing Engine](#_bookmark24) should be designed to execute the fuzz target multiple times within the same process to enhance efficiency.
* The fuzz target should handle a wide range of inputs, from valid to malformed and even empty, to increase robustness.
* Any abrupt halt or unexpected exit from the fuzz target usually signifies a bug within the system and warrants closer examination.
* For faster testing, the fuzz target’s execution speed should be optimized. Moreover, it should consume minimal memory to avoid out-of-memory errors.
* If the fuzz target relies on a global state, it should be designed to avoid altering it.
* Regular testing should be encouraged by integrating the fuzz target into a continuous integration system.
* The fuzz target should be deterministic to ensure consistent results.
* It is crucial to optimize the performance of the fuzz target as the fuzzing process requires multiple iterations.

##### Isolating Crashes and Test Cases

Fuzzing often results in many system crashes, which can be challenging and time-consuming to analyze. Therefore, due to time and budget constraints, it’s important to isolate and filter crash-inducing test cases effectively. A

ranking-based method to categorize these cases based on their bug-inducing capabilities and distinctiveness was proposed by Chen et [al.[167].](#_bookmark258)

Crashes can also be isolated through clustering methods, differentiating crashes based on uniqueness and debug information. For example, AFL uses the absence of a pre-existing execution path to define the uniqueness of a [crash[175].](#_bookmark266)

##### Runtime Information During Fuzzing

Advanced fuzzing techniques, such as symbolic execution and dynamic analysis, are used to gather runtime information, such as code coverage and data flow.

However, their efficiency can be limited due to challenges like path explosion, a phenomenon caused by the multiple execution paths that stem from a single conditional branch in the target program.

Godefroid suggested using ‘function summaries’ for lower-level functions to reduce the number of execution paths. This allows higher-level functions to reuse [them[176].](#_bookmark267) Additionally, heuristic search algorithms like random path selection and automatic partial loop summarization help identify the most relevant [paths[164].](#_bookmark255)

In summary, effective fuzzing implementation requires careful consideration of various factors. The seed selection and generation, quality of the fuzz target, test case management, use of runtime information, and scalability all play significant roles in the process. Employing advanced techniques and maintaining a clear focus on these aspects can optimize the fuzzing process and enhance its ability to uncover hidden bugs and vulnerabilities in the target system.

# Embedded System, Challenges, Tools And Fuzzers

In this chapter, the critical role of fuzzing is illuminated as a mechanism to enhance the security and dependability of embedded systems. An initial comparative exploration is presented, distinguishing conventional fuzzing techniques and the unique challenges introduced when implementing fuzzing within the context of embedded systems. This is followed by an extensive examination of a selection of widely-used fuzzing tools and fuzzers, assessing their operational competencies, user accessibility, efficacy, and potential limitations. Key tools encompassed within this analysis include [AFL++[49],](#_bookmark140) [libFuzzer[50],](#_bookmark141) [ClusterFuzz[177],](#_bookmark268) [OSS-Fuzz[178],](#_bookmark269) FuzzT[est[179],](#_bookmark270) [Atheris[123],](#_bookmark214) and [Hypothesis[180].](#_bookmark271) The chapter culminates with a meticulous survey of relevant open-source initiatives and current research trajectories within the field of fuzzing.

## Embedded System

Embedded systems, specialized computer systems engineered for specific tasks, diverge significantly from conventional, multi-purpose computers. These integrated systems, composed of hardware, software, and occasionally mechanical components, are designed to perform specific functions either autonomously or within a larger system. In contrast to general-purpose computers designed for multitasking, embedded systems typically focus on executing a single task or a related group of [tasks[12][48][181].](#_bookmark272)

The [figure:7](#_bookmark64) shows the different components of embedded systems.

Just like conventional computer systems, embedded systems’s core components usually include a Central Processing Unit [(CPU)[154],](#_bookmark245) Random Access Memory (RAM), and input/output (I/O) interfaces. However, embedded systems often lack user-friendly interfaces, and instead might feature a minimal user interface or none at [all[183].](#_bookmark274)

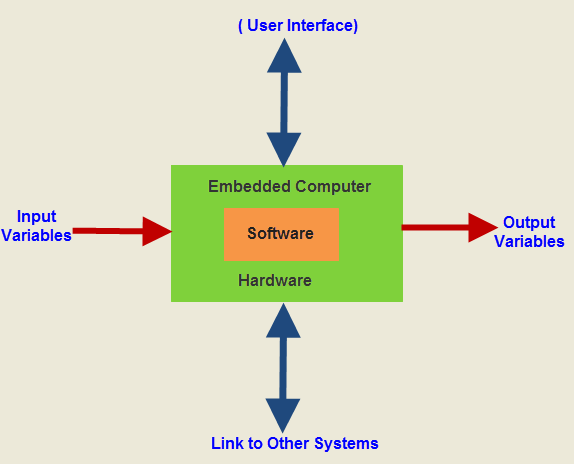


Figure 7: Embedded [System[182]](#_bookmark273)

Reflecting on the prior discussions of embedded systems and fuzzing in sections [1](#_bookmark27) and [3](#_bookmark43) respectively, this chapter explores further the significance and methodology of fuzzing within the domain of embedded systems.

### Types of Embedded Devices

Embedded systems can be classified into several types, such as standalone, real-time, networked, and mobile [systems[184].](#_bookmark275) These systems play essential roles in various sectors, from consumer [electronics[3]](#_bookmark94) like digital [watches[185],](#_bookmark276) televisions, and cameras, to industrial [machinery[8],](#_bookmark99) medical [equipment[7],](#_bookmark98) and [automobiles[5][186][183].](#_bookmark274)

The [figure:8](#_bookmark66) shows the different types of embedded devices.

##### Standalone Devices:

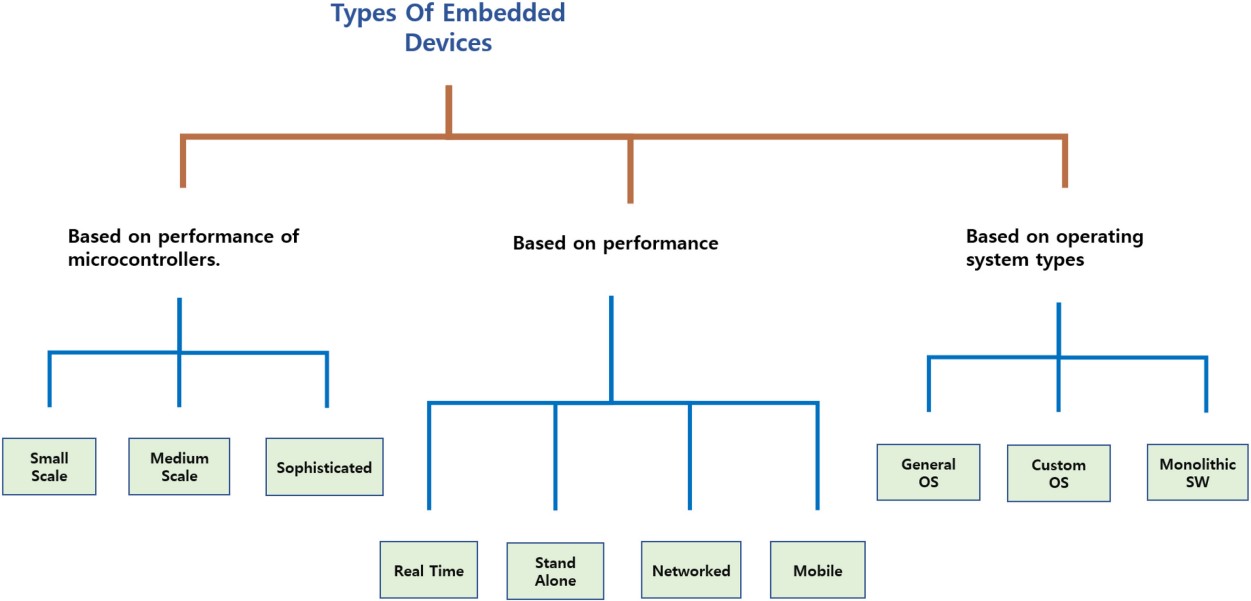


Figure 8: Types of Embedded Devices

Standalone devices constitute a subset of embedded systems functioning independently, without the necessity for a host system. They process input data within their own interfaces, producing outputs to control or drive attached devices or to generate displays. These devices have found broad applicability in various appliances due to their flexibility and efficiency. They often employ lightweight user environments for optimal performance [[48][187][188].](#_bookmark279)

**Real-time Embedded Systems:** Real-time embedded systems deliver outputs within specific time constraints, crucial for timely project completion. These systems interact with external environments via interfaces and often utilize sensors. They typically fall into two categories: soft systems, which prioritize process completion, and hard systems, which emphasize strict adherence to time constraints. Real-time operating systems (RTOS) manage hardware resources and application software to ensure task completion with precision and [consistency[189][48][187][188].](#_bookmark279)

**Networked embedded systems:** Networked embedded systems form key components of wired or wireless networks, heavily relying on web server communication. Ubiquitously found in security systems, ATMs, and [Point Of](#_bookmark15) [Sale[190]](#_bookmark281) systems, these systems manage a network of devices or workstations to perform various functions. If an embedded system forms part of or depends on a device network, it is classified as a networked embedded system

[[189][48][187][188].](#_bookmark279)

In conclusion, the realm of embedded systems is vast and intricate, encompassing a wide variety of devices and applications. A consistent requirement across these diverse systems is the need for rigorous testing and security measures, a task significantly streamlined by the application of fuzzing techniques. This chapter offers an in-depth exploration of this compelling subject, spotlighting the unique challenges and solutions in embedded [fuzzing[2].](#_bookmark93) The following sections will discuss the specific methodologies used in fuzzing for embedded systems, the challenges encountered, and potential strategies to overcome them.

### Comparison Between Traditional and Embedded Fuzzing

The [table:6](#_bookmark68) provides the comparison between traditional and embedded system [fuzzing[48][99].](#_bookmark190)

In addition to the general differences mentioned in the above table, there are more aspects to consider when doing the comparison.

Beyond the primary distinctions outlined in the table, additional considerations arise when comparing traditional and embedded fuzzing techniques. Traditional fuzzing tools typically offer a more straightforward setup than their embedded counterparts, which may necessitate extensive manual configuration due to the wide range of hardware and software components involved. Furthermore, integration of traditional fuzzing into [Continuous Integration and Continuous](#_bookmark4) [Deployment](#_bookmark4) systems tends to be more straightforward, while embedded fuzzing often requires custom solutions to accommodate hardware specificities. Feedback in traditional fuzzing is typically derived directly from the target application or library, simplifying the process of monitoring fuzzing operations and identifying anomalies. In contrast, embedded fuzzing might garner feedback from a multitude of sources, encompassing firmware and hardware components of the target device, which can complicate monitoring and issue [detection[48][187].](#_bookmark278)

##### Traditional Fuzzing Embedded Fuzzing

##### Target

##### Systems

Desktop and server applica- tions, web applications, and li- braries.

Real-time operating systems (RTOS), IoT devices, firmware.

##### Diverse Targets

Server applications. Specific hardware and proto- cols with limited resources and real-time constraints.

**Challenges** Input generation and code cov-

erage.

Compatibility with specific hardware and operating sys- tems.

##### Reliability Constraints

##### Testing

##### Environment

No strict time limitations, allow- ing comprehensive and thor- ough fuzzing tests to enhance reliability of findings.

Simulated or actual systems with standard operating sys- tems.

Operates under real-time con- straints, demanding efficient and timely fuzzing tests. The reduced testing time could po- tentially affect the thorough- ness and thus the reliability of the findings.

Emulators, simulators, or ac- tual devices with specific hard- ware and protocols.

##### Physical Interaction

##### Limited

##### Resources

Not a primary concern. Required to consider environ-

ment and physical states of the system.

Not a primary concern. Must optimize fuzzing tools to

work efficiently within limited resources.

**Scalability** Can be easily parallelized to

scale with the number of target applications or libraries.

Physical access to the target devices or emulators can limit scalability.

##### Tools and Techniques

AFL, libFuzzer, Honggfuzz. FirmFuzz, IoTcube, FUZZY.

Table 6: Comparison of Traditional Fuzzing and Embedded Fuzzing

### Challenges in Embedded Fuzzing

Fuzz testing embedded systems presents unique challenges necessitating the development of specialized tools and strategies to maintain system security and reliability. This subsection discusses the primary concerns that researchers and practitioners encounter while fuzz testing embedded systems, emphasizing the need for inventive approaches to effectively address these issues.

Yun et al. identifies the following primary challenges:

* + - * **Heterogeneous targets:** The variability in hardware, operating systems, and communication protocols in embedded systems presents a significant [challenge[48].](#_bookmark139)
      * **Limited resources:** The restricted processing power and memory typically characteristic of embedded systems can limit the applicability of traditional fuzzing [techniques[48].](#_bookmark139)
      * **Real-time and reliability constraints:** Some embedded systems have strict real-time requirements, and fuzzing must not interfere with their regular [operations[48].](#_bookmark139)
      * **Scalability issues:** Fuzzing often requires physical access to target devices or emulators, which can impede [scalability[48].](#_bookmark139)

Muench et al. highlights further obstacles:

* + - * **Limited visibility:** The internal state of embedded systems is often difficult to observe, complicating the interpretation of fuzz testing impacts and issue [identification[1].](#_bookmark92)
      * **Reproduction of crashes:** The diverse nature of embedded systems and the use of custom hardware may hinder the consistent and accurate reproduction of [crashes[1].](#_bookmark92)
      * **Root cause analysis:** There exist difficulties in discerning the root cause of crashes in embedded [systems[1].](#_bookmark92)

Eisele et al. enumerates additional complexities:

* + - * **Physical interaction:** Embedded systems often interact with the physical world, further complicating fuzzing [tasks[2].](#_bookmark93)
      * **Device-specific hardware and protocols:** Custom hardware components and communication protocols in embedded devices may not be compatible with existing fuzzing tools and [techniques[2].](#_bookmark93)
      * **Lack of source code:** Absence of source code for embedded systems can hinder software analysis for potential vulnerabilities and hamper the development of appropriate fuzzing test [cases[2].](#_bookmark93)

Manès et al. discusses the following challenges:

* + - * **Stateful applications:** Fuzzing stateful applications, such as network protocols or multi-threaded applications, necessitates effective handling of state transitions, synchronization, and concurrent [execution[148].](#_bookmark239)
      * **Oracle problem:** Identifying whether a specific input has triggered a vulnerability can pose challenges. Fuzzers often depend on simple

oracles like crashes, which may not be sufficient for detecting all types of [vulnerabilities[148].](#_bookmark239)

* + - * **Performance:** The efficiency of fuzzers is critical for their efficacy. Slow fuzzers may not explore sufficient execution paths within a reasonable timeframe, thus failing to uncover [vulnerabilities[148].](#_bookmark239)

## Fuzzing Tools and Fuzzers

Fuzzing embedded systems presents unique challenges and requirements, necessitating the development of dedicated tools and techniques. This section provides an overview of several fuzzing tools, their functionality, ease of use, and their advantages and limitations.

Table 7: Different Types of Fuzzers

##### Name Functionality Ease of use Effectiveness and Limita-

##### tions

[AFL[191][175]](#_bookmark266) Coverage-

guided, genetic algorithms, bi- nary program fuzzing

Simple setup, easy to use

Widely adopted, has dis- covered numerous vulner- abilities. Lacks support for multi-threaded applica- tions, and limited effective- ness in fuzzing structured data formats

[libFuzzer[50]](#_bookmark141) Coverage-

guided, in- process fuzzing, targets li-

braries with well-defined APIs

Easy integration with LLVM- based projects, requires writing custom harness code

Has found vulnerabilities in widely-used libraries and software components. In- process fuzzing can re- sult in performance bottle- necks, limited support for non-LLVM compilers

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Functionality** | **Ease of use** | **Effectiveness and Limita- tions** |
| hongg- | Security- | Moderate setup | Effective in identifying vul- |
| [fuzz[192]](#_bookmark283) | oriented, sup- | complexity, flexi- | nerabilities in various types |
|  | ports multiple | ble configuration | of software. Lacks the ease |
|  | platforms and | options | of use and setup simplic- |
|  | feedback sig- |  | ity compared to AFL, limited |
|  | nals |  | documentation |
| [Boofuzz[193]](#_bookmark284) | Network pro- | Moderate setup | Effective in finding vulnera- |
|  | tocol fuzzing, | complexity, flexi- | bilities in network protocols |
|  | custom protocol | ble configuration | and services. Limited sup- |
|  | specifications | options | port for non-network proto- |
|  |  |  | col targets, can be complex |
|  |  |  | to configure for custom pro- |
|  |  |  | tocols |
| [Radamsa[194]](#_bookmark285) | Black-box | Simple to use | Has identified vulnerabili- |
|  | fuzzer for file | for black-box | ties in a wide range of |
|  | formats, net- | testing scenar- | software. Lacks feedback- |
|  | work protocols, | ios | driven fuzzing capabilities, |
|  | and command- |  | limited in identifying com- |
|  | line utilities |  | plex vulnerabilities |
| [AFL++[49]](#_bookmark140) | Enhanced ver- | Similar to AFL, | Inherits AFL’s effectiveness |
|  | sion of AFL with | simple setup, | and adds various improve- |
|  | improved muta- | and easy to use | ments. Still inherits some |
|  | tion strategies | for most users | limitations of AFL, such as |
|  | and perfor- |  | the lack of support for multi- |
|  | mance |  | threaded applications and |
|  |  |  | structured data formats |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Functionality** | **Ease of use** | **Effectiveness and Limita- tions** |
| Firm- | Automated IoT | Requires un- | Useful for identifying vul- |
| [Fuzz[122][48]](#_bookmark139) | firmware intro- | derstanding of | nerabilities in firmware im- |
|  | spection and | firmware images | ages. Limited to firmware |
|  | analysis | and emulation | images, requires specific |
|  |  |  | knowledge and expertise in |
|  |  |  | firmware analysis |
| Firma- | Emulation | Moderate | Effective in analyzing |
| [dyne[195]](#_bookmark286) | and dynamic | setup com- | firmware for potential vul- |
|  | analysis of | plexity, requires | nerabilities. Limited to |
|  | Linux-based | knowledge of | Linux-based firmware, |
|  | embedded | firmware and | emulation may not accu- |
|  | firmware | Linux systems | rately represent the actual |
|  |  |  | hardware environment |
| AVAT[AR[196](#_bookmark287)] | Framework | Requires un- | Can be effective when |
|  | for dynamic | derstanding | combined with fuzzers |
|  | analysis and | of embedded | for analyzing embedded |
|  | instrumentation | systems and | firmware. Steep learning |
|  | of embedded | experience with | curve, requires customiza- |
|  | systems | dynamic analy- | tion for specific target |
|  |  | sis | systems |
| [Atheris[197]](#_bookmark288) | Coverage- | Easy to use, | Effective in identifying vul- |
|  | guided, native | Python script | nerabilities in Python and |
|  | and Python | fuzzing | native extensions. Re- |
|  | fuzzer with in- |  | quires Python 3.3 or later, |
|  | tegration with |  | and it is mostly tested on |
|  | libFuzzer |  | Linux |

## Open-source Projects and Ongoing Research

This section examines open-source projects and ongoing research in the realm of fuzzing. The effectiveness of security testing in revealing vulnerabilities in software systems has seen an increased spotlight in recent years due to the discovery of high-profile bugs. The [Heartbleed[198]](#_bookmark289) bug, discovered in OpenSSL in 2014, and the Shellshock [bug[106],](#_bookmark197) found in the Bash shell in the same year, are prime examples of serious vulnerabilities that have had wide-ranging impacts on the internet. Although not found through fuzzing, their discovery highlighted the critical importance of rigorous security testing. [Section-3.1.2,](#_bookmark47) refers the accomplishments and vulnerabilities revealed by fuzzing techniques over time.

Highlighted in this discussion are several high-profile open-source fuzzing initiatives, including Google’s [OSS-Fuzz[178],](#_bookmark269) an assortment of AFL-based [fuzzers[49],](#_bookmark140) and a GitHub repository committed to research in fuzzing. The overarching goal of these projects is to push the boundaries of what is currently possible in fuzzing. This is achieved by providing a platform that fosters collaboration and facilitates knowledge sharing among members of the fuzzing community.

The overview of these open-source projects and the active research in fuzzing presented in this section is intended to provide invaluable insights into the current trends and potential future trajectories of fuzzing as an indispensable tool in software security testing.

##### [OSS-Fuzz:[178]](#_bookmark269)

Google’s OSS-Fuzz represents a pioneering initiative in the domain of open-source fuzzing. Its uniqueness lies not merely in its goal of enhancing

security and stability—which is indeed a common objective across all fuzzing projects—but in its commitment to continuous, automated fuzzing for selected open-source software projects.

OSS-Fuzz stands out in its adaptability, integrating with a variety of fuzzing

engines such as [libFuzzer[50]](#_bookmark141) and [AFL++[49].](#_bookmark140) Furthermore, it supports multiple programming languages, demonstrating its versatility in accommodating projects written in C, C++, Rust, and Go. This broad compatibility reinforces its value to the diverse landscape of open-source [software[199].](#_bookmark290)

##### OneFuzz:[1[13]](#_bookmark204)

OneFuzz is a fuzzing-as-a-service platform developed by Microsoft, that aims to automate the detection and reporting of software bugs. It is a self-hosted platform that provides a range of features, including task scheduling, crash analysis, and distributed fuzzing. OneFuzz can be integrated with existing development workflows, making it a useful tool for organizations looking to improve the security and stability of their software systems

##### Afl Based Fuzzers:

[American Fuzzy Lop](#_bookmark3) is a coverage-guided, or feedback-based, fuzzer that utilizes a dynamic approach to identify potential vulnerabilities in software applications. It modifies the target executable to measure and optimize code coverage, mutating input data in a manner that maximizes the coverage achieved. This process is iteratively repeated, seeking to uncover instances in which the program crashes, thereby identifying potential security vulnerabilities. AFL has proven to be highly effective in practice, as evidenced by its extensive usage and success in uncovering numerous vulnerabilities in widely-used software. Furthermore, AFL is renowned for its ease of use, making it a popular choice among security researchers and practitioners[1[14][125].](#_bookmark216)

Several fuzzers have been developed based on AFL’s architecture, enhancing its capabilities or tailoring it for specific use cases. Some notable AFL-based fuzzers mentioned below:

##### [AFL++:[49][200]](#_bookmark291)

An enhanced version of AFL which introduces various performance improvements and new features, such as improved mutation strategies, faster execution, more and better mutations, improved instrumentation, and custom module support.

##### [AFLFast:[201]](#_bookmark292)

A fuzzer that introduces a power schedule, aiming to prioritize seeds that are more likely to explore new paths and increase coverage. By doing so, AFLFast aims to improve the efficiency of fuzzing and maximize the code coverage in a shorter time [frame[161].](#_bookmark252)

##### T[riforceLinuxSyscallFuzzer:[202]](#_bookmark293)

This fuzzing technique specifically targets Linux system calls, aiming to identify potential vulnerabilities in the system call interface.

##### [AFL-unicorn:[203]](#_bookmark294)

AFL-Unicorn represents a noteworthy extension of the American Fuzzy Lop (AFL) framework, characterized by its versatility across various architectures, data input formats, and binary formats—including the [firmware[62]](#_bookmark153) of bare-metal devices such as Wi-Fi components and [baseband[204]](#_bookmark295) processors in mobile phones. This utility is designed to examine binary files and then create a fuzzer that emulates the state at the entry point of the parsing routine.

The efficacy of AFL-Unicorn is underpinned by the successful integration of AFL’s coverage-guided fuzzing technique and [the[205]](#_bookmark296) engine. This synergistic combination facilitates the exploration of multiple execution paths within the targeted binary, including those within proprietary, closed-source firmware like that running on baseband processors. This capability contributes to a thorough and comprehensive fuzzing process, even in the context of complex and traditionally opaque [systems[206].](#_bookmark297)

# Analysis: Current vs. Proposed

This chapter delineates a comprehensive study of the existing CI/CD pipeline employed for embedded systems in the designated case company. The focal point of the current analysis is the integration of diverse tools with the Zephyr RTOS. A subsequent proposal for the pipeline’s augmentation, incorporating fuzzing, is presented with the aim of uncovering and mitigating potential vulnerabilities, thereby enhancing the security and dependability of the software solutions devised.

## Embedded System With Root Of Trust Architecture

Reflecting on the prior discussions of embedded systems in [sections1](#_bookmark27) [and4](#_bookmark62) respectively, this chapter explores current state analysis of the case in company and proposed state of analysis with respect to fuzzing.

The [figure:9](#_bookmark75) illustrates high-level depiction of the architecture for a secure embedded system that leverages a Root of Trust (RoT) used in the company.

In an increasingly digital and interconnected world, the security of embedded systems is paramount. Embedded systems, ranging from tiny [IoT](#_bookmark11) devices to

large-scale industrial control systems, form the backbone of modern infrastructure. However, their pervasiveness also makes them a tempting target for attackers.

Vulnerabilities such as [Meltdown[207]](#_bookmark298) and [Spectre[208]](#_bookmark299) are some recent examples. From data theft to device manipulation, the potential implications of a successful attack are dire. Hence, there’s a necessity for a robust and secure foundation upon which these systems operate. This is where a secure architecture, underpinned by a hardware [ROT,](#_bookmark14) becomes [critical[209].](#_bookmark300)

**Hardware:** This represents the physical components and sub-systems compromise [Central Processing Unit (CPU),](#_bookmark5) [memory[210]](#_bookmark301) and other peripherals. It acts as a base layer on which rest of the system operates.

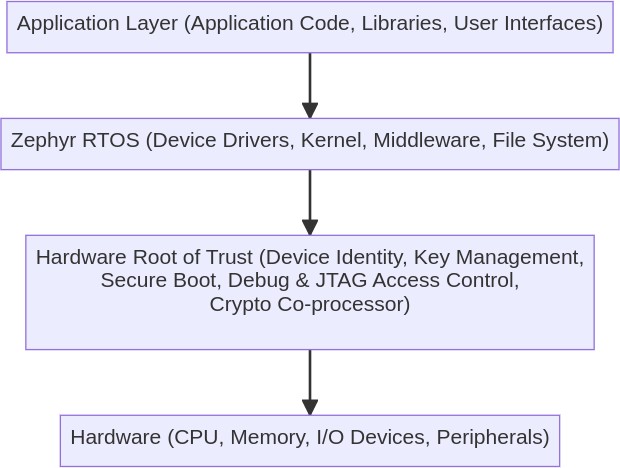


Figure 9: Embedded Root of Trust [Architecture[182]](#_bookmark273)

**Hardware Root of Trust:** Roots of Trust refer to secure components within a computer system, encompassing hardware, software, or [firmware[62],](#_bookmark153) that carry out crucial security operations like data encryption, certificate validation and key management[211]. This is a principle that initiates a trust sequence, which is essential for verifying that computers start up using authentic code. They serve as the foundational elements upon which the security of other system components is established. Given their critical role, these elements must be designed with a high level of [security[60][209][212][21](#_bookmark303)1].

*Hardware Root of Trust* ensures the integrity of the lower level system operations such as secure [boot[213],](#_bookmark304) Debug and JTAG access control, Key management and Crypto Co-processors. The *secure boot* is a mechanism that the system uses to ensure that it “boots” in a known secure state. The [ROT](#_bookmark14) plays a critical role in this process, verifying the integrity of the boot firmware and subsequently loaded software[21[1][213].](#_bookmark304)

The *debug and JTAG access control* needs careful handling as it can be used maliciously although an essential feature for debugging and troubleshooting

[purpose[214][215].](#_bookmark306)

The *device identity and key management* component helps in ensuring the use, storage and generation of cryptographic keys. A unique identification is assigned to each device in the system for authentication. Key management typically involves a [hardware security module (HSM)](#_bookmark10) for storage of the [keys[52].](#_bookmark143) The keys are used for different operations such as [encryption[216],](#_bookmark307) signing, and [authentication[217].](#_bookmark308)

The *crypto core* is a component used in the [ROT](#_bookmark14) implementations designed to be resistant to attack. This usually isolated from rest of the devices, helps to protect the crypto core from being compromised. Some of the common cryptographic functions are [AES[218],](#_bookmark309) [RSA[219],](#_bookmark310) and [ECC[220].](#_bookmark311)

**Zephyr RT**[**OS[221][222][223]**](#_bookmark314) serves as a vital bridge between the hardware and application layers in an embedded system. This compact, open-source Real-Time Operating System (RTOS) is thoughtfully engineered for devices with resource limitations. Zephyr RTOS plays a critical role in facilitating the Root of Trust (RoT) functionalities, such as secure boot, cryptographic procedures, and device [validation[221][222][223].](#_bookmark314)

Key elements within the Zephyr RTOS include:

1. Device Drivers: The device drivers play a significant role in bringing the hardware components, necessary for the Root of Trust, into operation. These hardware components could be the cryptographic engine or the secure boot bootloader. Besides initialization, these drivers make it possible for the kernel to manage and command these devices.
2. [Kernel[224]:](#_bookmark315) The kernel acts as the system’s core, ensuring that only verified software is allowed to boot up on the device. Its role is critical in maintaining the security condition of the device, by managing processes, task scheduling, memory, and inter-process communication.
3. Middleware: This component offers a range of security services including secure storage solutions for cryptographic keys and secure transmission protocols. It essentially forms a bridge between applications and the underlying network services.
4. File System: The file system is employed to safeguard the security configuration of the device, along with other classified information. Its main function is to prevent unauthorized access or tampering with this sensitive data.

Together, these components within the Zephyr RTOS form the backbone of a secure, robust RoT, offering a trustworthy platform for all software operations within the system.

**Application Layer** includes the end-user applications, libraries, and user interfaces. It interacts with the Zephyr RTOS to perform its operations. It provides the important security features for the users.

## CI/CD for Embedded Software System

In the context of the case in company, an integrated [CI/CD](#_bookmark4) pipeline paired with the Zephyr RTOS enhances the software development process. This integration leads to faster firmware updates, better code quality, and more efficient development cycles.

Embedded systems, with their direct connections to hardware and real-world interactions, require a specialized development approach. A robust [CI/CD](#_bookmark4) pipeline is essential for ensuring the reliability and efficiency of the software. This section provides a detailed overview of the key stages in a [CI/CD](#_bookmark4) flow.

##### Development Environment and Version Control:

Using a version-controlled codebase like [Git[225]](#_bookmark316) ensures an organized progression in the software development lifecycle. [Gerrit[226],](#_bookmark317) used for hosting repositories, prompts a rigorous review for every code change, which contributes to improved code consistency.

The [figure:10](#_bookmark77) illustrates IT Infrastructure for CI/CD flow for the case in company.

##### Compilation Builds with GCC and IAR:

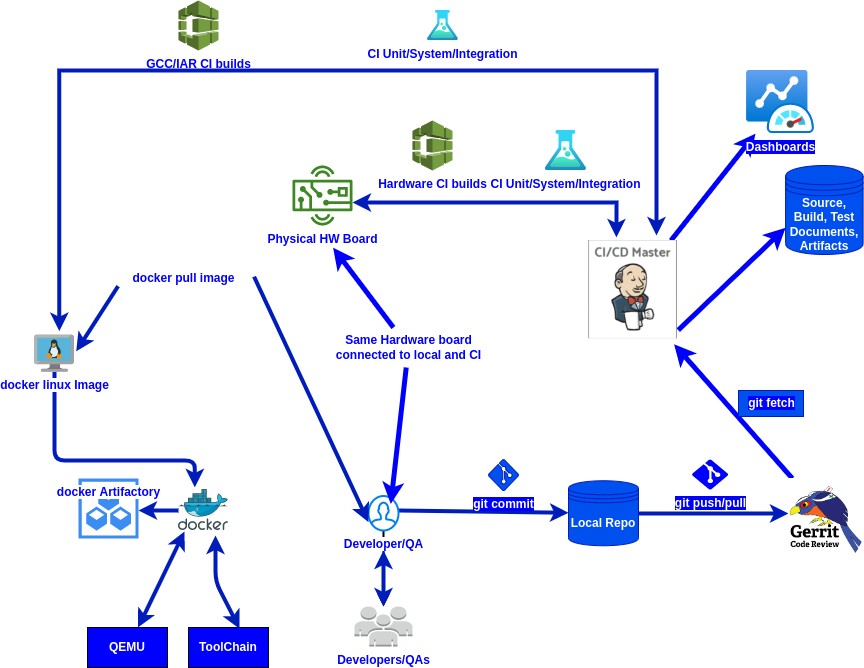


Figure 10: Embedded CI/CD flow

For ensuring versatility and compatibility, multiple compiler configurations are essential. The [GNU Compiler Collection (GCC)](#_bookmark9) offers an extensive suite of software compilers for various programming languages. It is widely recognized and utilized for its efficiency and [versatility[227].](#_bookmark318) On the other hand, [Ingenjörsfirma](#_bookmark25) [Anders Rundgren](#_bookmark25) provides specialized compiler solutions for embedded systems, ensuring optimal performance and reduced footprint, crucial for

resource-constrained [devices[228].](#_bookmark319) Both these compilers are harnessed to guarantee that the Zephyr RTOS firmware is compatible across diverse target platforms and meets the performance requirements of embedded systems.

Automated processes compile the code after each modification. Tools like Clang Static Analyzer[[229]](#_bookmark320) and Coverity[[230]](#_bookmark321) are then utilized for static analysis, detecting potential issues and inconsistencies.

##### Using [Docker[231]](#_bookmark322) for Environment Consistency and QEMU:

[Docker[231]](#_bookmark322) containers are used to maintain a consistent development environment by packaging necessary dependencies for the Zephyr RTOS firmware compilation. This approach reduces environmental discrepancies during builds. [Quick](#_bookmark26) [Emulator[232]](#_bookmark323) serves as a hardware emulator, allowing for testing in a simulated environment. This early-stage testing is crucial for detecting potential problems before deployment to actual devices.

The docker [image[233]](#_bookmark324) with the [QEMU](#_bookmark16) is pushed and saved in the jfrog [artifactory[234]](#_bookmark325) for the developers to use.

##### Testing Frameworks:

Ztest also called as Zephyr Test Framework which provides unit testing framework, which is applied to validate the correctness of individual segments of the Zephyr RTOS [code[235].](#_bookmark326)

The [Pytest[236]](#_bookmark327) framework, checks the overall functionality and interactions between the system’s components. The Pytest framework works as system and integration testing.

##### Deployment and Continuous Monitoring:

Jenkins, an established continuous integration [tool[237],](#_bookmark328) orchestrates an array of critical operations within the development [workflow[238].](#_bookmark329) Upon each code submission to the repository, a cascade of processes is triggered. This cascade encompasses code compilation, unit testing facilitated by the Ztest framework, and system and integration testing conducted using the Pytest framework.

Developers receive prompt notifications of any anomalies detected during the build or testing phases.

The Jenkins interface provides comprehensive dashboards, detailing metrics such as build failure instances and processing durations. Additionally, comprehensive reports are archived for retrospective analysis. Continual monitoring after deployment is instrumental in assuring the consistent and effective operation of

the firmware.

## Proposed: CI/CD Pipeline with Fuzzing

In light of the previously detailed CI/CD pipeline, it becomes imperative to explore enhancements that could further fortify the software development lifecycle. One such augmentation revolves around the incorporation of fuzzing within the CI/CD pipeline, thereby aiming to unearth vulnerabilities that conventional testing approaches might overlook.

Fuzzing, or fuzz testing, is a dynamic code testing technique that involves injecting malformed or random data into a system to uncover vulnerabilities, such as memory leaks, crashes, or other forms of unstable behaviors. In the context of a CI/CD pipeline, integrating fuzzing translates to routinely subjecting the developing software to a barrage of irregular inputs, thereby aspiring to discern and rectify potential security vulnerabilities before deployment.

### 5.3.1 Integration into Existing Pipeline

To assimilate fuzzing within the existing CI/CD pipeline, certain modifications and additions are necessitated. Firstly, a fuzzing tool compatible with the development environment needs to be selected. Several robust, open-source fuzzing tools such as AFL (American Fuzzy Lop) and LibFuzzer are available, offering varying degrees of customization and coverage.

Once an appropriate fuzzing tool is elected, it is integrated into the CI/CD pipeline, preferably at stages involving testing. Post the standard unit and integration tests, the codebase is subjected to fuzz testing. This additional layer of testing ensures that alongside verifying the correctness of code functionalities and integrations, the robustness of the software against malicious or unexpected inputs is also assessed.

Benefits and Challenges:

The integration of fuzzing into the CI/CD pipeline brings forth several benefits. Primarily, it enhances software security by proactively identifying and addressing vulnerabilities, thereby reducing the risks associated with software exploits.

Additionally, it fosters the development of more resilient software, as developers become cognizant of potential issues and refine their coding practices [accordingly[239].](#_bookmark330)

However, this integration is not without its challenges. Fuzzing can be resource-intensive, potentially prolonging the duration of the CI/CD pipeline.

Moreover, it might yield false positives, necessitating additional time and resources to discern genuine vulnerabilities. Hence, a balanced approach, where the depth of fuzzing is aligned with the project’s criticality and available resources, is [essential[240].](#_bookmark331)

The figure:1[1](#_bookmark80) illustrates the proposed IT Infrastructure for CI/CD flow for the case in company with fuzz testing.

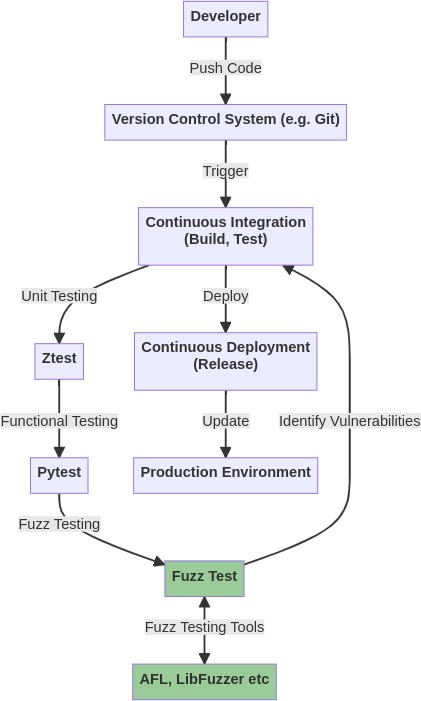


Figure 11: CI/CD flow With Fuzz Testing

# Design, Implementations and Results

This chapter outlines the methodology applied in designing and implementing file fuzzer integration with a target program, which interprets inputs from files within the Zephyr RTOS. This program, being one of the samples in Zephyr RT[OS[221],](#_bookmark312) has been chosen to underscore the practical relevance of this study.

While various fuzzers have been discussed in the preceding sections, [AFL++[49]](#_bookmark140) and [libFuzzer[50]](#_bookmark141) have been selected for demonstration due to their distinct capabilities and widespread usage. These fuzzers serve to emphasize the significance of the study, considering the prevalent application of real-time operating systems in modern embedded devices.

The fuzzing tests have been conducted on a Linux system, where [AFL++[49]](#_bookmark140) and [libFuzzer[50]](#_bookmark141) have already been installed, and the Zephyr RTOS environment has been set up.

## Implementations Using AFL++

This section delineates the detailed steps for setting up the [AFL++[49]](#_bookmark140) fuzzer using a [Docker[231]](#_bookmark322) container and compiling the target program.

The selection of the AFL++ Docker image as the basis for the initial setup was driven by the recommendations provided by the AFL++ documentation. This particular image is endorsed due to its comprehensive inclusion of all necessary AFL++ tools, consolidated in a singular location, thereby facilitating a streamlined and efficient setup [process[49].](#_bookmark140)

Steps to start fuzzing with AFL++:

Step 1: Docker Script for AFL++

The first step involves pulling the AFL++ Docker image and running a container:

#### 1 #!/bin/bash

#### 2 # Pull the AFL++ Docker image

3 $ docker pull aflplusplus/aflplusplus

#### 4 # Run a container using the pulled image

5 $ docker run -ti -v /home/uname:/home/uname --name afl\_container aflplusplus/aflplusplus

#### 6 # Get into the container

7 $ docker exec -it afl\_container /bin/bash

This script facilitates the deployment of the AFL++ fuzzer in a Docker container, streamlining the setup process for fuzzing tests.

Step 2: List of Files

The relevant files for the next step are:

* main.c: The target program in [appendix-1.](#_bookmark343)
* CMakeLists.txt: Build configuration file.

The detailed implementation of the target program can be found in [Appendix-1.](#_bookmark343) This program, written in C, exemplifies the integration of fuzz testing in a Zephyr RTOS environment.

Explanation of the Main Program:

The main.c program serves as the target for the fuzz testing, designed to illustrate potential vulnerabilities within the Zephyr RTOS. It’s structured to simulate a scenario where specific inputs could trigger a failure case, providing a practical example for the fuzzing tests conducted.

This particular program was chosen due to its inherent characteristics that are pivotal for fuzz testing. Fuzz testing is coverage-based and necessitates the

concealment of a failure point— in this instance, a write through a null pointer—deep within the call structure of the program. While such a concealed failure is typically challenging to uncover with randomly-selected input, a fuzzer is designed to locate it in a relatively linear time by systematically exploring each function.

Even in scenarios characterized by a low probability of discovering the failure, as in 1 in 256, which conventionally demands extensive computational time and resources, the fuzzer is capable of identifying the vulnerability in approximately 20 seconds. This efficiency is attributed to the distinct testability of each code segment, necessitating the generation of specific handler functions and the deactivation of certain optimizations.

The program contains several key components:

* **global\_null\_ptr**: A pointer that, under certain conditions, can be triggered to write through a null, simulating a failure.
* **key Array and found Array**: These arrays are used to check against the input data and keep track of discovered keys.
* **GEN\_CHECK Macro**: This macro generates a series of check functions that recursively call each other to simulate a deep call tree.

Upon execution, the program reads input data from a file. It then processes this data through the series of check functions generated by the GEN\_CHECK macro. If a match with the key array is found, it simulates a failure by attempting a write through the global\_null\_ptr. The failure case is hidden deep within the call tree, making it difficult to discover through random inputs but still discoverable through fuzz testing.

For a more detailed view of the main.c file, refer to [Appendix-1.](#_bookmark343)

**Explanation of the CMakeLists.txt:** The CMakeLists.txt file is used for configuring the build system of the sample\_zephyr\_afl project in the zephyr RTOS. Here is an explanation of its components:

* cmake\_minimum\_required(VERSION 3.20.0): This command

specifies the minimum version of CMake that is required, which is version 3.20.0 in this case.

* project(sample\_zephyr\_afl): This command sets the name of the project to sample\_zephyr\_afl.
* option(USE\_AFL "Use afl" OFF): This command declares an option named USE\_AFL that determines whether to use AFL. It is set to OFF by default.
* The conditional block if(USE\_AFL) checks whether USE\_AFL is turned on. If it is, the compiler is set to afl-gcc-fast[[241],](#_bookmark332) and additional compiler flags for coverage are added. If USE\_AFL is turned off, the compiler is set to gcc[[227].](#_bookmark318)
* add\_executable(zephyr\_aflfuzzer src/main.c): This command specifies that an executable named zephyr\_aflfuzzer should be created from the source file src/main.c.

For a more detailed view of the CMakeLists.txt file, refer to [Appendix-1.](#_bookmark343)

Step 3: Build and Compile

A new directory, build, is created to contain all the build files. The cmake command is invoked with the -DUSE\_AFL=ON option, indicating that the build should be configured to use afl-gcc-fast[[241].](#_bookmark332)

|  |  |  |
| --- | --- | --- |
| 1 | $ | mkdir build |
| 2 | $ | cd build |
| 3 | $ | cmake -DUSE\_AFL=ON .. |
| 4 | $ | make |

Step 4: Valid Input for the Target

A binary file is created with a specific sequence of bytes. This file will serve as the initial and valid input or corpus for the fuzzer, allowing it to explore different code paths.

1 $ echo -n -e '\x9e\x21\x0c\x18\x9d\xd1\x7e' > input\_data/test\_data.bin

Step 5: Testing, Results, Analysis and Report

AFL++ is invoked with the aflfuzz command inside the docker container, specifying the input directory containing the corpus input\_data and the output directory output. The built binary zephyr\_aflfuzzer is then executed with AFL++’s syntax for specifying input files.

1 $ afl-fuzz -i input\_data/ -o output/ --

./sample\_zephyr\_afl/build/zephyr\_aflfuzzer @@

This section breaks down the components of this [command[49]:](#_bookmark140)

* afl-fuzz: This is the command to initiate the AFL++ fuzzer.
* -i input\_data/: The -i flag specifies the input directory containing the ‘corpus’ of sample input files, which serve as a basis to generate new test cases.
* -o output/: The -o flag denotes the output directory where AFL stores the results of the fuzzing session, including any crashes or hangs it discovers.
* --: This symbol denotes the end of the options passed to the AFL fuzzer. Any parameter listed after this is considered as an argument to the program being fuzzed, not as an option to AFL.
* ./sample\_zephyr\_afl/build/zephyr\_aflfuzzer: This is the path to the target binary that will be fuzzed by AFL.
* @@: This placeholder is replaced by AFL with the name of a file containing the test case each time the target binary is run, allowing AFL to run the program with a multitude of different inputs without needing to modify the command line each time.

The [Figure-12](#_bookmark83) shows the AFL++ execution screen.

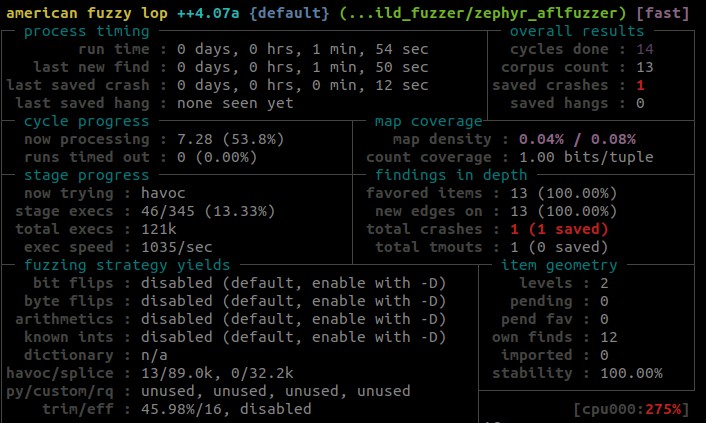


Figure 12: AFL++ Execution Screen

Important elements of the user interface shown [below[49]:](#_bookmark140)

* **process\_timing:** This section shows how long the fuzzer has been running and the current time.
* **stage progress:** Here, AFL++ displays the progress of the current fuzzing stage in terms of execution and the total number of inputs to be tested in this stage.
* **findings in depth:** This area provides details about the paths discovered, including the number of unique crashes, hangs, and paths found during the fuzzing session.
* **item geometry:** This displays information related to the paths taken through the program, such as the depth and width of the explored paths.

As depicted in [Figure-12,](#_bookmark83) the binary generated (as detailed in Appendix-1) resulted in AFL++ creating a total of 13 corpus and identifying one crash. Remarkably, a scenario with a probability of 1 in 256, which would traditionally require several months to years of computation in a large datacenter, was resolved by the fuzzer in under two minutes.

The resulting outputs are stored in the *output* directory. The coverage report of the

execution can be generated using *afl-cov*[[242].](#_bookmark333) The corresponding command is as follows:

1 $ afl-cov -d output/ --coverage-cmd "./sample\_zephyr\_afl/build/zephyr\_aflfuzzer @@" --code-dir

./sample\_zephyr\_afl/

The [Figure-13](#_bookmark84) shows the afl-cov report of the execution.

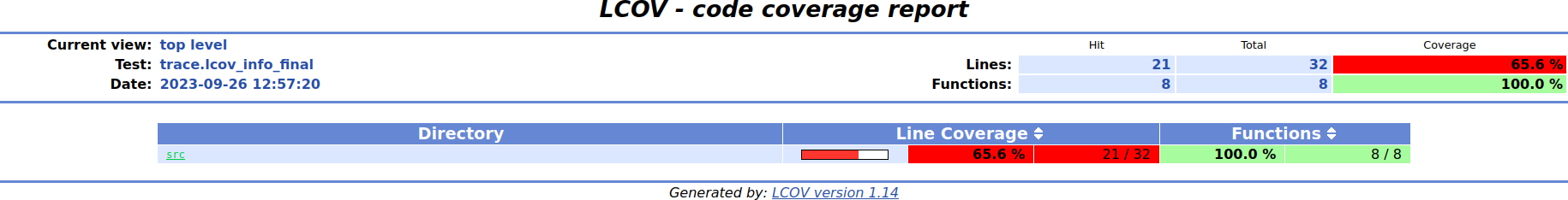


Figure 13: LCOV Coverage [Report[243]](#_bookmark334)

AFL++ includes a visualization tool called afl-plot[[244],](#_bookmark335) which is instrumental in assessing the performance of fuzzing campaigns. This tool generates a series of graphs that provide insights into several crucial metrics throughout the fuzzing process. These metrics include, but are not limited to, the number of crashes discovered, the speed of execution, and the progression of path discovery over time.

1 $ afl-plot output/default/ output/output\_plot/

The [Figure-14](#_bookmark85) shows the afl-plot report of the execution.

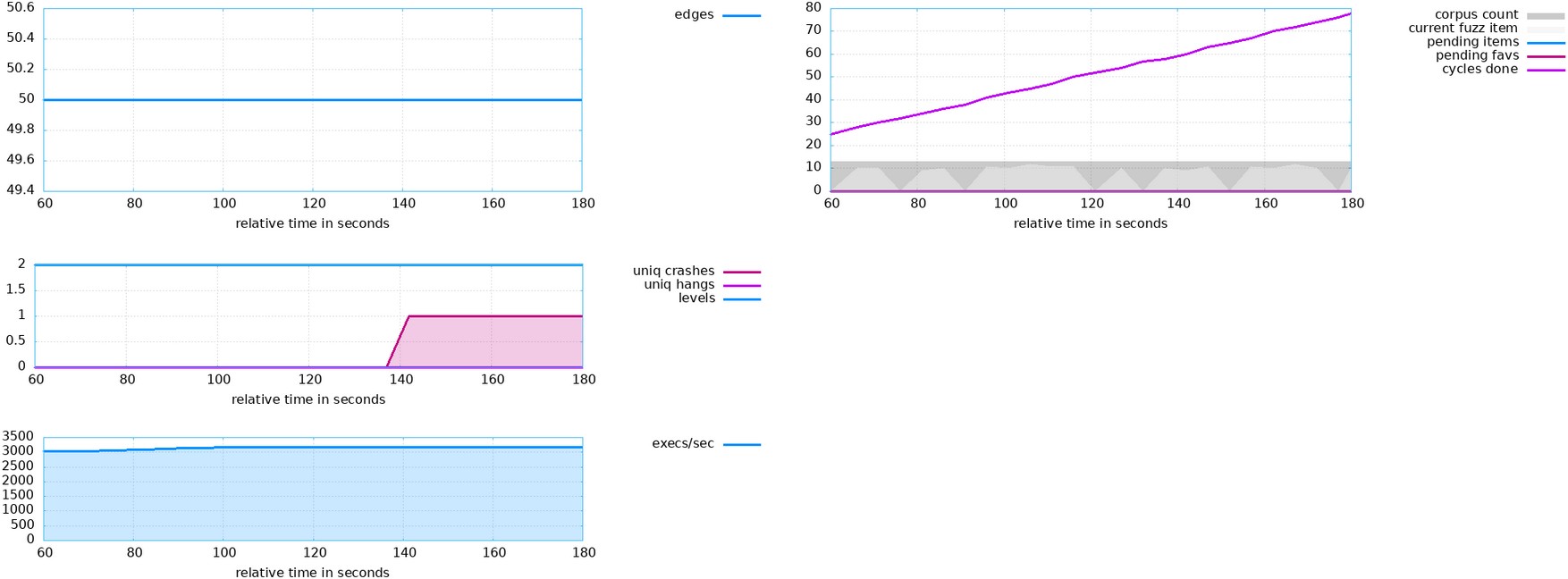


Figure 14: afl-plot Report

The fuzzing tests adeptly illustrate the utilization of the AFL++ fuzzer, accompanied by tools such as afl-cov and afl-plot, highlighting the vulnerabilities that can be unearthed, alongside generating comprehensive coverage and graphical representations. This proof of concept underscores the potency of employing fuzzing with AFL++, demonstrating its utility in uncovering concealed vulnerabilities and enhancing system robustness.

The AFL++ fuzzer, renowned for its efficiency and effectiveness, exemplifies its capability to discern vulnerabilities, even those embedded deep within the code structure, in a remarkably truncated timeframe. The accompanying tools, afl-cov and afl-plot, augment this process by providing detailed coverage reports and graphical data, thereby facilitating a more nuanced understanding of the system’s susceptibility to potential threats. The insights derived from this study illuminate the myriad possibilities and benefits of integrating AFL++ into the development cycle, underscoring its potential contribution to fortifying system security.

## Implementations Using libFuzzer

[LibFuzzer[50],](#_bookmark141) an integral part of the LLVM [Compiler[245]](#_bookmark336) Infrastructure, is chosen for this Proof of Concept (PoC) due to its distinct attributes and capabilities.

LibFuzzer is renowned for its efficiency in performing coverage-guided fuzzing, which is instrumental in identifying hidden vulnerabilities within a [system[246].](#_bookmark337)

The fuzzer’s seamless integration with the Clang [compiler[247]](#_bookmark338) and its capability to perform in-memory fuzzing make it a valuable tool in assessing the security and robustness of software systems.

Steps to start fuzzing with libFuzzer:

Step 1: Setting up Environment

To set up the necessary environment, a Linux-based operating system, specifically Ubuntu, is utilized. Furthermore, the installation of a recent version of

the Clang compiler is requisite for the successful execution of the [libFuzzer[247].](#_bookmark338)

Step 2: List of Files

The relevant files for the next step are:

* main.c: The target program in [Appendix-2.](#_bookmark344)
* CMakeLists.txt: Build configuration file.

The detailed implementation of the target program can be found in [Appendix-2.](#_bookmark344)

Comparing this program in [Appendix-2](#_bookmark344) to the one utilized for AFL++ in [Appendix-1,](#_bookmark343) there are fundamental differences that can be highlighted:

This is a simple example of fuzz test integration with Zephyr apps that displays LLVM libfuzzer’s most important feature: it’s ability to detect and explore deep and complicated call trees by exploiting coverage information gleaned from instrumented binaries.

* **Entry Point:** In the context of libFuzzer, the program uses a function named LLVMFuzzerTestOneInput as the entry point, which is specifically designed for integration with libFuzzer. This contrasts with the AFL++ target program, where the main function serves as the entry point.
* **Input Handling:** The libFuzzer target program directly uses the input data provided to the LLVMFuzzerTestOneInput function, representing a more straightforward handling of input compared to the AFL++ target program, which might necessitate additional steps for input retrieval and processing.

**Explanation of the CMakeLists.txt:** The CMakeLists.txt file is used for configuring the build system of the sample\_zephyr\_libFuzzer project in the zephyr RTOS. Here is an explanation of its components:

* cmake\_minimum\_required(VERSION 3.20.0): Specifies the minimum required version of CMake to be 3.20.0.
* project(sample\_zephyr\_libfuzzer): Names the project as

sample\_zephyr\_libfuzzer.

* option(USE\_LIBFUZZER "Build with libfuzzer" OFF): Declares an option USE\_LIBFUZZER, which is turned OFF by default but can be

set to ON to enable building with libFuzzer.

* set(CMAKE\_C\_COMPILER clang): Sets the C compiler for the project to Clang.
* if(USE\_LIBFUZZER): Checks if USE\_LIBFUZZER is enabled.
  + add\_definitions(-DUSE\_LIBFUZZER): Adds a definition for USE\_LIBFUZZER for the preprocessor.
  + set(CMAKE\_C\_FLAGS "...

-fsanitize=fuzzer,address,undefined"): Sets the compiler flags to enable sanitizers and fuzzer.

* + set(CMAKE\_EXE\_LINKER\_FLAGS "...

-fsanitize=fuzzer,address,undefined"): Sets the linker flags to enable sanitizers and fuzzer.

* + set(CMAKE\_C\_FLAGS "... -fprofile-instr-generate

-fcoverage-mapping"): Enables flags for generating profile instrumentation and coverage mapping.

* add\_executable(zephyr\_libfuzzer src/main.c): Defines an

executable target named zephyr\_libfuzzer from the source file

src/main.c.

* if(USE\_LIBFUZZER): If USE\_LIBFUZZER is enabled, sets additional linker flags for the target zephyr\_libfuzzer to enable the necessary sanitizers for fuzzing.

Step 3: Build and Compile

A new directory, build, is created to contain all the build files. The cmake command is invoked with the -DUSE\_LIBFUZZER=ON option, indicating that the build should be configured to use clang[[247].](#_bookmark338)

|  |  |  |
| --- | --- | --- |
| 1 | $ | mkdir build |
| 2 | $ | cd build\_fuzzer |
| 3 | $ | cmake -DUSE\_LIBFUZZER=ON .. |
| 4 | $ | make |
| 5 | $ | ./zephyr\_libfuzzer |

Step 4: Testing, Results and Analysis

The CMake configuration for the sample\_zephyr\_libfuzzer project sets a robust foundation for enabling fuzz testing with libFuzzer and Clang’s sanitizers. The flexibility to toggle the libFuzzer integration is an essential attribute, as it allows the program to be compiled with or without the fuzzing infrastructure. When the USE\_LIBFUZZER option is activated, the system adeptly applies the necessary compiler and linker flags to incorporate libFuzzer, address sanitizer, and undefined behavior sanitizer. The added instrumentation flags for profile generation and coverage mapping suggest that the project not only focuses on uncovering vulnerabilities but also on gauging the coverage of the testing process. This kind of approach ensures a two-fold outcome: vulnerability discovery and code execution coverage analysis. Ultimately, it provides a comprehensive insight into the robustness against arbitrary and malicious inputs, rendering a pivotal report on its security stature.

The [Figure-15](#_bookmark87) shows the libFuzzer execution output.

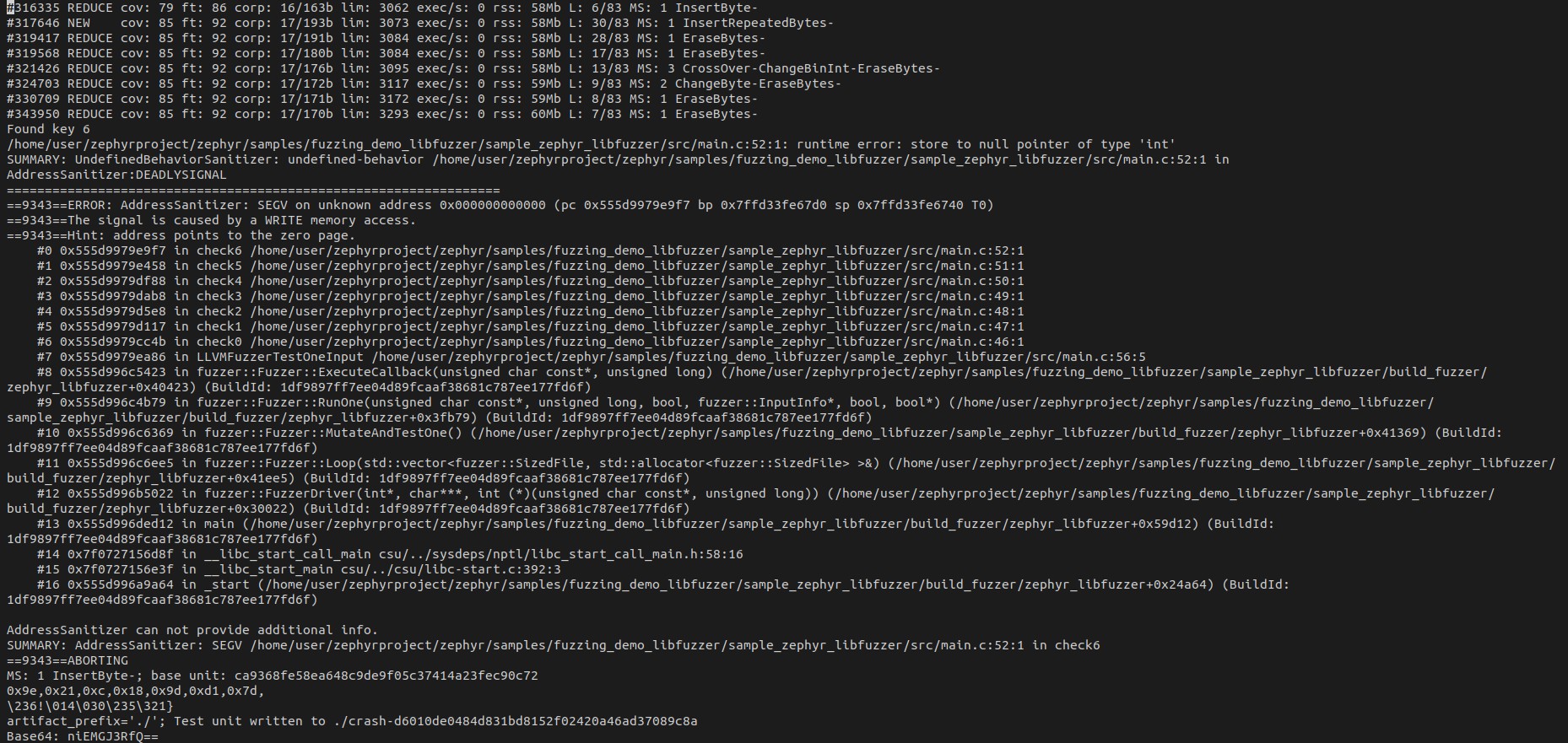


Figure 15: AFL++ Execution Screen

The fuzzer initialized its execution with a specific random seed, identified as [75803490815.](#_bookmark87) To reproduce the same sequence of fuzzing events and outcomes, one may rerun the fuzzer specifying the argument -seed=758034908.

1 INFO: Running with entropic power schedule (0xFF, 100).

2 INFO: Seed: 758034908

3 INFO: Loaded 1 modules (157 inline 8-bit counters): 157 [0x555d997e79a8, 0x555d997e7a45),

4 INFO: Loaded 1 PC tables (157 PCs): 157 [0x555d997e7a48,0x555d997e8418),

5 INFO: -max\_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes

6 INFO: A corpus is not provided, starting from an empty corpus

In its default configuration, libFuzzer operates under the assumption that the input size is constrained to 4096 bytes or less. To modify this default input size constraint, two potential strategies exist: the application of the -max\_len=N argument, where N represents the desired maximum input length, or the initiation of the fuzzer using a seed corpus that contains non-empty entries. For this PoC, no corpus or input is provided, hence the fuzzer starts with an empty corpus.

#### 1 #330709 REDUCE cov: 85 ft: 92 corp: 17/171b lim: 3172 exec/s: 0 rss: 59Mb L: 8/83 MS: 1 EraseBytes-

#### 2 #343950 REDUCE cov: 85 ft: 92 corp: 17/170b lim: 3293 exec/s: 0 rss: 60Mb L: 7/83 MS: 1 EraseBytes-

3 Found key 6

4

/home/user/zephyrproject/zephyr/samples/fuzzing\_demo\_libfuzzer/sample\_ runtime error: store to null pointer of type 'int'

5 SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior

/home/user/zephyrproject/zephyr/samples/fuzzing\_demo\_libfuzzer/sample\_

**in**

6 AddressSanitizer:DEADLYSIGNAL

7 =================================================================

8 ==9343==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x555d9979e9f7 bp 0x7ffd33fe67d0 sp 0x7ffd33fe6740 T0)

9 ==9343==The signal is caused by a WRITE memory access.

10 ==9343==Hint: address points to the zero page.

During its execution, libFuzzer has processed a minimum of 316,335(#330709) distinct inputs. From these, it identified 17 unique inputs with a combined size of 170 bytes (corpus: 17 entries totaling 170 bytes) that cumulatively achieve coverage over 85 specific points within the target program.

Remarkably, for one particular input, the integrated AddressSanitizer[[152]](#_bookmark243) identified an anomaly characterized as undefined behavior. Consequently, this prompted the termination of the fuzzer’s execution.

Prior to its termination, libFuzzer took the precaution of persisting the problematic input to the disk, ensuring that the specific circumstances leading to the anomaly could be independently revisited. To replicate the observed behavior without engaging in further fuzzing, one can execute the following command:

1 ./zephyr\_libfuzzer crash-d6010de0484d831bd8152f02420a46ad37089c8a

To amalgamate the profiling data, the following command utilizing llvm-profdata

can be employed:

1 llvm-profdata merge -sparse default.profraw -o default.profdata

This operation produces the files default.profraw and default.profdata, capturing the consolidated profiling information.

For a terminal-based summary of the coverage, the llvm-cov tool provides a report as follows:

1 llvm-cov report ./zephyr\_libfuzzer -instr-profile=default.profdata

Additionally, to create a more detailed visualization in HTML format, the subsequent command can be used:

1 llvm-cov show ./zephyr\_libfuzzer -instr-profile=default.profdata

-format=html -output-dir=coverage

This command generates an HTML report, which can be conveniently reviewed in a web browser, providing an insightful representation of the code’s coverage.

# Discussions and Conclusions

## Discussions

Fuzzing, in the realm of software testing, is multifaceted. At its core, fuzzing can be relatively straightforward to deploy, acting as a quick initial sieve for potential vulnerabilities. However, in other contexts, its implementation demands extensive groundwork, especially when targeting complex or novel systems. The findings from fuzzing may vary – while some vulnerabilities are promptly detected, others remain elusive. One inherent limitation of fuzzing is its inability to conclusively determine a system’s robustness. The absence of detected bugs doesn’t necessarily indicate a system’s invulnerability; instead, it could highlight areas where the fuzzer might require optimization or redirection.

The landscape of fuzzing is dynamic. Although numerous open-source fuzzers continuously emerge, often enhancing specific components of fuzzing, their evolution is twofold. Some lose traction, lacking the maintenance to remain relevant. In contrast, others merge into larger projects contributing their innovations to more extensive frameworks. A notable examples in the realm of modern fuzzing tools are AFL++, libFuzzer, Atheris and other custom fuzzers which exemplify the principle of continuous adaptation and improvement in the fuzzing community.

The dawn of AI-enhanced [fuzzing[248]](#_bookmark339) introduces another layer of complexity.

AI-driven fuzzing tools, leveraging machine learning models, promise adaptive test scenarios and heightened vulnerability detection rates. They aim to overcome the traditional limitations of fuzzers, offering the potential for more context-aware

and target-specific vulnerability detection. However, while promising, this AI infusion is still in initial stages, requiring comprehensive evaluation over time.

Reflecting on this journey of exploring fuzzing, it became evident that the field, although a sub-discipline of software testing, offers a vast expanse of knowledge. Venturing into this domain presented numerous challenges, from grasping foundational concepts to navigating intricate tooling setups. Notably, substantial time was invested in comprehending the basics and configuring systems aptly for fuzzing. With this foundational understanding now in place, future endeavors can pivot towards assessing varied fuzzing methodologies across diverse scenarios.

In conclusion, while fuzzing provides an invaluable avenue for enhancing system security, the path forward involves continuous learning, tool assessment, and methodological refinement. As the field evolves, especially with AI’s inclusion, it’s imperative to stay updated, prioritizing objectives to harness the maximum potential of fuzzing in software testing.

## Conclusions

This research highlights the significant potential of fuzzing within software testing for embedded systems. Hands-on experimentation using both libFuzzer and AFL++ within the embedded system architecture of the case company demonstrated their effectiveness and ability to swiftly identify vulnerabilities. The study showed that these fuzzing tools can detect issues more rapidly than certain traditional methods.

A notable aspect of these fuzzing tools is the incorporation of built-in reporting mechanisms. This functionality demystifies the intricate task of determining and comprehending code coverage, assuring that potential vulnerabilities are duly identified. The automation and clarity these tools provide can significantly enhance the efficiency of the testing process.

Looking ahead, there exists a strong rationale for deeper integration of fuzzing into

standard software development lifecycles, especially within continuous testing frameworks. As the intricacy of software escalates, so does the need for thorough and streamlined testing processes. The adoption of fuzzing tools, as evidenced by the outcomes using libFuzzer and AFL++, presents a tactical advantage for organizations, guaranteeing software that is both operational and resilient to potential threats.

To conclude, this thesis, while presenting an overview of the prevailing scenario, also serves as a roadmap for the case company’s onward testing strategies. The importance of persistently evolving testing methodologies is accentuated, and with the knowledge derived from the empirical tests, the company stands poised for the adoption of more sophisticated and integrated testing approaches.

# References

1. Muench, Marius; Stijohann, Jan; Kargl, Frank; Francillon, Aurélien & Balzarotti, Davide. 2018. “What you corrupt is not what you crash: Challenges in fuzzing embedded devices.” In: *NDSS*.
2. Eisele, Max; Maugeri, Marcello; Shriwas, Rachna; Huth, Christopher & Bella, Giampaolo. 2022. “Embedded fuzzing: a review of challenges, tools, and solutions”. In: Cybersecurity 5.1, p. 18.
3. Andrae, Anders SG & Andersen, Otto. 2010. “Life cycle assessments of consumer electronics—are they consistent?” In: The International Journal of Life Cycle Assessment 15, pp. 827–836.
4. Paulin, Pierre G; Liem, Clifford; May, Trevor C & Sutarwala, Shailesh. 1995. “DSP design tool requirements for embedded systems: A telecommunications industrial perspective”. In: Journal of VLSI signal processing systems for signal, image and video technology 9.1-2, pp. 23–47.
5. Automotive industry | History, Overview, Definition, Developments, & Facts | Britannica n.d. [https://www.britannica.com/technology/automotive-industry.](https://www.britannica.com/technology/automotive-industry) (Accessed on 06/03/2023).
6. Bieber, Pierre; Blanquart, Jean-Paul; Descargues, Gilles;

Dulucq, Michael; Fourastier, Yannick; Hazane, Eric; Julien, Mathias; Léonardon, Laurent & Sarouille, Gabrielle. 2012. “Security and safety assurance for aerospace embedded systems”. In: *Embedded real time software and systems (ERTS2012)*.

1. Jafari, Roozbeh; Ghiasi, Soheil & Sarrafzadeh, Majid. 2007. Medical embedded systems. Springer.
2. Thramboulidis, Kleanthis C; Doukas, G & Koumoutsos, G. 2007. “A SOA-based embedded systems development environment for industrial automation”. In: EURASIP Journal on Embedded Systems 2008,

pp. 1–15.

1. Gridling, Gunther & Weiss, Bettina. 2007. “Introduction to microcontrollers”. In: Vienna University of Technology Institute of Computer Engineering Embedded Computing Systems Group.
2. Kang, Won Min; Moon, Seo Yeon & Park, Jong Hyuk. 2017. “An enhanced security framework for home appliances in smart home”. In: Human-centric Computing and Information Sciences 7, pp. 1–12.
3. Kostic, Zoran & Seetharaman, Selvaraj. 1997. “Digital signal processors in cellular radio communications”. In: IEEE Communications Magazine 35.12, pp. 22–35.
4. Marwedel, Peter. 2021. Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things. Springer Nature.
5. Camposano, Raul & Wilberg, Jörg. 1996. “Embedded system design”. In: Design Automation for Embedded Systems 1.1, pp. 5–50.
6. IoT connected devices worldwide 2019-2030 | Statista n.d. (Accessed on 10/22/2022).
7. How Many IoT Devices Are There in 2023? N.d. [https://techjury.net/blog/how-many-iot-devices-are-there/.](https://techjury.net/blog/how-many-iot-devices-are-there/) (Accessed on 03/25/2023).
8. State of the IoT 2020: 12 billion IoT connections n.d.

[https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-](https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/) [surpassing-non-iot-for-the-first-time/.](https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/) (Accessed on 03/25/2023).

1. Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper - Cisco n.d. [https://www.cisco.com/c/en/us/solutions/collateral/executive-](https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html) [perspectives/annual-internet-report/white-paper-c11-741490.html.](https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html) (Accessed on 03/25/2023).
2. Wang, Dongyu; Lo, Dixon; Bhimani, Janak & Sugiura, Kazunori. 2015. “Anycontrol–iot based home appliances monitoring and controlling”. In: *2015 IEEE 39th Annual Computer Software and Applications*

*Conference*. Vol. 3. IEEE, pp. 487–492.

1. Poongodi, T; Krishnamurthi, Rajalakshmi; Indrakumari, R; Suresh, P & Balusamy, Balamurugan. 2020. “Wearable devices and IoT”. In: A handbook of Internet of Things in biomedical and cyber physical system, pp. 245–273.
2. Priyan, MK & Devi, G Usha. 2019. “A survey on internet of vehicles: applications, technologies, challenges and opportunities”. In: International Journal of Advanced Intelligence Paradigms 12.1-2, pp. 98–119.
3. Internet of things (IoT) in nano-integrated wearable biosensor devices for healthcare applications - ScienceDirect n.d. [https://www-sciencedirect-](https://www-sciencedirect-com.ezproxy.metropolia.fi/science/article/pii/S2590137022000486) [com.ezproxy.metropolia.fi/science/article/pii/S2590137022000486.](https://www-sciencedirect-com.ezproxy.metropolia.fi/science/article/pii/S2590137022000486) (Accessed on 06/04/2023).
4. Baranwal, Tanmay; Pateriya, Pushpendra Kumar, et al. 2016. “Development of IoT based smart security and monitoring devices for

agriculture”. In: *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence)*. IEEE, pp. 597–602.

1. Pradhan, Bikash; Bhattacharyya, Saugat & Pal, Kunal. 2021. “IoT-based applications in healthcare devices”. In: Journal of healthcare engineering 2021, pp. 1–18.
2. Patil, VC; Al-Gaadi, KA; Biradar, DP & Rangaswamy, M. 2012. “Internet of things (Iot) and cloud computing for agriculture: An overview”. In: Proceedings of agro-informatics and precision agriculture (AIPA 2012), India, pp. 292–296.
3. KrebsOnSecurity Hit With Record DDoS – Krebs on Security n.d. [https:](https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/)

[//krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/.](https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/) (Accessed on 06/04/2023).

1. Antonakakis, Manos; April, Tim; Bailey, Michael; Bernhard, Matt; Bursztein, Elie; Cochran, Jaime; Durumeric, Zakir; Halderman, J Alex; Invernizzi, Luca; Kallitsis, Michalis, et al. 2017. “Understanding the mirai botnet”. In: *26th USENIX security symposium ( USENIX Security 17)*, pp. 1093–1110.

*{ } { }*

1. Large DDoS attacks cause outages at Twitter, Spotify, and other sites | TechCrunch n.d. [https://techcrunch.com/2016/10/21/many-sites-](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [including-twitter-and-spotify-suffering-outage/?guccounter=1&guce\_](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce\_referrer\_](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [sig=AQAAACUQcZRMSZBGfovGj-](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI\_](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [ZcT7c1yvQ3QZfaR1qX\_](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce\_9\_](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) [edReucn\_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL.](https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAACUQcZRMSZBGfovGj-5oofsBcBcYBVwes3rlSX630JTBdQjWezoD40UI_ZcT7c1yvQ3QZfaR1qX_fAEtBe7sKR21FtT5snWGgo83Gwkh66PLfCZbmfhvffPX8dnXce_9_edReucn_BqjHLSoynQU3R1LfWuQ2LLSmLBix7xwh5kL) (Accessed on 06/04/2023).
2. Lindqvist, Ulf & Neumann, Peter G. 2017. “The future of the Internet of Things”. In: Communications of the ACM 60.2, pp. 26–30.
3. Treck TCP Datasheet - Treck | Embedded TCP/IP | Internet Protocols

n.d. [https://treck.com/treck-tcp-datasheet/.](https://treck.com/treck-tcp-datasheet/) (Accessed on 06/04/2023).

1. What is URGENT/11 vulnerability? | Armis Research n.d. [https://www.armis.com/research/urgent11/.](https://www.armis.com/research/urgent11/) (Accessed on 06/04/2023).
2. IoT riddled with BadAlloc vulnerabilities n.d. [https://www.malwarebytes.com/blog/news/2021/04/iot-riddled-with-](https://www.malwarebytes.com/blog/news/2021/04/iot-riddled-with-badalloc-vulnerabilities) [badalloc-vulnerabilities.](https://www.malwarebytes.com/blog/news/2021/04/iot-riddled-with-badalloc-vulnerabilities) (Accessed on 06/04/2023).
3. VxWorks | Industry Leading RTOS for Embedded Systems n.d. [https://www.windriver.com/products/vxworks.](https://www.windriver.com/products/vxworks) (Accessed on 06/04/2023).
4. Seri, Ben; Vishnepolsky, Gregory & Zusman, Dor. 2019. “Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS”. In: White Paper, ARMIS, URGENT/11.
5. (1) New Messages! N.d. [https://www.imperva.com/learn/application-](https://www.imperva.com/learn/application-security/remote-code-execution/) [security/remote-code-execution/.](https://www.imperva.com/learn/application-security/remote-code-execution/) (Accessed on 06/04/2023).
6. 10 of the Biggest Data Breaches in History | DataProt n.d. [https://dataprot.net/articles/biggest-data-breaches/.](https://dataprot.net/articles/biggest-data-breaches/) (Accessed on 06/04/2023).
7. The 15 biggest data breaches of the 21st century | CSO Online n.d. [https://www.csoonline.com/article/2130877/the-biggest-data-breaches-](https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html) [of-the-21st-century.html.](https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html) (Accessed on 06/04/2023).
8. Myers, Glenford J; Sandler, Corey & Badgett, Tom. 2011. The art of software testing. John Wiley & Sons.
9. Broekman, Bart & Notenboom, Edwin. 2003. Testing embedded software. Pearson Education.
10. Itkonen, Juha; Mantyla, Mika V & Lassenius, Casper. 2009. “How do testers do it? An exploratory study on manual testing practices”. In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, pp. 494–497.
11. Asfaw, Daniel. 2015. “Benefits of automated testing over manual testing”. In: International Journal of Innovative Research in Information Security 2.1, pp. 5–13.
12. Whittaker, J.A. 2000a. “What is software testing? And why is it so hard?” In: IEEE Software 17.1, pp. 70–79.
13. Deng, Yinlin; Xia, Chunqiu Steven; Yang, Chenyuan;

Zhang, Shizhuo Dylan; Yang, Shujing & Zhang, Lingming. 2023. “Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt”. In: arXiv preprint arXiv:2304.02014.

1. Cunningham, Sean; Gambo, Jemil; Lawless, Aidan; Moore, Declan; Yilmaz, Murat; Clarke, Paul M & O’Connor, Rory V. 2019. “Software testing: a changing career”. In: *Systems, Software and Services Process Improvement: 26th European Conference, EuroSPI 2019, Edinburgh,*

*UK, September 18–20, 2019, Proceedings 26*. Springer, pp. 731–742.

1. Koopman, Philip & Wagner, Michael. 2016. “Challenges in autonomous vehicle testing and validation”. In: SAE International Journal of Transportation Safety 4.1, pp. 15–24.
2. Eisele, Max. 2022. “Debugger-driven Embedded Fuzzing”. In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*,

pp. 483–485.

1. Fowler, Daniel S; Bryans, Jeremy; Shaikh, Siraj Ahmed &

Wooderson, Paul. 2018. “Fuzz testing for automotive cyber-security”. In:

*2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, pp. 239–246.

1. Felderer, Michael; Büchler, Matthias; Johns, Martin; Brucker, Achim D; Breu, Ruth & Pretschner, Alexander. 2016. “Security testing: A survey”. In: *Advances in Computers*. Vol. 101. Elsevier, pp. 1–51.
2. Yun, Joobeom; Rustamov, Fayozbek; Kim, Juhwan & Shin, Youngjoo. 2022. “Fuzzing of Embedded Systems: A Survey”. In: ACM Computing Surveys 55.7, pp. 1–33.
3. Fioraldi, Andrea; Maier, Dominik; Eißfeldt, Heiko & Heuse, Marc. Aug. 2020. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[<https://www.usenix.org/conference/woot20/presentation/fioral](https://www.usenix.org/conference/woot20/presentation/fioraldi)di>.

1. libFuzzer – a library for coverage-guided fuzz testing. — LLVM 17.0.0git documentation n.d. [https://llvm.org/docs/LibFuzzer.html.](https://llvm.org/docs/LibFuzzer.html) (Accessed on 02/25/2023).
2. What is an IP (Intellectual Property) core in Semiconductors? - AnySilicon

n.d. [https://anysilicon.com/ip-intellectual-property-core-semiconductors/.](https://anysilicon.com/ip-intellectual-property-core-semiconductors/) (Accessed on 06/09/2023).

1. What is Key Management? How does Key Management work? | Encryption Consulting n.d. [https://www.encryptionconsulting.com/education-center/what-is-key-](https://www.encryptionconsulting.com/education-center/what-is-key-management) [management.](https://www.encryptionconsulting.com/education-center/what-is-key-management) (Accessed on 06/09/2023).
2. What is Provisioning? - Keeper Security n.d. [https:](https://www.keepersecurity.com/resources/glossary/what-is-provisioning/)

[//www.keepersecurity.com/resources/glossary/what-is-provisioning](https://www.keepersecurity.com/resources/glossary/what-is-provisioning/)/. (Accessed on 06/09/2023).

1. Types of Security Protocols - Bitdefender Cyberpedia n.d. [https://www.bitdefender.com/cyberpedia/types-of-security-protocols/.](https://www.bitdefender.com/cyberpedia/types-of-security-protocols/) (Accessed on 06/09/2023).
2. Kwon, Sang Jib; Park, Eunil & Kim, Ki Joon. 2014. “What drives successful social networking services? A comparative analysis of user acceptance of Facebook and Twitter”. In: The Social Science Journal 51.4, pp. 534–544.
3. Differential Power Analysis (DPA) Countermeasures - Silicon Labs n.d. [https://www.silabs.com/security/differential-power-analysis.](https://www.silabs.com/security/differential-power-analysis) (Accessed on 06/09/2023).
4. SIDE-CHANNEL ATTACKS: How Differential Power Analysis (DPA) and Simple Power Analysis (SPA) Works - AnySilicon n.d. [https://anysilicon.com/side-channel-attacks-differential-power-analysis-](https://anysilicon.com/side-channel-attacks-differential-power-analysis-dpa-simple-power-analysis-spa-works/) [dpa-simple-power-analysis-spa-works/.](https://anysilicon.com/side-channel-attacks-differential-power-analysis-dpa-simple-power-analysis-spa-works/) (Accessed on 06/09/2023).
5. Standaert, François-Xavier. 2010. “Introduction to side-channel attacks”. In: Secure integrated circuits and systems, pp. 27–42.
6. Putting A Hardware Root-of-Trust To Work In An Anti-Counterfeiting IC

n.d. [https://semiengineering.com/putting-a-hardware-root-of-trust-to-](https://semiengineering.com/putting-a-hardware-root-of-trust-to-work-in-an-anti-counterfeiting-ic/) [work-in-an-anti-counterfeiting-ic/.](https://semiengineering.com/putting-a-hardware-root-of-trust-to-work-in-an-anti-counterfeiting-ic/) (Accessed on 06/09/2023).

1. What is Root of Trust? - Utimaco n.d. [https://utimaco.com/products/technologies/hardware-security-](https://utimaco.com/products/technologies/hardware-security-modules/what-root-trust) [modules/what-root-trust.](https://utimaco.com/products/technologies/hardware-security-modules/what-root-trust) (Accessed on 06/09/2023).
2. What is Hardware? Types & Examples you need to know n.d. [https://softwarelab.org/what-is-hardware/.](https://softwarelab.org/what-is-hardware/) (Accessed on 06/09/2023).
3. What Is Firmware? Firmware Definition, Types and Examples | Fortinet

n.d. [https://www.fortinet.com/resources/cyberglossary/what-is-firmware.](https://www.fortinet.com/resources/cyberglossary/what-is-firmware) (Accessed on 06/09/2023).

1. ISO - ISO 9001 and related standards — Quality management n.d. [https://www.iso.org/iso-9001-quality-management.html.](https://www.iso.org/iso-9001-quality-management.html) (Accessed on 06/09/2023).
2. What is Exploratory Testing? N.d. [https://www.guru99.com/exploratory-testing.html.](https://www.guru99.com/exploratory-testing.html) (Accessed on 06/09/2023).
3. Baker, Yolanda S; Agrawal, Rajeev & Bhattacharya, Sambit. 2013. “Analyzing security threats as reported by the united states computer emergency readiness team (US-CERT)”. In: *2013 IEEE International Conference on Intelligence and Security Informatics*. IEEE, pp. 10–12.
4. DevOps in CI/CD & Its benefits to your Business. - Silicon Mind n.d. [https://silicon-mind.com/what-is-ci-cd-in-devops-continuous/.](https://silicon-mind.com/what-is-ci-cd-in-devops-continuous/) (Accessed on 06/09/2023).
5. Software | Definition, Types, & Facts | Britannica n.d. [https://www.britannica.com/technology/software.](https://www.britannica.com/technology/software) (Accessed on 01/04/2023).
6. Stallings, William. 1998. Operating systems internals and design principles. Prentice-Hall, Inc.
7. Definition of application program | PCMag n.d. [https://www.pcmag.com/encyclopedia/term/application-program.](https://www.pcmag.com/encyclopedia/term/application-program) (Accessed on 06/10/2023).
8. What is Systems Software? | Webopedia n.d. [https://www.webopedia.com/definitions/systems-software/.](https://www.webopedia.com/definitions/systems-software/) (Accessed on 06/10/2023).
9. Yetisen, Ali Kemal; Martinez-Hurtado, JL;

Cruz Vasconcellos, Fernando da; Simsekler, MC Emre;

Akram, Muhammad Safwan & Lowe, Christopher R. 2014. “The regulation of mobile medical applications”. In: Lab on a Chip 14.5, pp. 833–840.

1. Lehman, Meir M & Ramil, Juan F. 2002. “Software evolution and software evolution processes”. In: Annals of Software Engineering 14, pp. 275–309.
2. Kernighan, Brian W & Ritchie, Dennis M. 2002. “The C programming language”. In.
3. Van Rossum, Guido et al. 2007. “Python Programming Language.” In: *USENIX annual technical conference*. Vol. 41. 1. Santa Clara, CA, pp. 1–36.
4. Rebouças, Marcel; Pinto, Gustavo; Ebert, Felipe; Torres, Weslley; Serebrenik, Alexander & Castor, Fernando. 2016. “An empirical study on the usage of the swift programming language”. In: *2016 IEEE 23rd*

*international conference on software analysis, evolution, and reengineering (SANER)*. Vol. 1. IEEE, pp. 634–638.

1. Samuel, Stephen & Bocutiu, Stefan. 2017. Programming kotlin. Packt Publishing Ltd.
2. What Is Swift Used For? N.d. [https://www.codecademy.com/resources/blog/what-is-swift-used-for/.](https://www.codecademy.com/resources/blog/what-is-swift-used-for/) (Accessed on 06/10/2023).
3. What Is Kotlin Used For? N.d. [https://www.codecademy.com/resources/blog/what-is-kotlin-used-for/.](https://www.codecademy.com/resources/blog/what-is-kotlin-used-for/) (Accessed on 06/10/2023).
4. Goosen, Leila. 2008. “A brief history of choosing first programming languages”. In: *History of Computing and Education 3 (HCE3) IFIP 20 th World Computer Congress, Proceedings of the Third IFIP Conference on the History of Computing and Education WG 9.7/TC9, History of*

*Computing, September 7–10, 2008, Milano, Italy*. Springer, pp. 167–170.

1. Kahn, Beverly K; Strong, Diane M & Wang, Richard Y. 2002. “Information quality benchmarks: product and service performance”. In: Communications of the ACM 45.4, pp. 184–192.
2. Ryker, Randy; Nath, Ravinder & Henson, James. 1997. “Determinants of computer user expectations and their relationships with user satisfaction: an empirical study”. In: Information Processing & Management 33.4,

pp. 529–537.

1. Pham, Hoang. 2007. System software reliability. Springer Science & Business Media.
2. Mishra, Alok; Alzoubi, Yehia Ibrahim; Gill, Asif Qumer &

Anwar, Memoona Javeria. 2022. “Cybersecurity enterprises policies: a Comparative study”. In: Sensors 22.2, p. 538.

1. Rust, Roland T & Zahorik, Anthony J. 1993. “Customer satisfaction, customer retention, and market share”. In: Journal of retailing 69.2, pp. 193–215.
2. Boehm, Barry W; Brown, John R & Lipow, Myron. 1976. “Quantitative evaluation of software quality”. In: *Proceedings of the 2nd international conference on Software engineering*, pp. 592–605.
3. Kiruthika, Jay & Khaddaj, Souheil. 2015. “Software quality issues and challenges of internet of things”. In: *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. IEEE, pp. 176–179.
4. Chung, Lawrence; Nixon, Brian A; Yu, Eric & Mylopoulos, John. 2012. Non-functional requirements in software engineering. Vol. 5. Springer Science & Business Media.
5. Cavano, Joseph P & McCall, James A. 1978. “A framework for the measurement of software quality”. In: *Proceedings of the software quality assurance workshop on Functional and performance issues*,

pp. 133–139.

1. Jain, Parita; Sharma, Arun & Ahuja, Laxmi. 2018. “The Impact of Agile Software Development Process on the Quality of Software Product”. In: *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pp. 812–815.
2. Whittaker, James A. 2000b. “What is software testing? And why is it so hard?” In: IEEE software 17.1, pp. 70–79.
3. Bentley, John E; Bank, Wachovia & Charlotte, NC. 2005. “Software testing fundamentals—concepts, roles, and terminology”. In:

*Proceedings of SAS Conference*, pp. 1–12.

1. Jamil, Muhammad Abid; Arif, Muhammad;

Abubakar, Normi Sham Awang & Ahmad, Akhlaq. 2016. “Software testing techniques: A literature review”. In: *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*. IEEE, pp. 177–182.

1. Arumugam, Arun Kumar. 2019. “Software Testing Techniques & New Trends”. In: International Journal of Engineering Research & Technology (IJERT) 8.12.
2. Mili, Ali & Tchier, Fairouz. 2015. Software testing: Concepts and operations. John Wiley & Sons.
3. Klees, George; Ruef, Andrew; Cooper, Benji; Wei, Shiyi &

Hicks, Michael. 2018. “Evaluating fuzz testing”. In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 2123–2138.

1. Li, Jun; Zhao, Bodong & Zhang, Chao. 2018. “Fuzzing: a survey”. In: Cybersecurity 1.1, pp. 1–13.
2. Kim, Hyoung Chun; Choi, Young Han & Lee, Dong Hoon. 2011. “Efficient file fuzz testing using automated analysis of binary file format”. In: Journal of Systems Architecture 57.3, pp. 259–268.
3. SEGEDY, Bc PATRIK. N.d. “FUZZ TESTING OF REST API”. In: ().
4. Shen, Quanjiang; Wen, Mi; Zhang, Lei; Wang, Liangliang; Shen, Leyan & Cheng, Jing. 2021. “A systematic review of fuzzy testing for information systems and applications”. In: *2021 2nd International Conference on*

*Electronics, Communications and Information Technology (CECIT)*, pp. 156–162.

1. Takanen, Ari. 2009. “Fuzzing: the Past, the Present and the Future”. In: Actes du, pp. 202–212.
2. What is a Mission Critical System? - Definition from Techopedia n.d. [https://www.techopedia.com/definition/23583/mission-critical-system.](https://www.techopedia.com/definition/23583/mission-critical-system) (Accessed on 06/10/2023).
3. Böhme, Marcel; Cadar, Cristian & Roychoudhury, Abhik. 2020. “Fuzzing: Challenges and reflections”. In: IEEE Software 38.3, pp. 79–86.
4. Vidas, Tim; Franco, Kristopher; Heffner, Jason, et al. 2019. “FUZZY: A Framework for Secure Fuzzing of Zigbee Implementations”. In: arXiv preprint arXiv:1904.04824.
5. Beaman, Craig; Redbourne, Michael; Mummery, J Darren &

Hakak, Saqib. 2022. “Fuzzing vulnerability discovery techniques: survey, challenges and future directions”. In: Computers & Security, p. 102813.

1. Durumeric, Zakir; Li, Frank; Kasten, James; Amann, Johanna; Beekman, Jethro; Payer, Mathias; Weaver, Nicolas; Adrian, David;

Paxson, Vern; Bailey, Michael, et al. 2014. “The matter of heartbleed”. In: *Proceedings of the 2014 conference on internet measurement conference*, pp. 475–488.

1. Shetty, Rushank; Choo, Kim-Kwang Raymond & Kaufman, Robert. 2018. “Shellshock vulnerability exploitation and mitigation: a demonstration”. In: *International Conference on Applications and Techniques in Cyber*

*Security and Intelligence: Applications and Techniques in Cyber Security and Intelligence*. Springer, pp. 338–350.

1. Wang, Pengfei; Krinke, Jens; Lu, Kai; Li, Gen & Dodier-Lazaro, Steve. 2017a. “How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel”. In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security’17)*. Vol. 26. The Advanced Computing Systems Association.
2. Haller, Istvan; Slowinska, Asia; Neugschwandtner, Matthias &

Bos, Herbert. 2013. “Dowsing for overflows: a guided fuzzer to find buffer boundary violations.” In: *USENIX Security Symposium*. USA, pp. 49–64.

1. Röning, J et al. 1999. “PROTOS–Security Testing of Protocol Implementations”. In: Computer Engineering Laboratory, University of Oulu 2003.
2. Godefroid, Patrice; Levin, Michael Y & Molnar, David. 2012. “SAGE: whitebox fuzzing for security testing”. In: Communications of the ACM 55.3, pp. 40–44.
3. Adobe Flash, Reader, and Acrobat Vulnerability | CISA n.d. [https://www.cisa.gov/news-events/alerts/2010/06/08/adobe-flash-reader-](https://www.cisa.gov/news-events/alerts/2010/06/08/adobe-flash-reader-and-acrobat-vulnerability) [and-acrobat-vulnerability.](https://www.cisa.gov/news-events/alerts/2010/06/08/adobe-flash-reader-and-acrobat-vulnerability) (Accessed on 04/02/2023).
4. “Mayhem” Declared Preliminary Winner of Historic Cyber Grand Challenge n.d. [https://www.darpa.mil/news-events/2016-08-04.](https://www.darpa.mil/news-events/2016-08-04) (Accessed on 04/02/2023).
5. GitHub - microsoft/onefuzz: A self-hosted Fuzzing-As-A-Service platform

n.d. [https://github.com/microsoft/onefuzz.](https://github.com/microsoft/onefuzz) (Accessed on 04/02/2023).

1. american fuzzy lop n.d. [https://lcamtuf.coredump.cx/afl/#bugs.](https://lcamtuf.coredump.cx/afl/#bugs) (Accessed on 04/02/2023).
2. syzkaller/found\_bugs.md at master · google/syzkaller · GitHub n.d. [https://github.com/google/syzkaller/blob/master/docs/found\_bugs.md.](https://github.com/google/syzkaller/blob/master/docs/found_bugs.md) (Accessed on 04/02/2023).
3. GitHub - dvyukov/go-fuzz: Randomized testing for Go n.d. [https://github.com/dvyukov/go-fuzz#trophies.](https://github.com/dvyukov/go-fuzz#trophies) (Accessed on 04/02/2023).
4. GitHub - google/honggfuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based) n.d. [https://github.com/google/honggfuzz#trophies.](https://github.com/google/honggfuzz#trophies) (Accessed on 04/02/2023).
5. Sapienz: Intelligent automated software testing at scale - Engineering at Meta n.d. [https://engineering.fb.com/2018/05/02/developer-](https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/) [tools/sapienz-intelligent-automated-software-testing-at-scale/.](https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/) (Accessed on 04/02/2023).
6. ClusterFuzz | Documentation for ClusterFuzz n.d. [https://google.github.io/clusterfuzz/#trophies.](https://google.github.io/clusterfuzz/#trophies) (Accessed on 04/02/2023).
7. Lin, Van-Thuan; Nguyen, Duc-Hiep; Chieu, Tuong &

Roychoudhury, Abhik. 2020. “AFLNet: A Greybox Fuzzer for Network Protocols”. In: *2020 IEEE/ACM 42nd International Conference on*

*Software Engineering (ICSE)*. IEEE, pp. 510–521.

1. Kim, Seungsoo; Kim, Hyunwoo; Jeong, Jaeho; Lee, Sooel &

Yoon, Yongdae. 2019. “IoTcube: An Automated IoT App Security Testing Framework”. In: *Proceedings of the Network and Distributed System*

*Security Symposium*.

1. Srivastava, Prashast; Peng, Hui; Li, Jiahao; Okhravi, Hamed; Shrobe, Howard & Payer, Mathias. 2019. “Firmfuzz: Automated iot firmware introspection and analysis”. In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pp. 15–21.
2. Google. 2020. Atheris: A Coverage-guided, Native Python Fuzzer. [https://github.com/google/atheris.](https://github.com/google/atheris)
3. What is Fuzz Testing [Complete Guide] | Code Intelligence n.d. [https://www.code-intelligence.com/what-is-fuzz-testing.](https://www.code-intelligence.com/what-is-fuzz-testing) (Accessed on 06/11/2023).
4. fuzzing/afl-based-fuzzers-overview.md at master · google/fuzzing · GitHub n.d. [https://github.com/google/fuzzing/blob/master/docs/afl-](https://github.com/google/fuzzing/blob/master/docs/afl-based-fuzzers-overview.md) [based-fuzzers-overview.md.](https://github.com/google/fuzzing/blob/master/docs/afl-based-fuzzers-overview.md) (Accessed on 04/16/2023).
5. DeMott, Jared. 2006. “The evolving art of fuzzing”. In: Def Con 14.
6. Shahriar, Hossain; North, Sarah & Mawangi, Edward. 2014. “Testing of memory leak in android applications”. In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE,

pp. 176–183.

1. Godefroid, Patrice. 2020. “Fuzzing: Hack, art, and science”. In: Communications of the ACM 63.2, pp. 70–76.
2. Pietikäinen, Pekka; Kettunen, Atte & Röning, Juha. 2016. “Steps Towards Fuzz Testing in Agile Test Automation”. In: International Journal of Secure Software Engineering (IJSSE) 7.1, pp. 38–52.
3. Unit Testing Vs Fuzz Testing - Two Sides of the Same Coin? N.d. [https://www.code-intelligence.com/blog/unit-testing-vs-fuzz-testing.](https://www.code-intelligence.com/blog/unit-testing-vs-fuzz-testing) (Accessed on 06/11/2023).
4. Oehlert, Peter. 2005. “Violating assumptions with fuzzing”. In: IEEE Security & Privacy 3.2, pp. 58–62.
5. fuzzing/why-fuzz.md at master · google/fuzzing · GitHub n.d. [https://github.com/google/fuzzing/blob/master/docs/why-fuzz.md.](https://github.com/google/fuzzing/blob/master/docs/why-fuzz.md) (Accessed on 06/11/2023).
6. Fu, Jingzhou; Liang, Jie; Wu, Zhiyong; Wang, Mingzhe & Jiang, Yu. 2022. “Griffin: Grammar-Free DBMS Fuzzing”. In: *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–12.
7. Fuzzing VIM. AAAAAAAAAA….: It’s almost a year now I… | by Dhiraj | Medium n.d.

[https://medium.com/@dhiraj\_mishra/fuzzing-vim-53d7cf9b5561.](https://medium.com/%40dhiraj_mishra/fuzzing-vim-53d7cf9b5561) (Accessed on 06/11/2023).

1. Presentation Title n.d. [http:](http://events17.linuxfoundation.org/sites/events/files/slides/Weir-Fuzzing.pdf)

[//events17.linuxfoundation.org/sites/events/files/slides/Weir-Fuzzing.pdf.](http://events17.linuxfoundation.org/sites/events/files/slides/Weir-Fuzzing.pdf) (Accessed on 06/11/2023).

1. Thiel, David. 2008. “Exposing vulnerabilities in media software”. In: *Proc. Black Hat EU*.
2. Rhoden, Barret; Sturton, Cynthia & Lewis, Colleen. N.d. “Fuzzing like a Ninja: Using Structured Random Data to Precisely Fuzz Media Players”. In: ().
3. Schumilo, Sergej; Aschermann, Cornelius; Gawlik, Robert;

Schinzel, Sebastian & Holz, Thorsten. 2017. “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels.” In: *USENIX Security Symposium*, pp. 167–182.

1. Corina, Jake; Machiry, Aravind; Salls, Christopher; Shoshitaishvili, Yan; Hao, Shuang; Kruegel, Christopher & Vigna, Giovanni. 2017. “Difuze: Interface aware fuzzing for kernel drivers”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138.
2. Wang, Junjie; Chen, Bihuan; Wei, Lei & Liu, Yang. 2017b. “Skyfire: Data-driven seed generation for fuzzing”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 579–594.
3. openssl/README.md at master · openssl/openssl · GitHub n.d. [https://github.com/openssl/openssl/blob/master/fuzz/README.md.](https://github.com/openssl/openssl/blob/master/fuzz/README.md) (Accessed on 06/11/2023).
4. GitHub - libressl/fuzz: fuzz code from openssl updated to target libressl

n.d. [https://github.com/libressl/fuzz.](https://github.com/libressl/fuzz) (Accessed on 06/11/2023).

1. Fuzz testing in Chromium n.d. [https://chromium.googlesource.com/](https://chromium.googlesource.com/chromium/src/%2B/main/testing/libfuzzer/README.md) [chromium/src/+/main/testing/libfuzzer/README.md.](https://chromium.googlesource.com/chromium/src/%2B/main/testing/libfuzzer/README.md) (Accessed on 06/11/2023).
2. Fuzzing — Firefox Source Docs documentation n.d. [https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html.](https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html) (Accessed on 06/11/2023).
3. GitHub - nccgroup/fuzzowski: the Network Protocol Fuzzer that we will want to use. N.d. [https://github.com/nccgroup/fuzzowski.](https://github.com/nccgroup/fuzzowski) (Accessed on 06/11/2023).
4. SnapFuzz: New fuzzing tool speeds up testing of network applications | The Daily Swig n.d. [https://portswigger.net/daily-swig/snapfuzz-new-](https://portswigger.net/daily-swig/snapfuzz-new-fuzzing-tool-speeds-up-testing-of-network-applications) [fuzzing-tool-speeds-up-testing-of-network-applications.](https://portswigger.net/daily-swig/snapfuzz-new-fuzzing-tool-speeds-up-testing-of-network-applications) (Accessed on 06/11/2023).
5. McNally, Richard; Yiu, Ken; Grove, Duncan & Gerhardy, Damien. 2012. “Fuzzing: the state of the art”. In.
6. Manès, Valentin JM; Han, HyungSeok; Han, Choongwoo; Cha, Sang Kil; Egele, Manuel; Schwartz, Edward J & Woo, Maverick. 2019. “The art, science, and engineering of fuzzing: A survey”. In: IEEE Transactions on Software Engineering 47.11, pp. 2312–2331.
7. Miller, Charlie; Peterson, Zachary NJ, et al. 2007. “Analysis of mutation and generation-based fuzzing”. In: Independent Security Evaluators, Tech. Rep 4.
8. Lyu, Chenyang; Ji, Shouling; Zhang, Xuhong; Liang, Hong; Zhao, Binbin; Lu, Kangjie & Beyah, Raheem. 2022. “Ems: History-driven mutation for coverage-based fuzzing”. In: *29rd Annual Network and Distributed*

*System Security Symposium, NDSS*, pp. 24–28.

1. Pang, Chengbin; Liu, Hongbin; Wang, Yifan; Gong, Neil Zhenqiang; Mao, Bing & Xu, Jun. 2023. “Generation-based fuzzing? Don’t build a new generator, reuse!” In: Computers & Security 129, p. 103178.
2. GitHub - google/sanitizers: AddressSanitizer, ThreadSanitizer, MemorySanitizer n.d. [https://github.com/google/sanitizers.](https://github.com/google/sanitizers) (Accessed on 06/12/2023).
3. Österlund, Sebastian; Razavi, Kaveh; Bos, Herbert & Giuffrida, Cristiano. 2020. “Parmesan: Sanitizer-guided greybox fuzzing”. In: *Proceedings of the 29th USENIX Conference on Security Symposium*, pp. 2289–2306.
4. What is a Central Processing Unit? – Arm® n.d. [https://www.arm.com/glossary/cpu.](https://www.arm.com/glossary/cpu) (Accessed on 06/17/2023).
5. What is RAM (Random Access Memory)? - Definition from SearchStorage n.d. [https://www.techtarget.com/searchstorage/definition/RAM-random-](https://www.techtarget.com/searchstorage/definition/RAM-random-access-memory) [access-memory.](https://www.techtarget.com/searchstorage/definition/RAM-random-access-memory) (Accessed on 06/17/2023).
6. AddressSanitizer · google/sanitizers Wiki · GitHub n.d. [https://github.com/google/sanitizers/wiki/AddressSanitizer.](https://github.com/google/sanitizers/wiki/AddressSanitizer) (Accessed on 06/17/2023).
7. MemorySanitizer · google/sanitizers Wiki · GitHub n.d. [https://github.com/google/sanitizers/wiki/MemorySanitizer.](https://github.com/google/sanitizers/wiki/MemorySanitizer) (Accessed on 06/17/2023).
8. Stepanov, Evgeniy & Serebryany, Konstantin. 2015. “MemorySanitizer: fast detector of uninitialized memory use in C++”. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, pp. 46–55.
9. UndefinedBehaviorSanitizer — Clang 17.0.0git documentation n.d. [https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.](https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html) (Accessed on 06/17/2023).
10. Jääskelä, Esa. 2016. “Genetic algorithm in code coverage guided fuzz testing”. In: University of Oulu, p. 16.
11. Böhme, Marcel; Pham, Van-Thuan & Roychoudhury, Abhik. 2016. “Coverage-based greybox fuzzing as markov chain”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1032–1043.
12. The Magic Behind Feedback-Based Fuzzing n.d. [https://www.code-](https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing) [intelligence.com/blog/the-magic-behind-feedback-based-fuzzing.](https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing) (Accessed on 06/12/2023).
13. Luciano Remes | What is a Fuzzer? N.d. [https://www.lremes.com/posts/fuzzing/.](https://www.lremes.com/posts/fuzzing/) (Accessed on 06/12/2023).
14. Liang, Hongliang; Pei, Xiaoxiao; Jia, Xiaodong; Shen, Wuwei &

Zhang, Jian. 2018. “Fuzzing: State of the art”. In: IEEE Transactions on Reliability 67.3, pp. 1199–1218.

1. Bekrar, Sofia; Bekrar, Chaouki; Groz, Roland & Mounier, Laurent. 2012. “A taint based approach for smart fuzzing”. In: *2012 IEEE Fifth*

*International Conference on Software Testing, Verification and Validation*. IEEE, pp. 818–825.

1. Peng, Hui; Shoshitaishvili, Yan & Payer, Mathias. 2018. “T-Fuzz: fuzzing by program transformation”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 697–710.
2. Chen, Yang; Groce, Alex; Zhang, Chaoqiang; Wong, Weng-Keen;

Fern, Xiaoli; Eide, Eric & Regehr, John. 2013. “Taming compiler fuzzers”. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pp. 197–208.

1. Godefroid, Patrice. 2007a. “Random testing for security: blackbox vs. whitebox fuzzing”. In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International*

*Conference on Automated Software Engineering (ASE 2007)*, pp. 1–1.

1. Canakci, Sadullah; Delshadtehrani, Leila; Eris, Furkan;

Taylor, Michael Bedford; Egele, Manuel & Joshi, Ajay. 2021. “Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, pp. 529–534.

1. Godefroid, Patrice; Levin, Michael Y; Molnar, David A, et al. 2008. “Automated whitebox fuzz testing.” In: *NDSS*. Vol. 8, pp. 151–166.
2. Godefroid, Patrice; Kiezun, Adam & Levin, Michael Y. 2008. “Grammar-based whitebox fuzzing”. In: *Proceedings of the 29th ACM SIGPLAN conference on programming language design and*

*implementation*, pp. 206–215.

1. Herrera, Adrian; Gunadi, Hendra; Magrath, Shane; Norrish, Michael; Payer, Mathias & Hosking, Antony L. 2021. “Seed selection for successful fuzzing”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 230–243.
2. Jurczyk, M. 2016. “Effective file format fuzzing”. In: Black hat Europe.
3. Maier, Dominik; Radtke, Benedikt & Harren, Bastian. Aug. 2019. “Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing”. In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association.

[<https://www.usenix.org/conference/woot19/presentation/maie](https://www.usenix.org/conference/woot19/presentation/maier)r>.

1. GitHub - google/AFL: american fuzzy lop - a security-oriented fuzzer n.d. [https://github.com/google/AFL.](https://github.com/google/AFL) (Accessed on 02/25/2023).
2. Godefroid, Patrice. 2007b. “Compositional dynamic test generation”. In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 47–54.
3. ClusterFuzzLite - ClusterFuzzLite n.d. (Accessed on 10/23/2022).
4. GitHub - google/oss-fuzz: OSS-Fuzz - continuous fuzzing for open source software. N.d. (Accessed on 10/23/2022).
5. GitHub - google/fuzztest n.d. [https://github.com/google/fuzztest.](https://github.com/google/fuzztest) (Accessed on 06/18/2023).
6. GitHub - HypothesisWorks/hypothesis: Hypothesis is a powerful, flexible, and easy to use library for property-based testing. N.d. [https://github.com/HypothesisWorks/hypothesis/.](https://github.com/HypothesisWorks/hypothesis/) (Accessed on 06/18/2023).
7. Introduction of Embedded Systems | Set-1 - GeeksforGeeks n.d. [https:](https://www.geeksforgeeks.org/introduction-of-embedded-systems-set-1/)

[//www.geeksforgeeks.org/introduction-of-embedded-systems-set-1](https://www.geeksforgeeks.org/introduction-of-embedded-systems-set-1/)/. (Accessed on 06/18/2023).

1. Introduction To Embedded System Basics and Applications n.d. [https:](https://www.elprocus.com/basics-of-embedded-system-and-applications/)

[//www.elprocus.com/basics-of-embedded-system-and-applications/.](https://www.elprocus.com/basics-of-embedded-system-and-applications/) (Accessed on 06/18/2023).

1. Main Types of Embedded Systems Worth Knowing - Velvetech n.d. [https://www.velvetech.com/blog/types-of-embedded-systems/.](https://www.velvetech.com/blog/types-of-embedded-systems/) (Accessed on 06/18/2023).
2. Classification of Embedded Systems - GeeksforGeeks n.d. [https://www.geeksforgeeks.org/classification-of-embedded-systems/.](https://www.geeksforgeeks.org/classification-of-embedded-systems/) (Accessed on 06/18/2023).
3. What is a smartwatch? N.d. [https:](https://www.futurelearn.com/info/courses/embedded-systems/0/steps/64773)

[//www.futurelearn.com/info/courses/embedded-systems/0/steps/64773.](https://www.futurelearn.com/info/courses/embedded-systems/0/steps/64773) (Accessed on 06/18/2023).

1. Li, Qing & Yao, Caroline. 2003. Real-time concepts for embedded systems. CRC press.
2. What Are Embedded Systems? N.d. [https://www.trentonsystems.com/blog/what-are-embedded-systems.](https://www.trentonsystems.com/blog/what-are-embedded-systems) (Accessed on 06/18/2023).
3. Types of Embedded Systems | 4 Types of System on Performance Based n.d. [https://www.educba.com/types-of-embedded-systems/.](https://www.educba.com/types-of-embedded-systems/) (Accessed on 06/18/2023).
4. Introduction To Embedded System Basics and Applications n.d. [https:](https://www.elprocus.com/basics-of-embedded-system-and-applications/)

[//www.elprocus.com/basics-of-embedded-system-and-applications/.](https://www.elprocus.com/basics-of-embedded-system-and-applications/) (Accessed on 06/18/2023).

1. What Is a POS System and How Does It Work? N.d. [https://www.investopedia.com/terms/p/point-of-sale.asp.](https://www.investopedia.com/terms/p/point-of-sale.asp) (Accessed on 06/18/2023).
2. Zalewski, Michal. 2014. “American fuzzy lop (2017)”. In: URL: [http://lcamtuf.](http://lcamtuf/) coredump. cx/afl.
3. GitHub - google/honggfuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based) n.d. [https://github.com/google/honggfuzz.](https://github.com/google/honggfuzz) (Accessed on 02/25/2023).
4. Pereyda, Joshua. 2019. “boofuzz Documentation”. In: THIS REFERENCE STILL NEEDS TO BE FIXED.
5. Aki Helin / radamsa · GitLab n.d. [https://gitlab.com/akihe/radamsa.](https://gitlab.com/akihe/radamsa) (Accessed on 04/15/2023).
6. Chen, Daming D; Woo, Maverick; Brumley, David & Egele, Manuel. 2016. “Towards automated dynamic analysis for linux-based embedded firmware.” In: *NDSS*. Vol. 1, pp. 1–1.
7. Zaddach, Jonas; Bruno, Luca; Francillon, Aurelien; Balzarotti, Davide,

et al. 2014. “AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems’ Firmwares.” In: *NDSS*. Vol. 23, pp. 1–16.

1. GitHub - google/atheris n.d. [https://github.com/google/atheris.](https://github.com/google/atheris) (Accessed on 06/09/2023).
2. Heartbleed Bug n.d. [https://heartbleed.com/.](https://heartbleed.com/) (Accessed on 07/02/2023).
3. Short Intro to OSS-Fuzz n.d.

[https://www.code-intelligence.com/blog/intro-to-oss-fuzz.](https://www.code-intelligence.com/blog/intro-to-oss-fuzz) (Accessed on 07/02/2023).

1. GitHub - AFLplusplus/AFLplusplus: The fuzzer afl++ is afl with community patches, qemu 5.1 upgrade, collision-free coverage, enhanced laf-intel & redqueen, AFLfast++ power schedules, MOpt mutators, unicorn\_mode, and a lot more! N.d. [https://github.com/AFLplusplus/AFLplusplus.](https://github.com/AFLplusplus/AFLplusplus) (Accessed on 04/16/2023).
2. GitHub - mboehme/aflfast: AFLFast (extends AFL with Power Schedules)

n.d. [https://github.com/mboehme/aflfast.](https://github.com/mboehme/aflfast) (Accessed on 04/16/2023).

1. GitHub - nccgroup/TriforceLinuxSyscallFuzzer: A linux system call fuzzer using TriforceAFL n.d. [https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.](https://github.com/nccgroup/TriforceLinuxSyscallFuzzer) (Accessed on 04/16/2023).
2. GitHub - Battelle/afl-unicorn: afl-unicorn lets you fuzz any piece of binary that can be emulated by Unicorn Engine. N.d. [https://github.com/Battelle/afl-unicorn.](https://github.com/Battelle/afl-unicorn) (Accessed on 04/16/2023).
3. Baseband Processor - GIGABYTE Global n.d. [https://www.gigabyte.com/Glossary/baseband-processor.](https://www.gigabyte.com/Glossary/baseband-processor) (Accessed on 07/02/2023).
4. Unicorn – The Ultimate CPU emulator n.d. [https://www.unicorn-engine.org/.](https://www.unicorn-engine.org/) (Accessed on 07/02/2023).
5. afl-unicorn: Fuzzing Arbitrary Binary Code | by Nathan Voss | HackerNoon.com | Medium n.d. [https://medium.com/hackernoon/afl-](https://medium.com/hackernoon/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf) [unicorn-fuzzing-arbitrary-binary-code-563ca28936bf.](https://medium.com/hackernoon/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf) (Accessed on 04/16/2023).
6. Lipp, Moritz; Schwarz, Michael; Gruss, Daniel; Prescher, Thomas; Haas, Werner; Horn, Jann; Mangard, Stefan; Kocher, Paul;

Genkin, Daniel; Yarom, Yuval, et al. 2020. “Meltdown: Reading kernel memory from user space”. In: Communications of the ACM 63.6,

pp. 46–56.

1. Kocher, Paul; Horn, Jann; Fogh, Anders; Genkin, Daniel; Gruss, Daniel; Haas, Werner; Hamburg, Mike; Lipp, Moritz; Mangard, Stefan; Prescher, Thomas, et al. 2020. “Spectre attacks: Exploiting speculative execution”. In: Communications of the ACM 63.7, pp. 93–101.
2. Introducing the Rambus CMRT - CryptoManager Root of Trust n.d. [https://www.rambus.com/blogs/introducing-the-rambus-cryptomanager-](https://www.rambus.com/blogs/introducing-the-rambus-cryptomanager-root-of-trust-cmrt/) [root-of-trust-cmrt/.](https://www.rambus.com/blogs/introducing-the-rambus-cryptomanager-root-of-trust-cmrt/) (Accessed on 07/15/2023).
3. What is Computer Memory and What are Different Types? N.d. [https://www.techtarget.com/whatis/definition/memory.](https://www.techtarget.com/whatis/definition/memory) (Accessed on 07/15/2023).
4. Hardware Root of Trust: Everything you need to know - Rambus n.d. [https://www.rambus.com/blogs/hardware-root-of-trust/.](https://www.rambus.com/blogs/hardware-root-of-trust/) (Accessed on 07/31/2023).
5. Root of Trust - Semiconductor Engineering n.d. [https://semiengineering.com/knowledge\_centers/semiconductor-](https://semiengineering.com/knowledge_centers/semiconductor-security/root-of-trust/) [security/root-of-trust/.](https://semiengineering.com/knowledge_centers/semiconductor-security/root-of-trust/) (Accessed on 07/15/2023).
6. Zimmer, Vincent & Krau, Michael. 2016. “Establishing the root of trust”. In: UEFI. org document dated August.
7. What is JTAG and how can I make use of it? - XJTAG Tutorial n.d. [https://www.xjtag.com/about-jtag/what-is-jtag/.](https://www.xjtag.com/about-jtag/what-is-jtag/) (Accessed on 07/31/2023).
8. JTAG hardware debugger – JTAG n.d.

[https://www.jtag.com/jtag-hw-debugger/.](https://www.jtag.com/jtag-hw-debugger/) (Accessed on 07/31/2023).

1. What is Encryption Key Management? | Thales n.d. [https://cpl.thalesgroup.com/faq/key-secrets-management/what-](https://cpl.thalesgroup.com/faq/key-secrets-management/what-encryption-key-management) [encryption-key-management.](https://cpl.thalesgroup.com/faq/key-secrets-management/what-encryption-key-management) (Accessed on 07/31/2023).
2. Needham, Roger M & Schroeder, Michael D. 1978. “Using encryption for authentication in large networks of computers”. In: Communications of the ACM 21.12, pp. 993–999.
3. Nechvatal, James; Barker, Elaine; Bassham, Lawrence; Burr, William; Dworkin, Morris; Foti, James & Roback, Edward. 2001. “Report on the development of the Advanced Encryption Standard (AES)”. In: Journal of research of the National Institute of Standards and Technology 106.3,

p. 511.

1. Milanov, Evgeny. 2009. “The RSA algorithm”. In: RSA laboratories, pp. 1–11.
2. Bos, Joppe W; Halderman, J Alex; Heninger, Nadia; Moore, Jonathan; Naehrig, Michael & Wustrow, Eric. 2014. “Elliptic curve cryptography in practice”. In: *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*. Springer, pp. 157–175.
3. Security — Zephyr Project Documentation n.d. [https://docs.zephyrproject.org/latest/security/index.html.](https://docs.zephyrproject.org/latest/security/index.html) (Accessed on 07/31/2023).
4. Zephyr Project Documentation — Zephyr Project Documentation n.d. [https://docs.zephyrproject.org/latest/.](https://docs.zephyrproject.org/latest/) (Accessed on 07/31/2023).
5. Zephyr Project New - Zephyr Project n.d. [https://www.zephyrproject.org/.](https://www.zephyrproject.org/) (Accessed on 07/31/2023).
6. Kernel in Operating System - GeeksforGeeks n.d. [https://www.geeksforgeeks.org/kernel-in-operating-system/.](https://www.geeksforgeeks.org/kernel-in-operating-system/) (Accessed on 07/31/2023).
7. Loeliger, Jon & McCullough, Matthew. 2012. Version Control with Git: Powerful tools and techniques for collaborative software development. ” O’Reilly Media, Inc.”
8. Milanesio, Luca. 2013. Learning Gerrit Code Review. Vol. 144. Packt Publishing.
9. GCC, the GNU Compiler Collection - GNU Project n.d. [https://gcc.gnu.org/.](https://gcc.gnu.org/) (Accessed on 09/06/2023).
10. About | IAR n.d. [https://www.iar.com/about/.](https://www.iar.com/about/) (Accessed on 09/06/2023).
11. Kremenek, Ted. 2008. “Finding software bugs with the clang static analyzer”. In: Apple Inc, pp. 2008–08.
12. Imtiaz, Nasif; Murphy, Brendan & Williams, Laurie. 2019. “How do developers act on static analysis alerts? an empirical study of coverity usage”. In: *2019 IEEE 30th International Symposium on Software*

*Reliability Engineering (ISSRE)*. IEEE, pp. 323–333.

1. Anderson, Charles. 2015. “Docker [software engineering]”. In: Ieee Software 32.3, pp. 102–c3.
2. Bellard, Fabrice. 2005. “QEMU, a fast and portable dynamic translator.” In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA, p. 46.
3. Bui, Thanh. 2015. “Analysis of docker security”. In: arXiv preprint arXiv:1501.02967.
4. Artifactory - Universal Artifact Repository Manager - JFrog n.d. [https://jfrog.com/artifactory/.](https://jfrog.com/artifactory/) (Accessed on 09/06/2023).
5. Test Framework — Zephyr Project Documentation n.d. [https://docs.zephyrproject.org/latest/develop/test/ztest.html.](https://docs.zephyrproject.org/latest/develop/test/ztest.html) (Accessed on 09/06/2023).
6. Hunt, John & Hunt, John. 2019. “Pytest testing framework”. In: Advanced Guide to Python 3 Programming, pp. 175–186.
7. Smart, John Ferguson. 2011. Jenkins: The Definitive Guide: Continuous Integration for the Masses. ” O’Reilly Media, Inc.”
8. Sayfan, Gigi. 2017. Mastering kubernetes. Packt Publishing Ltd.
9. How CI/CD-Integrated Fuzzing Improves Automotive Software Security

n.d. [https://www.code-intelligence.com/blog/cicd-integrated-fuzzing-](https://www.code-intelligence.com/blog/cicd-integrated-fuzzing-automotive-software-security) [automotive-software-security.](https://www.code-intelligence.com/blog/cicd-integrated-fuzzing-automotive-software-security) (Accessed on 09/28/2023).

1. Fuzzing: Challenges and Reflections n.d. [https://www.computer.org/csdl/](https://www.computer.org/csdl/magazine/so/2021/03/09166552/1mgaKsMFDYA) [magazine/so/2021/03/09166552/1mgaKsMFDYA.](https://www.computer.org/csdl/magazine/so/2021/03/09166552/1mgaKsMFDYA) (Accessed on 09/28/2023).
2. AFLplusplus/instrumentation/README.gcc\_plugin.md at stable · AFLplusplus/AFLplusplus · GitHub n.d. [https://github.com/AFLplusplus/](https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.gcc_plugin.md) [AFLplusplus/blob/stable/instrumentation/README.gcc\_plugin.md.](https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.gcc_plugin.md) (Accessed on 09/25/2023).
3. GitHub - mrash/afl-cov: Produce code coverage results with gcov from afl-fuzz test cases n.d. [https://github.com/mrash/afl-cov.](https://github.com/mrash/afl-cov) (Accessed on 09/26/2023).
4. Ubuntu Manpage: lcov - a graphical GCOV front-end n.d. [https://manpages.ubuntu.com/manpages/xenial/man1/lcov.1.html.](https://manpages.ubuntu.com/manpages/xenial/man1/lcov.1.html) (Accessed on 09/26/2023).
5. AFL Function Reference - PLOT n.d. [http://www.amibroker.com/guide/afl/plot.html.](http://www.amibroker.com/guide/afl/plot.html) (Accessed on 09/26/2023).
6. libFuzzer – a library for coverage-guided fuzz testing. — LLVM 17.0.0git documentation n.d. [https://llvm.org/docs/LibFuzzer.html#trophies.](https://llvm.org/docs/LibFuzzer.html#trophies) (Accessed on 04/02/2023).
7. fuzzing/tutorial/libFuzzerTutorial.md at master · google/fuzzing · GitHub

n.d. [https:](https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md)

[//github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md.](https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md) (Accessed on 09/28/2023).

1. Clang C Language Family Frontend for LLVM n.d. [https://clang.llvm.org/.](https://clang.llvm.org/) (Accessed on 09/28/2023).
2. Google Online Security Blog: AI-Powered Fuzzing: Breaking the Bug Hunting Barrier n.d. [https://security.googleblog.com/2023/08/ai-powered-](https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html) [fuzzing-breaking-bug-hunting.html.](https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html) (Accessed on 09/30/2023).
3. Fuzzing Example — Zephyr Project Documentation n.d. [https://docs.](https://docs.zephyrproject.org/latest/samples/subsys/debug/fuzz/README.html) [zephyrproject.org/latest/samples/subsys/debug/fuzz/README.html.](https://docs.zephyrproject.org/latest/samples/subsys/debug/fuzz/README.html) (Accessed on 09/24/2023).
4. zephyr/samples/subsys/debug/fuzz/README.rst at main ·

zephyrproject-rtos/zephyr · GitHub n.d. [https://github.com/zephyrproject-](https://github.com/zephyrproject-rtos/zephyr/blob/main//samples/subsys/debug/fuzz/README.rst) [rtos/zephyr/blob/main//samples/subsys/debug/fuzz/README.rst.](https://github.com/zephyrproject-rtos/zephyr/blob/main//samples/subsys/debug/fuzz/README.rst) (Accessed on 09/24/2023).

# AFL++ Target Program

The following is the detailed implementation of the target [program[249][250]](#_bookmark341) used in the fuzz testing with Zephyr apps by utilizing the [AFL++[49]:](#_bookmark140)

main.c

#### 1 #include <stdio.h>

#### 2 #include <stdbool.h>

#### 3 #include <stdint.h>

#### 4 #include <stdlib.h>

5

#### 6 /\* Fuzz testing is coverage-based, so we want to hide a failure case

#### 7 \* (a write through a null pointer in this case) down inside a call

#### 8 \* tree in such a way that it would be very unlikely to be found by

#### 9 \* randomly-selected input. But the fuzzer can still find it in

#### 10 \* linear(-ish) time by discovering each new function along the way

#### 11 \* and then probing that new space. The 1 in 2^56 case here would

#### 12 \* require months-to-years of work for a large datacenter, but the

#### 13 \* fuzzer gets it in 20 seconds or so. This requires that the code for

#### 14 \* each case be distinguishable/instrumentable though, which is why we

#### 15 \* generate the recursive handler functions this way and disable

#### 16 \* inlining to prevent optimization.

17 *\**

#### 18 \* https://github.com/zephyrproject- rtos/zephyr/blob/main/samples/subsys/debug/fuzz/src/main.c

19 *\*/*

20

21 **int** \*global\_null\_ptr;

22

23 *// crash*

24 **static const uint8\_t** key[] = { 0x9e, 0x21, 0x0c, 0x18, 0x9d, 0xd1, 0x7d };

25 **bool** found[**sizeof**(key)/**sizeof**(key[0])];

26

#### 27 #define LASTKEY (sizeof(key)/sizeof(key[0]) - 1)

28

#### 29 #define GEN\_CHECK(cur, nxt) \

#### 30 void check##nxt(uint8\_t \*data, size\_t sz); \

#### 31 void check##cur(uint8\_t \*data, size\_t sz) \

32 *{ \*

#### 33 if (cur < sz && data[cur] == key[cur]) { \

#### 34 if (!found[cur]) { \

#### 35 printf("Found key %d\n", cur); \

#### 36 found[cur] = true; \

37 *} \*

#### 38 if (cur == LASTKEY) { \

#### 39 \*global\_null\_ptr = 0; /\* boom! \*/ \

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 40 |  |  | *} else {* | |  | *\* |
| 41 |  |  |  | *check##nxt(data,* | *sz);* | *\* |
| 42 |  |  | *}* |  |  | *\* |
| 43 |  | *}* |  |  |  | *\* |
| 44 | *}* |  |  |  |  |  |
| 45 |  |  |  |  |  |  |

46 GEN\_CHECK(0, 1)

47 GEN\_CHECK(1, 2)

48 GEN\_CHECK(2, 3)

49 GEN\_CHECK(3, 4)

50 GEN\_CHECK(4, 5)

51 GEN\_CHECK(5, 6)

52 GEN\_CHECK(6, 0)

53

54

55 **int** main(**int** argc, **char** \*\*argv)

56 {

#### 57 /\*Matches key

#### 58 causes crash\*/

#### 59 //uint8\_t test\_data[] = { 0x9e, 0x21, 0x0c, 0x18, 0x9d, 0xd1, 0x7d };

#### 60 // no crash

#### 61 //uint8\_t test\_data[] = { 0x9e, 0x21, 0x0c, 0x18, 0x9d, 0xd1, 0x7e };

#### 62 // Open the file

63 **if** (argc != 2) {

64 printf("Usage: %s <input\_file>**\n**", argv[0]);

65 **return** 1;

66 }

67

#### 68 // Open the file

69 **FILE** \*file = fopen(argv[1], "rb");

70 **if** (file == NULL) {

71 printf("Could not open file %s**\n**", argv[1]);

72 **return** 1;

73 }

74

#### 75 // Determine the size of the file

76 fseek(file, 0, SEEK\_END);

77 **size\_t** size = ftell(file);

78 fseek(file, 0, SEEK\_SET);

79

#### 80 // Allocate memory for the data

81 **uint8\_t** \*test\_data = malloc(size);

82 **if** (test\_data == NULL) {

83 printf("Could not allocate memory for test data**\n**");

84 fclose(file);

85 **return** 1;

86 }

87

#### 88 // Read the data from the file

89 **if** (fread(test\_data, 1, size, file) != size) {

90 printf("Error reading file %s**\n**", argv[1]);

91 free(test\_data);

92 fclose(file);

93 **return** 1;

94 }

95

#### 96 // Close the file

97 fclose(file);

98

#### 99 // Use the data

100 check0(test\_data, size);

101

102 *// Clean up*

103 free(test\_data);

104 **return** 0;

105 }

106

CMakeLists.txt

1 cmake\_minimum\_required(VERSION 3.20.0)

2 project(sample\_zephyr\_afl)

3

4 option(USE\_AFL "Use afl" OFF)

5

6 if(USE\_AFL)

7 set(CMAKE\_C\_COMPILER afl-gcc-fast)

8 set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -g -fpic -fprofile-arcs

-ftest-coverage")

9 else()

10 set(CMAKE\_C\_COMPILER gcc)

11 endif()

12

13 add\_executable(zephyr\_aflfuzzer src/main.c)

1. **libFuzzer Target Program**

The following is the detailed implementation of the target [program[249][250]](#_bookmark341) used in the fuzz testing with Zephyr apps by utilizing the libFuzzer:

main.c

#### 1 #include <stdio.h>

#### 2 #include <stdbool.h>

#### 3 #include <stdint.h>

4

5

#### 6 /\* Fuzz testing is coverage-based, so we want to hide a failure case

#### 7 \* (a write through a null pointer in this case) down inside a call

#### 8 \* tree in such a way that it would be very unlikely to be found by

#### 9 \* randomly-selected input. But the fuzzer can still find it in

#### 10 \* linear(-ish) time by discovering each new function along the way

#### 11 \* and then probing that new space. The 1 in 2^56 case here would

#### 12 \* require months-to-years of work for a large datacenter, but the

#### 13 \* fuzzer gets it in 20 seconds or so. This requires that the code for

#### 14 \* each case be distinguishable/instrumentable though, which is why we

#### 15 \* generate the recursive handler functions this way and disable

#### 16 \* inlining to prevent optimization.

17 *\**

#### 18 \* https://github.com/zephyrproject- rtos/zephyr/blob/main/samples/subsys/debug/fuzz/src/main.c

19 *\*/*

20

21 **int** \*global\_null\_ptr;

22

23 *// crash*

24 **static const uint8\_t** key[] = { 0x9e, 0x21, 0x0c, 0x18, 0x9d, 0xd1, 0x7d };

25 **bool** found[**sizeof**(key)/**sizeof**(key[0])];

26

#### 27 #define LASTKEY (sizeof(key)/sizeof(key[0]) - 1)

28

#### 29 #define GEN\_CHECK(cur, nxt) \

#### 30 void check##nxt(uint8\_t \*data, size\_t sz); \

#### 31 void check##cur(uint8\_t \*data, size\_t sz) \

32 *{ \*

#### 33 if (cur < sz && data[cur] == key[cur]) { \

#### 34 if (!found[cur]) { \

#### 35 printf("Found key %d\n", cur); \

#### 36 found[cur] = true; \

37 *} \*

#### 38 if (cur == LASTKEY) { \

#### 39 \*global\_null\_ptr = 0; /\* boom! \*/ \

#### 40 } else { \

#### 41 check##nxt(data, sz); \

42 *} \*

43 *} \*

44 *}*

45

46 GEN\_CHECK(0, 1)

47 GEN\_CHECK(1, 2)

48 GEN\_CHECK(2, 3)

49 GEN\_CHECK(3, 4)

50 GEN\_CHECK(4, 5)

51 GEN\_CHECK(5, 6)

52 GEN\_CHECK(6, 0)

53

#### 54 #ifdef USE\_LIBFUZZER

55 **int** LLVMFuzzerTestOneInput(**const uint8\_t** \*data, **size\_t** size) {

56 check0((**uint8\_t**\*)data, size);

57 **return** 0;

58 }

59 *#else*

60 **int** main(**void**)

61 {

#### 62 /\*Matches key

#### 63 causes crash\*/

#### 64 //uint8\_t test\_data[] = { 0x9e, 0x21, 0x0c, 0x18, 0x9d, 0xd1, 0x7d };

#### 65 /\*Does not match key

#### 66 does not crash\*/

67 **uint8\_t** test\_data[] = { 0x9e, 0x21, 0x0c, 0x18, 0x9d, 0xd1, 0x7e

};

68 check0(test\_data, **sizeof**(test\_data)/**sizeof**(test\_data[0]));

69 **return** 0;

70 }

71 *#endif*

72

CMakeLists.txt

1 cmake\_minimum\_required(VERSION 3.20.0)

2 project(sample\_zephyr\_libfuzzer)

3

4 option(USE\_LIBFUZZER "Build with libfuzzer" OFF)

5

6 set(CMAKE\_C\_COMPILER clang)

7

8 if(USE\_LIBFUZZER)

9 add\_definitions(-DUSE\_LIBFUZZER)

10 set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -g

-fsanitize=fuzzer,address,undefined")

11 set(CMAKE\_EXE\_LINKER\_FLAGS "${CMAKE\_EXE\_LINKER\_FLAGS} -g

-fsanitize=fuzzer,address,undefined")

12 set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -g -fprofile-instr-generate

-fcoverage-mapping")

13 endif()

14

15 add\_executable(zephyr\_libfuzzer src/main.c)

16

17 if(USE\_LIBFUZZER)

18 set\_target\_properties(zephyr\_libfuzzer PROPERTIES LINK\_FLAGS "-g -fsanitize=fuzzer,address,undefined")

19 endif()

20