A Fuzz Testing Approach for Embedded Avionic Software

Master of Science in Technology Thesis
University of Turku
Department of Computing
Cybersecurity
2023
Leonardo Xompero

Supervisors:
Tahir Mohammad (University of Turku)
Jouni Isoaho (University of Turku)
Jürgen Grossi (Airbus Helicopters Deutschland)

Acknowledgements

First and foremost, I want to express my gratitude to my supervisors Tahir Mohammad and Jouni Isoaho, who assisted me with the development of the thesis and provided valuable advice; and Jurgen Grossi, who assisted me during my internship at Airbus, teaching me how life at the company worked and, more importantly, how life in Germany worked.

Another particular thank you goes to my family, who supported me not only during my university years, but also throughout my life. As a result, I'd like to express my gratitude to my mother Lilian, my father Marziano, my siblings Giancarlo and Michelle, my uncle Sincler, and my pets Wolfie and Cema.

I'd also like to thank all of my friends whom I had the pleasure of meeting during my university years. Simone and Pietro, thank you for staying up all night speaking on Discord, having fun, and playing games together. Giovanni, a friend with whom I had the pleasure of always chatting about our hobbies. Samuele, Alberto, Matteo, and Tommaso were buddies with whom I enjoyed making jokes. My fellow travelers of Ravintola Elo, Emanuele, Samuele, Cristiano, and Marco; without them, my time in Finland would not have been the most enjoyable experience of my life.

I’d also like to thank my flatmates from Trento, Paolo (a true Sardegna enthusiast), and from Germany, Xavier (for all the pizzas you cooked), and Marc (for showing me the German beer culture). Also, many thanks to all of my Airbus coworkers, Daniel, Giuseppe, Paulino, Trevor, David, Carlos, Fabrizio, Steve, and all the others that I forgot to mention with whom I had a wonderful time.

I’ll end with a love letter to science from my favorite game, "Outer Wilds": "I learned a lot, by the end of everything. The past is past, now, but that’s... you know, that’s okay! It’s never really gone completely. The future is always built on the past, even if we won’t get to see it. Still, it’s um, time for something new, now.".
I sincerely hope that this work will be valuable in the future; in the worst-case scenario, I had a good time.

UNIVERSITY OF TURKU
Department of Computing

Leonardo Xompero: A Fuzz Testing Approach for Embedded Avionic Software

Master of Science in Technology Thesis, 84 p.
Cybersecurity
December 2023

Fuzz testing is a technique that can be used to test software in order to discover potential flaws and vulnerabilities. This particular approach is receiving a quick widespread adoption to test also embedded software since there is a huge increase in these kinds of software. This adoption also includes the avionic field, where fuzz testing is currently used to test the software to ensure the robustness of the software and its compliance with the standards that regulate its behavior. Airbus Helicopters tried to research this approach in order to discover its potentiality, leading to the creation of this work, which will focus on researching the application of fuzz testing to embedded avionic software.

The objective of the research was to find if it was possible to apply the fuzz testing on an embedded avionic software by using available fuzz tools, more specifically available open-source fuzzing tools. Moreover, since the scientific literature does not provide guidelines on how to perform this approach towards this specific kind of software, this work will try to give an idea of how to apply the fuzz testing on a targeted avionic system, which in this case is a software component of a NH90 Airbus Helicopter.

The results of this study demonstrate that it is feasible to apply a fuzz testing approach to embedded avionic software, but only if the target code has undergone adequate preparations. If not, this approach may prove challenging to implement. Together with the suggested compilers and used software, it was also shown that the used components and measurements were appropriate in the fuzz testing application.

Keywords: avionic embedded software, fuzz testing, software testing, cybersecurity, embedded systems

Contents

1 Introduction

1.1 Background

In recent years, software in general has been employed within several kinds of different devices [1]. Because it operates inside an embedded device (or an embedded system), this type of software is known as embedded software. Nonetheless, there is a growing concern for the security of these particular devices since there is widespread adoption and implementation of embedded systems [2]. Like any software, several kinds of threats pose various risks to the devices and the environments in which they are used.

The avionics field scenario represents a field in particular that may be vulnerable to these attacks [3]. Because it is necessary to regulate the many aircraft functions and operations, embedded software is also utilized in this scenario to control the aircraft. Thus, it plays a crucial role in this specific type of embedded system. As a result, it's critical to test the software and give these systems' security adequate attention.

Since avionic software and embedded software, in general, might have similar issues, it was important to find out whether there were any other tools or techniques that could help programmers quickly find possible vulnerabilities in the software.

To find answers to these questions, companies and certified authorities looked into several possible solutions. These options included stricter standards to help developers produce robust software as well as software testing tools [4]. Researchers have studied and evaluated several tools and techniques that may be used for testing specific attacks, like pen-testing, or more random approaches, like fuzz testing. These techniques and tools can be applied in avionic scenarios for many different kinds of objectives.

These methods were eventually used in the avionic software development industry because of their excellent results (fuzz testing, for example, provides an alternative to more traditional methods like pen testing) [5]. However, the scientific literature did not properly provide research studies on the use of these techniques, particularly fuzz testing. As a matter of fact, a few companies have already begun using this approach [6], but they have not disclosed any particular requirements or potential guidelines on how to use this method of testing. Thus, research is required to determine the most effective approach for testing embedded avionic software through the use of fuzz testing, while also providing some insights into the key requirements and challenges that need to be addressed.

1.2 Research Problem

These days, fuzz testing is gradually becoming more widely used in the context of avionic software, where companies like AdaCore already provide this type of software testing. Additionally, an increasing amount of study is being done on this topic to improve its application in various scenarios.

However, there is little, if any, research on the fuzz testing method for avionic systems that provides an in-depth overview of the primary issues that this technique may raise or more information on the specifics of its implementation. It also means that most works don't provide guidance on how to handle fuzz testing for avionic software in general, which forces more companies to rely on proprietary software for testing when there is open-source software that might accomplish the same task for a lesser cost. In addition, there is no obvious indication of the use of a fuzz testing tool for an Ada program, which may yield intriguing outcomes but has not been investigated further.

We aim to provide a clear understanding of the issues surrounding avionic software testing and qualification, as well as how fuzz testing may be able to address some of these issues, by providing an overview of the subject in this work. A potential strategy for using an open-source fuzz testing tool could also provide further ideas for future projects, such as companies designing their own fuzz testing tool using an already-existing one.

1.3 Research Objective

The goal of this work is to identify and specify a potential method for applying fuzz testing to proprietary embedded avionic software using an open-source fuzz testing tool. By using a strong typing language like Ada, which is designed to protect against improper data entry and other potentially inadequate inputs, one may thus see how the fuzzing tool behaves. Additionally, the work displays the key findings of the main experiment to determine whether the software under test exhibits vulnerabilities or other issues.

By focusing on this, a strategy to employ an open-source software fuzz testing tool for complex software, like that found in avionic systems, may be outlined. Given that the tool can be further optimized and improved, this could serve as a reference for future works that wish to employ this kind of technique without depending on proprietary software. It could also enable a more customized experience with fuzz testing.

1.4 Research Question

To increase understanding of the fuzz testing approach and to provide suggestions for ways to make it better, the thesis will go into more depth regarding its background. There will also be more background information on the needs for the avionic software system and potential vulnerabilities, many of which are similar to those found in regular embedded software. Thus, the primary research questions are:

- Can fuzz testing be applied to avionic software? Is it possible to use open-source software to apply this approach?
- What are the primary prerequisites and tools required for conducting fuzz testing? What are the best practices?

Several studies were conducted to identify the answers to these challenges.

1.5 Scope of The Thesis

The thesis's scope is restricted to a couple of case studies that are judged to be plausible. There are minimal chances that our target system, a private avionic software, may be targeted by such a threat since there are a lot of constraints and protections behind the embedded system. Therefore, several considerations were taken into account in order to define the characteristics of our scenario, which in turn defines the primary risks associated with the avionic software.

Fuzz testing is an automated software testing method that tries to attack a selected system, in which a malicious insider introduces random inputs into the system in an attempt to identify vulnerabilities or potential issues. Since a hostile insider is the only type of attacker that may have access to the test tools used to test the software, which is the only method in which input can be injected, they emerge as one of the primary threats in this case study.

1.6 Thesis Outline

With open-source software serving as the primary tool, this study aims to clarify and give guidelines for performing fuzz testing on embedded avionic software. As such, the chapters are arranged and their contents are structured as follows:

The background of avionic software in general is covered in Chapter 2. Since avionic software is primarily within the embedded software subject matter, there will be a discussion on the history of embedded software in general as well as potential dangers and weaknesses. Following this, a more detailed explanation of the avionic software will be provided, along with an overview of it and the standards and requirements it must adhere to be deployed. A section on software testing will cover additional information on the threats to avionic software. This section will introduce fuzz testing, one of the most crucial aspects of this work, and explain its primary applications, methods of operation, and tools that are now accessible. Furthermore, studies on the use of fuzz testing in research papers and works will be presented.

The usage of fuzz testing in the chosen avionic system is discussed in Chapter 3. Since the primary objective in this instance will be an Airbus software component, further background information about the Airbus organization as a whole and the component's architecture will be provided. By doing this, we will be able to learn more about the main objective of the component, along with all of its dependencies and purposes, to approach this type of component and employ fuzz testing. Additionally, the test environment structure will be covered, together with the environment's primary features, which include the language, the compiler being used, and the fuzz testing tool.

Chapter 4 is the most crucial since it describes how to conduct the experiment, including how to use the fuzz testing tool and how to create important scripts and components. It also clarifies the primary reasons for some of the decisions made. To finally address the research questions, a discussion of the findings along with the display of some graphs will be given.

In the end, a review of the key discoveries and future works will be covered in Chapter 5. This chapter will provide useful information for future research projects that might benefit from the requirements and suggestions provided in this work.

2 Literature Review and State of The Art

2.1 Embedded Software and Security

Embedded software is a very broad field that has been growing throughout the years. This expansion is fairly rapid and apparent because the market for new embedded devices is expanding significantly these days due in large part to the growth of the Internet of Things, autonomous systems, etc. It is crucial to pay close attention to this expansion since it will result in an unprecedentedly large number of devices to manage as well as an increase in the likelihood that products may malfunction.

A lack of oversight during the product's design phase or certain errors from the previous iteration could be the cause of the potential problems. It's not only this problem, though. An increasing number of criminals will attempt to identify and take advantage of potential weaknesses and flaws that could enable an attacker to carry out malevolent actions against individuals or corporations as embedded software becomes widely used in more and more devices.

As a result, there is a need to improve embedded software security, which could enable these technologies to be protected even further. Nowadays, usually, cybersecurity requirements are satisfied during the design and production of the software, by using the methodology of cybersecurity-by-design [7], however, there could still be some possible vulnerabilities.

2.1.1 History and Definition of Embedded Software

Software that is specifically designed for a particular embedded system or device is known as embedded software [8]. This particular kind of software finds a huge amount of adoption in various kinds of devices, which range from the most common house furniture to more complex systems such as automotive systems. It is usually designed specifically for the hardware it runs on, including time and memory constraints.

It is essential to consider the hardware of the device when writing code for this software, as the software will have the duty to control the embedded device. Failure to do so would prevent several functionalities from working. Furthermore, the embedded software may be easily customized to work with any sort of device, no matter how big or small, including avionics systems and other systems. The ability of embedded software to manage and carry out certain operations without requiring user input is one of the primary distinctions between embedded and traditional software. The actual code that operates in the hardware is called embedded software and it leaves the management and control of the programs to operating systems [8]. Sometimes the embedded software doesn't use any of the common operating systems found on devices (Windows, Linux, Mac, etc.). An operating system that operates in real-time is required instead. The software is typically written in low-level programming languages like Ada, C, and Pascal [9]. However, high-level languages have also found employment recently, particularly in modern systems [10], but could still have some problems of performance.

Essentially, embedded systems are computer systems created to carry out specific functions and integrated into other goods or devices. Their employment is spread throughout multiple industries, including consumer electronics, automotive, and aerospace. They are built with a high level of durability and eficiency in mind [11].

Several different kinds of communication channels can be used to provide certain external controls. Furthermore, the fact that there are multiple ways to communicate with the device presents several ways to access the software for potential attackers. This will serve as the primary motivation for investigating the subject of "embedded software security", particularly in the avionic scenario, which will be explored in this work.

2.1.2 Threats

Assessing the risks and potential threats is crucial since embedded software is used extensively and plays a significant role in embedded systems. According to a survey [12], the embedded system may be regarded as a very sensitive area to safeguard because it requires stringent specifications and limitations, all of which must be met in real-time, to provide the desired outcomes. There could be some catastrophic consequences if the requirements are not fulfilled.

Still, there is an abundance of potential risks that could affect embedded systems in general. By examining CVE ratings associated with embedded system attacks, the authors of the paper [13] developed and presented a potential taxonomy of attacks for embedded systems. Five different sorts of criteria were used in the study to establish the attack taxonomy, and these criteria when combined provide a useful way to categorize potential assaults. Precondition, vulnerability, target, attack method, and attack outcome are these requirements.

The following prerequisites were discovered to be necessary to comprehend how the attack could be carried out for the first of the criteria:

- Internet access: arguably one of the most dangerous and common sources of attack. In this case, the device is directly connected to the internet, hence allowing potential malicious attackers to have access and exploit potential bugs and weaknesses. Since the access is via the Internet, the attacker doesn't need to have access privileges, since the attack could be carried out in other ways, the important part is that the device is detectable via the Internet.
- Local or remote access: similar to internet access, this type of attack involves the use of external services or communication over a network (could be public or private). But, in this case, the attacker must have access rights, otherwise, it is not possible to carry out the attack. The rights could be also normal privileges, it is not strictly necessary that they have to be superuser or administrator privileges.
- Direct physical access: on contrary to local/remote access, this type of attack is carried out only if the attacker has physical access to the device. The attacker doesn't need to have the privileges since the device could not have them.
- Physically proximity: this type of attack is quite similar to the previous one, but the particularity is that the attacker must be near the device, and could also not have physical access to it (e.g. Radio area with wireless device).
- Others: This category contains all kinds of attacks that do not re-enter the previous category. Moreover, this category will also contain those attacks that

do not require a precondition to be carried out.

The vulnerability is the next most significant category to be considered, after the preconditions. Embedded software may be vulnerable to various vulnerabilities that affect embedded systems as a whole, perhaps leading to misbehavior or other potential risks. Consequently, the following categories of vulnerabilities could be targeted by a malicious attacker:

- Programming errors: some problems during the code or just some mistakes could create some problems for the program, by creating logical errors that could be exploited to perform the attack (e.g., overflow attacks).
- Web based: since some devices could have a web interface used to perform operations such as update or configuration, they may be targeted by attackers since the web server application is rarely updated.
- Weak access control/authentication: this vulnerability is given by a lack of concern over the access security, such as having weak passwords (some also use the default ones), which allows the attacker to easily bypass the control thus gaining the access to the system.
- Improper use of cryptography: the use of cryptography can in some cases create some problems if not used in a good manner. Some bad practices in using cryptography usually consist of using deprecated methods, weak seed generators, etc.
- Other: this category contains all the other vulnerabilities that do not enter the other categories.

Since embedded systems have many targets, it’s critical to know which kind of layer will be attacked to quickly classify them:

- Hardware: this involves targeting the hardware part of the embedded device.

- Firmware/OS: there is no real distinction between the firmware and the Operating System because an embedded device could lack the presence of an OS and thus the only target could be the firmware, which provides the main functionalities of the device.
- Application: the target is the application contained in the OS of the embedded device.
- Protocol: the target is the protocols used for the communication of the device with the environment.

Embedded systems are vulnerable to several kinds of attacks, just like any other device. It's critical to classify the many ways that the earlier vulnerabilities have been exploited:

- Control hijacking attacks: the attacker successfully hijacks the flow of the program to execute some potential malicious code, which allows the attacker to take control.
- Reverse engineering: the attacker analyzes the code of an embedded device to "reconstruct" the code thus allowing them to find potential vulnerabilities to exploit.
- Malware: the attacker can infect an embedded device with malicious software, which has the property of executing a malicious act, creating harmful behavior to the device and the environment. Since the malicious behavior is quite unpredictable, it could create disastrous events.
- Injecting crafted packets/inputs: as previously stated, the protocols could be targeted by a malicious attacker, which could inject into some packets a malicious program created to perform malicious operations onto the embedded

device. The packet crafting could also create protocol failures thus allowing more exploits for the attacker.

- Eavesdropping: this attack is different from the others since it is more passive. Here the attacker simply tries to sniff the packet to capture some information that is passed with the messages between devices. This is usually due to weak cryptography protection or that is not protected at all. The intercepted information could be used later to perform a replay attack, where the packet is crafted using information obtained from the interception.
- Brute-force search attacks: if the device has weak protection or authentication, a brute-force attack could be viable. Usually, a generation of random keys to find the correct one could result in a successful attack on the weak authentication. This is only feasible if the search space is of small size, otherwise, it could be quite unfeasible.
- Normal use: the attack exploits a device without any dificulty since the device does not have protection, thus allowing the attacker to exploit the vulnerability easily.
- Others: these are the other attacks that do not enter the other categories or that do not exploit the previously stated vulnerabilities.

Attacks may put the targeted assets and systems at grave risk and have a variety of effects, from a minor service interruption to a significant financial loss. The following are some possible outcomes of these attacks:

- Denial-of-service: the attack converges in a disruption of the service, which leads to the unavailability of the service and potential problems in a system.
- Code execution: if the attacker successfully exploits the vulnerabilities of the embedded device, then they are capable of executing the code thus allowing

them to have more control over the behavior of the device.

- Integrity violation: some data on the device could be affected by the attack, which could be modified or deleted. Also, the code could be affected, resulting in a possible manipulation of the system.
- Information leakage: the data are compromised by the attack and thus there is a leakage of them.
- Illegitimate access: the attacker can gain access to the device without having the rights and privileges. A particularity of this case is that it is not only about the fact that can have access to the device, but it can also consider the fact that the attacker gains more privileges when previously had less of them.
- Financial loss: the victim of the attack suffers from a severe financial loss, which could be disastrous or light damage.
- Degraded level of protection: the attack successfully lowers the protection of the device, allowing the attacker to further compromise the device. This is possible due to a change in security policies or other methods.
- Others: this category contains all the cases that are not entered in the previous categories.

The categorization makes it feasible to classify an attack accurately and makes it possible for researchers to identify potential categorizations of the threats in order to define the appropriate protection. Moreover, it offers a quick overview of the threats that affect the embedded software scenario.

2.1.3 Possible Measure of Protection Against Threats

Since embedded systems are generally vulnerable to many threats, numerous studies have been conducted to try and identify possible protections for these kinds of systems. The challenge was dificult since it required protection that both used little resources and did not interfere with the systems' ability to function as intended. In reality, one of the biggest dificulties in creating a safe system is the limitations imposed by limited resources on the system's ability to protect itself.

Certain solutions are designed to defend against specific types of attacks, depending on the vulnerabilities, the target (hardware, firmware, etc.), and how the attack is carried out. A good example of this is the requirement to defend against so-called run-time assaults, which aim to compromise run-time systems [14]. Run-time attacks are a specific type of harmful conduct that targets the systems' run-time. They accomplish this by applying return-oriented programming (ROP) or introducing malicious code, which, when performed at run-time, enables the attacker to take control of the systems and exploit them. The basis of this attack is the interruption of program flow, which is then directed to the malicious code that the attacker has injected. This is because embedded systems typically use programming languages like C or assembly, which increases their vulnerability to run-time attacks.

As a result, the work [14] relies on attempting to defend against the attack by employing what is known as the Control-Flow-Integrity (CFI) check on the flow. The CFI makes it feasible to verify if a program is correctly executing the predetermined control-flow graph (CFG), or the usual flow that was decided upon when creating it. If an attack is suspected of causing a redirection or abnormality in the CFG during runtime, the CFI should be able to identify it through a flow check, enabling the attack to be mitigated. The drawback of this strategy is that it still relies heavily on the embedded system's limited resources, which might lead to performance issues.

Some works try to solve the problem by optimizing the CFI to reach the desired goals. A CFI that counteracts the ROP attack was proposed in the paper [15], which also suggests a design for an embedded system's "fine-grained, hardware-based CFI defense". The approach uses a hardware-assisted CFI framework to accurately implement the CFI policies. Given that runtime assaults provide a potentially harmful attack vector, it is necessary to contemplate the integration of CFI within embedded systems.

Certain works base themselves only on the idea that embedded systems are severely resource-constrained. The absence of support for memory management for embedded systems is discussed in the work of [16], where they suggest an approach based on the usage of passwords for system security. The proposal describes how to manage the concepts of protection context, which is the set of memory page access rights and protection domain, which is the collection of one or more protection contexts, by using a memory protection unit (MPU), which is the best supporter for this purpose. Furthermore, the MPU plays a key role in the embedded system by acting as a mediator between the CPU, main memory, and I/O devices. Since passwords and protection domains are strongly correlated, they were able to use the password holder through a process to activate a corresponding domain. It was possible to regulate an alteration of the domain composition by utilizing a collection of protection primitives.

Not only is it dificult to identify and investigate potential vulnerabilities, but it can also be dificult to implement new forms of protection. This is because, as of right now, relatively little research has been done to determine whether or not real-time embedded systems' existing protections are being used, or whether the algorithms for these kinds of protections are being widely adopted. To close this gap, a study is presented in [17] which aims to identify the protection strategies that real-time embedded systems use. Additionally, it looks into the kind of impact that security measures have on embedded systems.

2.2 Avionic Software and Standards

As was previously mentioned, embedded systems as a whole are rapidly growing and becoming widely used in a variety of industries, from basic home components to more complex mechanics like avionics. The main dificulty, however, is that the software used in these systems needs to adhere to strict constraints and respect them; otherwise, there may be harmful risks that result from improper function execution.

Due to the extensive use of these systems, increased security measures targeted at shielding embedded systems from malevolent attackers are essential to safeguarding not only the system's assets but also the environment that these systems may control or be a part of. One of the key issues in this situation is figuring out how to balance resource usage and security since the majority of embedded systems currently have complicated structures and few resources.

The avionics industry is one of the primary domains where addressing cyber threats is crucial. Even though the embedded avionic system, also known as avionic software, is often highly controlled and sophisticated, possible attacks may still be launched against it. It is essential to first understand the characteristics of avionic software, the limitations and requirements that set it apart, and the primary threats to it to come up with a suitable response to potential attacks. Furthermore, it's critical to comprehend how the legislation handles this, as there are specific guidelines that control how the requirements truly affect the software.

2.2.1 Overview of Security in Avionic software

In general, avionic software is a component of a vast and intricate system that, by providing additional benefits and features, enables an aircraft to operate as a whole. This software also serves to ensure the system's security and safety, making it easier to do the necessary inspections and testing to confirm its security. To ensure proper software development, ethical control over requirements, and safety checks, several standards and guidelines were established to verify these needs [18].

However, to achieve this, it's critical to comprehend the specifications required by an avionic system, as well as the potential challenges that may arise, including accurately identifying potential software flaws. Following this, to control safety-critical systems like avionics, it is also necessary to meet the requirements given by software/hardware certification.

2.2.2 Requirements for Avionic software

Given the critical role that avionic software plays in aircraft operations, it must meet numerous requirements and limitations to function effectively [19]. In the first place, they must be able to operate smoothly in real-time, conform to strict standards, and be dependable in high-pressure situations. Additionally, because they are made up of a collection of sensors, controllers, computers, and other types of displays, each part needs to be well-connected to the others and have the proper amount of space inside the embedded environment.

Basically, we can condense all of the requirements for an avionic system into three major areas. A portion of the inspiration for these domains comes from [20]. The study suggests a model-based method for communicating technical requirements at the chosen level of the system architect, utilizing the DEPS (DEsign Problem

Specification) language. In essence, DARPA has determined that there are four crucial stages in the avionics system design process, which are

1. Abstraction-based design tools
2. System complexity metrics
3. Advanced methods of architecture synthesis
4. Robust uncertainty management

This allows us to identify the three primary categories that specify the fundamental guidelines that avionic software must adhere to in order to be as reliable as possible:

- Safety requirements: there is a need to use the right standards for the aircraft in order to guarantee the appropriate safety and availability of the aircraft, which are defined in these standards.
- Security requirements: essentially, there is a need to have separate levels of security regarding each specific function of the aircraft, depending on the system levels.
- Capacity constraints: these are basically the requirements regarding the memory capacity of each CPU used in the aircraft.

Furthermore, because the avionic software is still an embedded system and has memory and capacity limitations, it is crucial to consider memory as a constraint. However, because avionic software is incorporated in many aircraft types, it is very dependent on the architecture of the aircraft; therefore, before establishing the limitations and the general needs, the architecture must be examined.

2.2.3 Standards for Avionic Software

It is crucial to adhere to a set of guidelines and standards to ensure that the avionic software can meet the criteria in a way that guarantees the required security. As a result, certain standards set by particular certification authorities such as the standard DO-178b which is applied to safety-critical software in aircraft systems, must be followed by the hardware and software that together make up the avionic system. Control for the requirements must be carried out during the software life cycle since the use of standards to qualify the avionic software is an essential component of the software. Because the software is continuously planned and verified, the avionic system's safety may be ensured in this manner. Furthermore, several administrations, like the Federal Aviation Administration of the United States, mandate it.

However, it's crucial to clarify and look into the features of the applied standard to have a better understanding of what the standards regulate. Upon examining the limitations and the standards' definition, it becomes evident that automated methods for testing software requirements are essential, given that avionic software design is not significantly complicated by the need to conform to standards.

DO-178B

As was already said, the DO-178B is the most significant one in the avionic software and is thoroughly described in the study [18]. This standard, published in 1992 by the Radio Technical Commission for Aeronautics (RTCA), is also known as "Software Considerations in Airborne Systems and Equipment Certification". The primary goal of the document is to provide principles and limitations for the development of aviation systems generally, with a focus on software development rather than hardware development, which is regulated by other standards. An increased level of reliable safety for the aviation system can be attained by fulfilling the standards found in the document. Following the software section, the document's primary focus is on the production processes. Its primary unique feature is its flexibility, as it is not specifically designed for any particular software cycle. The three primary categories that compose the software life cycle and the processes that are detailed in this document are as follows:

- Software planning process: focuses more on the planning of the processes involved in developing the program than on its technical aspects or its production process.
- Software development process: this topic mostly refers to software development design, which outlines the specifications for the software's needs, design, coding, and integration.
- Integral process: here, however, the focus is more on software verification together with configuration management, quality assurance, and evidence of adhering to the appropriate certifications.

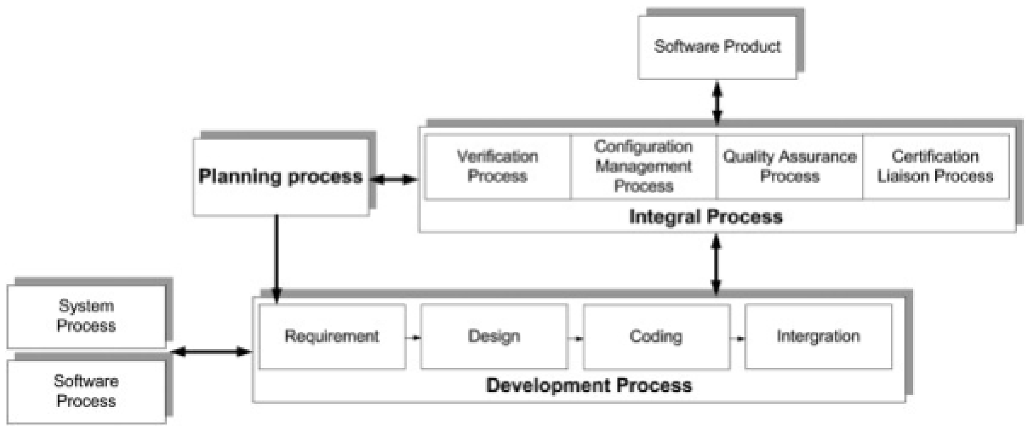


Figure 2.1: Schema for software life cycle presented by DO-178B [18]

The fact that the DO-178B presents the criticality of the systems at the system level by basing the system safety assessment procedure on failure circumstances (Catastrophic, Hazardous, Major, Minor, and No effect) is a further essential piece of information.

Furthermore, the certification explicitly states how the qualification tools must be used [18]. First and foremost, the tools are used to automate the manual tasks specified in the document by streamlining the requirements check of the program, which verifies that all limitations have been followed to by following the standards. The instruments specified in DO-178B must meet the following three requirements:

1. Tools have the ability to conceal defects from view or introduce new flaws into the product.

2. Processes could be simplified, automated, or removed.

3. The tool's output might not be independently validated.

If none of these conditions are met, no further tool qualification is required. Furthermore, neither the relevant tool history nor the design assurance level is taken into consideration by DO-178B for determining tool qualification.

As previously mentioned, the standard DO-178 has a significant impact on software development, and the importance of non-functional requirements varies according to the system's criticality. For instance, the DO-178 is largely utilized in the Airbus NH90 civilian model to identify these limitations [21]. These coding constraints, which are required for mission-critical code, include the following examples:

- The usage of dynamic programming techniques such as late binding or dynamic dispatch is forbidden.

- To guarantee error-free resource utilization and to estimate the amount of memory consumed at compile time, memory must be allocated statically.

- Any source code that is never executed in any configuration needs to be eliminated entirely.

- It is necessary for each source code line to have a low-level system requirement associated with it.

The aircraft can operate only when the limitations have been accepted by responsible engineering experts from a national certification authority.

ED-203A

Specific instructions for the security refutation objectives set by the European Organisation for Civil Aviation Equipment (EUROCAE) are contained in the document known as ED-203A, also named "Airworthiness Security Methods and Consideration". The term "refutation", as described in the document, refers to the "independent set of assurance activities beyond analysis and requirements [22]", therefore it refers to a sort of security evaluation of the analyzed object. It is therefore possible to refute the claim that there are exploitable vulnerabilities by meeting the definition of regulation. Additionally, the term "vulnerability" is explicitly specified here, stating that potential flaws or defects may also be identified as vulnerabilities. This is significant since there have been instances where maliciously capable vulnerabilities were not identified as such. Furthermore, even in cases that were thought to be settled and fixed, there remained a chance that new attacks would be launched against them as time went on.

Since the Airworthiness Security Process's Refutation phase is used to describe a system's security, find potential vulnerabilities, and assess how robust they are, a system is considered secure if the vulnerabilities are mitigated in one of the following possible methods [22]:

- Should the vulnerability originate from a software error, it has been addressed during the implementation process.

- Protected by a security measure, or set of measures. that are documented in the vulnerability dossier and appropriate to the level of danger to public safety associated with the exploitation of the vulnerability.
- Included in the vulnerability dossier and mitigated by a higher-level system in the aircraft platform as a whole.
- Recognized in the vulnerability dossier as a plausible possibility that the vulnerability won't be used.

If none of the vulnerabilities found fall into any of those scenarios, then they have to be a part of one of the following scenarios:

- There were no flaws or possible vulnerabilities found by the Refutation.
- Although the probable vulnerabilities were accurately detected, it was not possible to determine the applicable exploitation level that would have increased the likelihood of an exploit.
- Similar to the last instance, one of the vulnerabilities or weaknesses may be exploitable in this one as well, however, it is acceptable to take the risk of an exploited bug leading to a possible security breach or system security policy violation.

The document also describes the necessary testing and analytical tasks that need to be completed in order to correctly carry out the system's Refutation phase, which are as follows:

- Security penetration testing: a testing process that includes attempting to use a system vulnerability to read or alter the state of a recognized security asset.

- Fuzzing: an automated process that generates test cases and finds anomalies in the system. In order to generate extra test cases that are injected into the system to discover any abnormal behavior, fuzzing capabilities usually need modifying an initial corpus of test case inputs. Fuzzing is the process of identifying system inputs that can lead the system to enter an insecure state.
- Static code analysis: a semantic analysis performed at the source code level of the system to identify code constructs that may be considered unsafe or insecure.
- Dynamic code analysis: the analysis of how the system behaves when it is in action. Programming language semantics can be enforced through runtime constraint checks or dynamic code analysis using tools that find memory corruption problems through code instrumentation inserted during additional compilation steps.
- Formal proof: an explicit and limited series of formulas that can be statically verified with mathematical proof checkers (axioms, assumptions, rules, and inferences; often called contracts and assertions). In order to describe the anticipated behavior of the system, these models are generated both automatically by interactive theorem-proving tools and during software design.

There may be more refutation activities, but these are the most important ones and the ones that have already been implemented in the avionic field.

2.3 Software Testing

As seen in the previous section, avionic software must not only be protected against every potential threat to an embedded system, but it also needs to take into account how it conforms to all limitations and the various standards.

To ensure the avionic software's safety and compliance with standards, it is therefore essential to discover ways of testing it. This can be accomplished by employing tools or other techniques.

To ensure proper usage of the tool, one first needs to understand the potential issues and risks that could impact the system; some of these risks are comparable to those affecting embedded software as a whole. Furthermore, potential countermeasures for these vulnerabilities might offer some strategies for avoiding them, enabling even more testing enhancements. Examining potential examples that have previously occurred in this situation may also shed more light on how to solve the issue. The next step would be to investigate potential solutions that would meet the standards' requirements while also suggesting a testing ground for additional system security and safety improvements. To get started with fuzz testing, the main tool that will be examined in this work, potential methods for addressing this will be discussed..

In order to correctly understand the fuzz testing purpose in an avionic software, the behavior of the fuzz testing will be explained, together with how it works and how it is implemented. Moreover, a list of the state of the art of the various fuzz testing solutions will be explored. All of this will be important in order to understand how to define a fuzz testing approach on avionic software, which will be explained in the next chapters.

2.3.1 Threats to Avionics Software

Since avionic software is essentially an embedded software subcategory, it is vulnerable to similar attacks that take advantage of nearly identical weaknesses, leading to similar problems. Nonetheless, because avionic software offers more safeguards and controls than a typical embedded system, it may have additional hardware and software security layers. This reduces the likelihood of threats that target embedded systems in general and that can be applied to the avionic area.

Since attacks like spoofing, denial of service, exploiting, and counterfeiting are some of the threats that can be carried out against an avionic system, there are already many different types of research on avionic software generally in order to find solutions to mitigate attacks and potential threats to the aircraft [23]. As such, it is necessary to raise awareness of these issues. Several tests must be carried out, some of which may differ depending on the objectives, in order to identify potential vulnerabilities and reduce attacks. A variety of techniques, from the traditional pen test to the more modern fuzz test, can be used to carry out these tests.

Instead of focusing on determining whether attacks on the tested software are feasible, we will focus in this work on the testing part, which helps identify potential issues, as threats to an avionic system can also depend on the environment and the tools available to carry them out.

2.3.2 Fuzz Testing

The term "fuzz testing" originated in 1988 [24], where it was practically never used and it was a term that refers to simply random testing. Years later, additional research on automation techniques was conducted, and eventually, as time went on, companies began looking into this subject.

Fuzz testing, also known as fuzzing testing, is a software testing technique that generates an initial corpus by automatically creating test input data through pseudo-random mutation of user input data. This starting corpus makes it feasible to stress-test a specific piece of software in order to detect and understand any possible vulnerabilities that may be found as a result of these unexpected inputs.

The purpose of testing the system under these unusual operating settings and inputs is to determine whether it is suficiently strong to deal with threats of this nature. In fact, it is possible to confirm whether inputs caused crashes or other types of hangs during the fuzzing process. This makes it feasible to see how the system can transition between different states, some of which may be the undesirable states we are looking for in order to prevent potential vulnerabilities from being exploited. The main drawback of this technique is that it is not guaranteed that the implementation conforms to the functional specification with fuzzing, this means that to verify this a formal verification is needed.

In order for the fuzzing to properly work, it is important to first "instrument" the target, to understand what is happening inside the system during the testing [25]. This makes it easy to keep track of any crashes and hangs that may have occurred throughout the test, allowing one to replicate any potential vulnerabilities or attacks. To make the testing as autonomous as feasible, it can also be improved; for example, by building a test harness that has the ability to restart on its own to investigate more input combinations. There are also more improvements that can further enhance the performances and find more vulnerabilities [25], such as crash deduplication, crash triage based on suspected exploitability, and minimization of test cases.

2.3.3 Possible Uses

Desktop applications, generic libraries, and most recently, the avionics industry are among the domains where fuzz testing is rapidly becoming popular. An increasing number of companies or open-source libraries are attempting to use this technique to identify possible problems in their product code because of its ability to uncover a multitude of flaws and weaknesses. This is because it was demonstrated that using a fuzzer to discover possible vulnerabilities could achieve interesting results in various scenarios. For instance, the free software fuzzer AFL (American Fuzzy Lop) [26], was used to identify numerous vulnerabilities in a wide range of applications, including the iOS kernel, LibreOfice, and VLC.

Fuzz testing can also be applied in a variety of ways, greatly varying depending on the desired result. These methods differ primarily in that they include either black-box testing, which involves fuzzing a binary code in which the source code is inaccessible, or white-box testing, which involves fuzzing a binary code in which the source code is available and allows for potential code analysis. It's vital to remember that there is an alternative method as well, known as "gray-box testing", in which some of the source code is made available, providing additional information.

2.3.4 Additional Components for Fuzz

In the fuzzing tool, it is possible to further expand the capabilities of the fuzzer, by using external components that are able to amplify the code coverage of the fuzzer tool. In particular, in the case of AFL [27], it is possible to use additional components such as sanitizers and dictionaries, which are:

- Dictionaries: in order to further improve the performance of the fuzzer, it is possible to attach a "dictionary" [28]. The dictionaries are very useful since they provide a set of common words or values to the fuzzer which represents the expected inputs for the program. Therefore, adding a dictionary highly improves the eficiency of finding new units and performs really well in certain scenarios, such as the fuzzing of an SQL application [29], where it is dificult to stress-test by using a fuzzer since random mutations are unlikely to generate anything but trivially broken statements.

- Sanitizers: these are a set of tools that are designed to detect and report various types of programming errors and security issues in a program. They are initially developed to find possible problems in the code, therefore they assume a debugging role and can be applied to most languages to test their robustness. Thanks to these tools, it is possible to identify potential memory errors, like race conditions or other types of abnormal behavior. There are several sanitizers [30], which tasks can range from memory detector to race conditions detector. In AFL, these sanitizers can be applied when instrumenting targets for fuzzing [27], in order to discover possible bugs or other problems, but it has a huge impact on CPU and RAM usage.

2.3.5 Fuzz Testing in the Literature

To give a clearer overview of the primary findings of fuzz testing, many works in the scientific literature attempt to describe how and what kind of results it can achieve when applied to different applications. There is an increasing amount of research being done on embedded software in general, and these works investigate its applicability in various kinds of software. Sadly, no research has been done to explore this kind of use in the context of avionic software, or to put it another way, no research has been done to make it clear what kinds of strategies need to be used to utilize an open-source fuzz testing tool with avionic software.

In fact, the only fuzz testing approach that is applied to avionic software is provided by AdaCore, which offers the tool GNATfuzz [6] to offer an automated testing technique in order to detect abnormal and faulty behavior. This tool is capable of providing a fuzz testing approach for avionic software, however, it doesn't explain how the fuzz works and how it can be replicated by using open-source software, which is understandable since it provides proprietary software. It's important to note that this software presents the idea that avionic software can use AFL++ to do fuzz testing, as it employs the latter as a back-end fuzzing engine.

Considering how quickly fuzz testing was adopted, many works attempted to accurately assess this approach's capabilities in order to further testify of any possible drawbacks that this approach could cause. Researchers discovered possible issues with fuzz testing in [31], primarily because of the setup built around this approach's implementation, which led to potential issues in each evaluation of fuzz testing that was taken into consideration. Since an incorrect implementation of this could create misleading or wrong assessments, the work tried to include potential guidelines in order to achieve more robust results. Some of these possible issues resulted from not running the program multiple times or from timeouts that weren't justified. In order to produce more realistic results while improving suggestions for future work, some of these concerns were dealt with in this work.

Other works, such as [32], discussed the basic procedure and the main aspects to provide a more comprehensive understanding of the general features that classify fuzz testing. This work was taken into account since it provided insights into the primary issues and fuzz testing approaches, as well as a discussion of potentially useful fuzzers that are employed in well-known application areas. The paper provides further information on how to select amongst the different approaches while also providing a clear understanding of the primary fuzzing techniques that can be used for each particular kind of software (e.g., white-box and black-box fuzzing). Though we already had a very clear and defined aim for this project, it was still necessary to have some sort of explanation for using the chosen approach. Additionally, the article includes a review of potential fuzzers that can be applied to various target types. These were taken into consideration during the research process to find a viable fuzzer; ultimately, AFL++ was the preferred option. AFL++ is a better option because it already takes into account the features that were discussed as characteristics a fuzzer needs to have.

There are additional interesting studies that focus on fuzz testing's application to embedded software, which was important to this work. An overview of fuzz testing's use for embedded systems is provided in the work [33], which provides intriguing details on the methods that can be employed to fuzz an embedded system. The work covers potential fuzzers and goes into detail on the potentiality of employing fuzzing for embedded systems. It does not, however, address how it may be put into practice by providing some instructions on how to replicate and use the fuzzing for other types of embedded systems.

Other more specific works regarding fuzzing on embedded systems scenarios are [34] and [35]. The first one, which suggests a gray-box fuzz testing framework, provides a clear understanding of how to build a potential framework for using fuzz testing in the automotive industry to test specific autonomous car systems. The work produces a fascinating approach for applying fuzz testing to a car system, but it doesn't offer any general suggestions for applying this to other types of embedded systems or details on how the fuzz testing is actually carried out on the software.
The second paper, which is one of the few that genuinely shows step-by-step how the fuzz testing with AFL is carried out towards the target system, applies the fuzz testing to a different kind of embedded software scenario, specifically a Garming Datalink 90 protocol. Even though the work is primarily focused on possible DoS attacks against a particular protocol, it provides a clear understanding of the fuzz

testing process by enabling future studies to repeat the experiment and then look for additional system security-related information. Since this matches the goal of our work, it was an important paper to study.

Almost none of these papers attempt to apply a fuzz testing methodology to an Ada application. Given that the primary focus of this work is Ada-written avionic software, it was necessary to determine whether a fuzz testing strategy could be applied to an Ada program. The only work that attempted a similar strategy was [36], where AFL was used to test multiple Ada projects to identify probable issues and provided instructions on how to change the target source code to enable the application of fuzz testing. This work provides guidelines for setting up the target to be fuzzed by AFL and explains in detail how fuzz testing operates. The conclusion is that Ada is a useful example scenario to test using fuzzing because all of the implemented defensive code and runtime checks may be used as fuzzing targets. The key concepts that will be applied in this work are provided by the outlined approach.

3 Research Methodology

3.1 Airbus NH90 context

Among Airbus Helicopters' unique products, the Airbus NH90 is an extremely versatile aircraft that offers multiple options for customization. A component of this aircraft will be provided as the chosen target for the fuzz testing, which was kindly offered by Airbus Helicopters Deutschland.

3.1.1 Airbus Helicopters

Airbus Helicopters is a helicopter manufacturing division of Airbus, and it is the largest industry in terms of revenues and turbine helicopter deliveries. It was once known as the Eurocopter Group, but after merging with Airbus, it became an oficial Airbus division [37]. As a helicopter manufacturing division, it provides avionics products for many kinds of helicopters, from military to civil. Given the amount of products that Airbus produces, additional research and product improvements are constantly required.

The NHIndustries NH90, or NH90, is a modern multi-role rotorcraft that is one of Airbus Helicopters' primary products. It is built to meet and adhere to NATO specifications [38]. It is produced and supplied as a military helicopter in two variants: the NATO frigate helicopter (NFH) and the tactical transport (TTH). Additionally, the NH90 operates in many different kinds of scenarios because of its fully integrated mission system.

The helicopter is the result of a collaboration between Airbus Helicopters, Leonardo Helicopters, and Fokker Aerostructures, three distinct companies that specialize in avionics. This led to the development of a cutting-edge aircraft that could meet the standards and guidelines set by NATO and accomplish a wide range of tasks, including marine and inland missions. The fly-by-wire flight controls feature, which essentially replaces manual flight controls with an electronic interface to improve aircraft stabilization and adjust other characteristics in autonomy, was also introduced for the first time in production by the NH90 [39].

3.1.2 Variants and Characteristics

Since the NH90 is equipped with a fully integrated mission system, multiple versions of this particular kind of helicopter are available to provide a different approach depending on the operation for which the helicopter is needed. There are essentially two main versions that have been developed: the NATO frigate helicopter (code-named NFH) and the tactical transport version (code-named TTH). The versions were made in order to implement the aircraft's offer and give clients a wider selection.

The NH90 was initially intended to be constructed for four nations: France, Germany, Italy, and the Netherlands. It was intended to be used for all potential mission types, including land, air, and maritime operations. As a result, the expansion of variant types was brought due to a rise in different client desires. This is also the reason for the variants' numerous changes; nonetheless, certain components of the aircraft are significantly impacted by various types, which include [40]:

- Data Bus Framing

- MFD and DKU formats
- Varying set of application functions
- Dependencies between application functions
- Different operating environments

Since the same core vehicle is utilized in both versions, this shows it applies to both the TTH and the NFH versions [37]. This twin-engine rotorcraft core has essential characteristics such as aircraft monitoring, diagnostic systems, and a fly-by-wire control system with a 4-axis autopilot, which allows it to support several mission flights.

Furthermore, depending on the task, the aircraft may accommodate up to 20 fully equipped troops, providing a considerable volume for troop transportation. It's an excellent option for missions because it has a low structural weight and a high resistance to battle damage. Thus, the NH90 aims to accomplish the following three primary missions:

- Detection avoidance
- Self-protection
- Survivability

3.2 NH90 Architecture

The NH90's architecture is extremely complex since it must account for a wide range of scenarios that could occur during a mission. As a result, the design is rather complicated, and several parts work together to enable the aircraft to successfully complete the job.

Since the majority of these characteristics are kept secret by the corporation to not divulge sensible information to not trustworthy parties, the majority of information disclosed here is discussed in the work of Dordowsky [40] about the NH90 of Eurocopter. More of this information will be further improved with additional information obtained by the work on the project.

3.2.1 System Architecture

The NH90 is made up of two primary subsystems that work together to form the main system, which is what makes up the Avionic System Architecture. These are the two primary subsystems:

- CORE system: this is one of the subsystems that has the duty of managing the CORE functions, such as Vehicle Management, Communication, etc. This subsystem is managed by the CMC (Core Management Computer), which is the main component that will be studied and discussed in this work.
- MISSION system: this susbystems has the objective of managing the functions regarding the Missions of the aircraft. This subsystem is managed by the MTC (Mission Tactical Computer), which will not be discussed in this work.

Using various links, depending on the data that needs to be sent, these two subsystems are connected to their main computer. The primary types of connections are:

- MIL-STD 1553 bus [41]
- ARINC 429 line [42]
- Serial RS-485 line [43]

The main human-machine interface (HMI), which is made up of multiple distinct components, is unquestionably one of the other significant architectural elements that is especially significant for the CMC. These components include:

- Display and Keyboard Units (DKUs): as an interface for mission control and management between the pilot and the helicopter, the DKU is a keyboard-controlled electronic device with an LCD display [44]. The DKU uses communication via RS-485 to carry out the communication towards the subsystems.
- Multifunctional Displays (MFDs): usually, it is a screen with the ability to show different aircraft-related information, particularly flight or navigation data [45]. There are also possible keys that can be used for configuration.

3.2.2 Software Architecture

Due to its responsibility for accurately ensuring the operation of the components, software architecture plays a crucial role in aircraft design and must be of the highest quality. Therefore, each component that composes the architecture must be accurately designed and its role must also be defined.

The CORE and MISSION computers in the NH90 are controlled by a unique framework that was created to alleviate programmers to work on laborious and challenging tasks like data conversion, real-time scheduling, I/O handling, syntax errors, and various low-level errors, such as redundancy management, etc. This framework is known as NH90 Embedded System Software (NSS), and it interfaces with the Equipment Software (EQSW), which provides all the hardware-related functions. Both computers share the same platform, therefore there is only one NSS that interfaces with both of them. As we can see in Figure 3.1, the software architecture

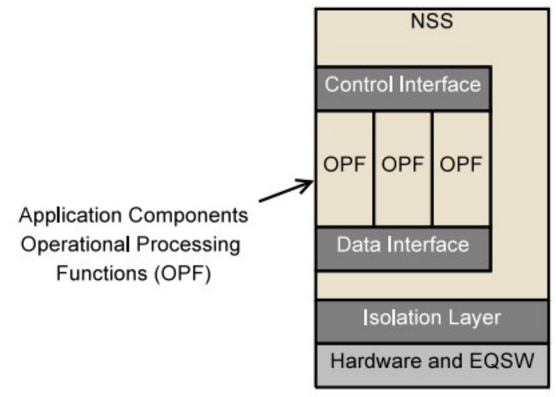


Figure 3.1: NH90 Main computers software architecture [37]

comprises several parts that perform different roles.

Operational Processing Functions (OPFs), which mainly include the mission functions and control for the avionic equipment, are another crucial component of the software architecture. These OPFs are heavily dependent on the CORE and MISSION computers because of their significant roles in the various missions. In addition, they depend on the helicopter variation because the mission and variant may require different types of functions. For this reason, the parts of the NSS instantiated in the CMC and MTC software are generated automatically to ensure that the limits imposed by the various rules are always respected. Although it is not recommended, this adaptation can also be made manually. The code generation is particularly useful since it provides the right connections between the NSS and the OPFs.

Always in Figure 3.1, we can see the Isolation Layer, which plays an important role because it creates a barrier between the hardware and the software, providing a kind of "cutting edge" where it is possible to separate the hardware and EQSW from the software and improve its modularity. This will play a significant role in the fuzz testing approach.

The DKU, which is located in the aircraft's cockpit, is in charge of interfacing the crew members with the two avionic subsystems and, therefore, with the CMC and MTC operational functions. In a nutshell, it is capable of receiving manual data entry and retrieving desired data, typically parameters related to crew commands, from the CORE and MISSION computers. Since the DKU is a machine on its own, its software architecture is quite different from the one that was explained, thus they will not be further explained.

3.2.3 Coding the NH90

Considering that the avionic system is basically an embedded software system, object-oriented high-level programming languages like Ada should be used to develop it. In order to obtain a high level of safety for the code and prevent the system from experiencing errors or flaws related to the type's issues, the language Ada is used. Ada was created especially for use in safety-critical embedded applications and real-time systems [21]. Ada offers several advantages, which include protected objects, synchronous message forwarding, explicit concurrency, and strong type. Ada's adoption of avionic software increased since the original design was made for an embedded real-time system. Due to Ada's rigidity, very little study has been done on the usage of fuzz testing for it. Additionally, since many fuzz tools are primarily made for other languages, using Ada brings additional challenges that are examined in this thesis.

3.2.4 Software Lifecycle NH90

The NH90 Systems must adhere to the software development processes indicated by the applied DOD-STD-2167A standard [46] in order to ensure a product that satisfies all functional but also safety and security requirements. Therefore, there is a need to use a common framework to work with and which can guarantee the satisfaction of the requirements.

Thus, the software development process is based on a framework known as 'V-Model' (symbolized by the letter 'V' due to its shape), which foresees on the left side the implementation activities (software specification, design, and coding) and on the right the verification activities (integration, testing at different levels, and ending with formal qualification testing). The procedure goes through the following phases, specifically for the NH90 [40]:

1. Software requirement analysis
2. Preliminary and detailed design
3. Coding
4. Host integration and testing
5. Target integration and testing

The Software must pass the Qualification tests in order to prove that the functional requirements are satisfied. Additionally, it must prove that the development requirements based on the applied standard (in this case DOD-STD-2167) have been met. The whole thing is submitted to the Customer (i.e., Qualification Authorities) who give its final approval. In regard to the design process, it is also critical to try to understand the likely inclusion of a future testing part, with a focus on fuzz testing, in order to conduct additional research into potential threats or software flaws.

3.3 Main Target

Once the general architecture of the NH90 is understood, the key components can be laid out in order to try to discover a possible approach to fuzz testing for the embedded avionic software of the NH90. As a result, this part will describe the primary target selected for the fuzz testing, along with some information about their key attributes and what will be essential to the test.

3.3.1 Core Management Computer

The Core Management Computer (CMC) is one of the key components that constitute the NH90 software architecture. In the NH90, all of the CMC and MTC Onboard Flight Resident Software (OFRS) components are combined into a single suite known as the NH90 embedded System Software (NSS). The CMC is a component of the TTH and NFH and provides a crucial role in the system, such as offering Operational Functions (CMC-OPF) to execute determinate tasks.

The NSS offers all of the CMC-OPS's shared services, and these services are computer-independent. Such is the executive, reusable components, redundancy management, rate monotonic real-time tasking architecture, and I/O drivers (e.g., Milbus 1553, RS-485, Arinc 429). Furthermore, all Core-related DKU I/O data is delivered to and received from the CMC computer via an RS485 Serial Line that connects the CMC computers to the DKU.

3.3.2 NSS Capabilities

The CMC uses the NSS as its common real-time operating system. The NSS manages all the various operational processing threads in addition to all the necessary system processing. We call these the Onboard Processing Functions (OPFs). For communication with the outside world and between OPFs, the NSS offers a variety of data interfaces. In addition to the range of functions that the NSS can carry out, it is critical to comprehend its role since it must be severed from the CMC in order to isolate the latter and enable the component to be "fuzzed". The NSS also is connected to the isolation layer, which is connected to the EQSW. The EQSW contains all the boards that are necessary for providing suitable processor parts and I/O part interfaces for the NH90-OFRS.

3.3.3 Issues and Threats to the System

The CMC component was the focus of the research and test. More research into potential threats or attacks was necessary in order to better understand the component's potential risks. Only a small number of potential risks will be highlighted without going into too much depth because the majority is confidential. Some of them are:

- Software attacks: because the program was developed in Ada83, it can resist numerous embedded attacks, the majority of which are focused on type exploitation or memory overflow. We shall talk more about these attacks in the section on fuzz testing.

- Physical attacks: a significant number of physical attacks were suggested to the company in order to assess how prepared it was for such situations. It was reasonable to conclude that with all the restrictions and clearance requirements, it was nearly impossible for an external attacker to carry out an attack on the helicopter, given answers that cannot be revealed for security reasons. Only in the unlikely event that the helicopter gets into the hands of a hostile attacker (usually during the combat scenario) might an attack be carried out. The unlikely situation of a malicious insider attack was also taken into consideration, regardless of its improbability.

It is important to note that, given the high level of protection of the company and of the helicopter components, the majority of attacks could be performed outside the avionic environment only if there are appropriate test benches and laboratories that are constructed to interface with this embedded software. That is also why a malicious insider attack was the one scenario that was considered.

3.3.4 Fuzzing Application

It can be dificult to apply fuzz testing to complex embedded systems, such as avionics embedded systems. Before the actual fuzzing begins, an extensive amount of analysis has to be performed. The CMC component, which is the software component that will be fuzzed in this particular case, must first be thoroughly studied. To precisely comprehend how to separate the CMC from the other components, it is necessary to investigate and study the architecture. It is possible to "construct" some sort of "wrapper function" after discovering the interactions that the CMC depends on. This is necessary to mimic a potential communication channel that the fuzzer may employ to inject the inputs.

All of this can be performed on a virtual machine, which is really convenient to use for testing and for repeating the experiment in different configurations. Additionally, using the virtual machine is preferable to causing unwanted issues in the workplace. Following the development of the wrapper functions, the isolated CMC may finally be connected to the fuzzer to enable fuzzing and thus, the identification of potential issues that are typically unnoticeable. The step-by-step process will be explored in Chapter 4.

3.4 Test Environment

The main experimental environment is rather complex due to the requirement for many applications and components that are not only necessary for the software to function effectively but also for understanding how to appropriately identify potential issues and operate the fuzzer.

3.4.1 Remote Machine Characteristics

The amount of data and components distributed across several systems created a somewhat dificult working environment for the tests. Specifically, the central code for the CMC component was located on a remote system that could only be accessed by a Unix Account having the necessary permissions to access the data. The primary operating system on the remote computer was Oracle Solaris, a proprietary Unix operating system that was created by Sun Microsystem and was just purchased by Oracle. Because it may be used with several other tools that are required in the software department, it is widely used in the company. The APEX Ada environment may be accessed within the system, giving the user access to the chosen CMC release type and version. From there, they can read and test the code.

3.4.2 Virtual Machine

A virtual machine was the ideal choice since it provided a completely functional working environment that would allow the experiment to be carried out multiple times with the exact same characteristics and therefore prevent any issues that could result from other factors. A neutral environment could be created with the virtual machine and modified as needed, such as in this case with the characteristics necessary for the fuzz, such as memory and libraries.

Ubuntu was chosen as the Operating System for the virtual machine mostly due to the vast amount of packages available and its complete compatibility with other libraries, which are mostly needed by the fuzzer AFL++ and GNAT.

3.4.3 Programming Language

The primary programming language of avionic software, particularly Airbus's embedded avionic software, is Ada83. The language is an excellent choice for programming in an avionic scenario, where an issue could cause disastrous scenarios and have grave consequences. It is an imperative, object-oriented, statically typed, structured, high-level programming language. Ada improves code safety and maintainability by using the compiler to identify problems rather than looking for runtime problems.

Thanks to the easier debugging due to strong typing and range checking and the ease of finding eventual flaws during the development [47], it makes an interesting case to be studied and to try to find a possible way to apply fuzz testing in such a rigid scenario. It is generally more different to implement fuzz testing in an Ada scenario than in a C/C++ scenario since the main focus of fuzzing testing is on the use of a corpus that is generated randomly from user inputs.

3.4.4 Compilers

Being a compiled language, Ada83 requires a suitable compiler to run properly, but it also requires specific instructions to apply the "instrumentation" to the object correctly and enable fuzz testing.
In order to correctly compile the Ada file, there is a need for two components:

- afl-clang/afl-gcc: serves as a code compiler and adds basic instrumentation to branch instructions. It is essentially an expansion of the traditional GCC compiler, with extra features added by AFL for target instrumentation. A fundamental block is defined as "a straight-line code sequence with no branches in except to the entry and no branches out except at the exit," and this is what the instrumentation is aimed at. Afl-clang also comes in several forms known as afl-clag-fast, which can enhance the fuzzer's performance in some cases. However, this work will not address it. By using the standard GCC to generate the assembly code that makes up the final output object, afl-gcc can be thought of as an Ada compiler to replace the traditional GCC. The instrumentation block that afl-gcc patches is typically applied to every jump instruction and label (jump destination) before your assembler is called to finish the compilation process. As GNAT is based on GCC, Ada can use GCC even though it generally uses GNAT as its compiler. This means that Ada code can be fuzzed using the assembly code produced by GCC. Gprbuild can be used with the appropriate "switch" to accomplish this.

- Gprbuild: a general-purpose build tool called GPRbuild is intended for building big multilingual systems that are divided into libraries and subsystems [48]. It works well with compiled languages that support independent compilation, such as Ada, C, C++, and Fortran, for this purpose. Three primary steps in the building process are managed by this tool: the compilation phase, in which each compilation unit of each subsystem is individually inspected, consistency checked, and compiled or recompiled by the relevant compiler; the post-compilation phase (binding), in which compiled units from a given language are passed to a post-compilation tool specific to that language; and the linking phase, in which all units or libraries from all subsystems are passed to a linker tool specific to the set of toolchains being used. Because the tool is generic, it can give equal build capabilities for all supported languages, which makes it a powerful tool that works well with afl-gcc.

3.4.5 AFL++

Of all the possible fuzzing tools that can be used, the one open-source tool that immediately results because of its simplicity and eficiency is without a doubt American Fuzzy Lop (AFL). This tool is a fuzzer developed by Michał Zalewski from the Google security team. Many vulnerabilities and flaws that were discovered in a variety of open-source libraries and tools have contributed to its success and rich history [49].

Specifically, we will use AFL++ for this test, which is an improved fork of the original AFL that offers faster execution times, more and better mutations, better instrumentation, support for new modules, etc. We chose this tool in order to find possible configurations for it to produce a better fuzzing approach for Ada, as it can give more functions with higher performances and has never been used to fuzz an Ada project. Given that AFL++ is built upon AFL, a mutational, coverage-guided fuzzer, it already has a number of capabilities that are essential for carrying out the fuzzing, including:

- Coverage Guided Feedback: this is a metric that provides information about the number of times in which an edge of the code (which was analyzed by the edge coverage) was executed. Thanks to this the fuzzer is capable of understanding if the input is considered interesting and thus putting it in the queue, which happens only if it explores a new part of an edge. These parts are stored in a bitmap to indicate the coverage of the code. Thanks to this feedback, it is possible to reduce the size of the test case, which allows for more speed and better performance.
- Mutations: there are two main types of mutation defined by AFL, which are deterministic and havoc. In the first case we have that single deterministic

mutations are performed on the contents of the test cases; while in havoc the mutations are randomly stacked and can change the size of the test case (by modifying its content).

- Persistent mode: this mode is capable of greatly improving performances by not applying a fork for each test case, instead it applies a loop into the target, which allows to have one test case per iteration, reducing thus possible bottlenecks.

Other features that are frequently included in fuzzer tools to improve eficiency include smart scheduling, which uses a variety of prioritization algorithms to schedule different parts of the fuzzing pipeline in order to maximize code coverage. Because fuzzers frequently produce a large number of incorrect inputs, features like the ability to mutate structured inputs might be helpful. For this reason, some tools, such as AFLSmart, can reduce the space of created inputs, thus making the creation of incorrect but right-structured inputs more feasible.

However, AFL++ further enhances these features by adding more components [50], which allows for more potential findings and more performances. Some of these features are:

- Seed Scheduling: the AFLFast is incorporated into AFL++, enabling more powerful schedules and further performance enhancement.
- Mutators: more mutators than the conventional ones, such as deterministic and havoc, are available. It is also feasible to use a customized mutator, which is simple to incorporate into the fuzzer.
- Instrumentations: AFL++ includes and supports a wide range of code-instrumentation tools, including LLVM, GCC, QUEMU, Unicorn, and QBDI.

This makes it possible to test various binaries and languages and to have more adoption and diversity in instrumentation types.

- Platform Support: the fact that multiple operating systems are supported makes it easier to apply fuzz testing in different environments.

Several features are presented in the documentation [51] and that will not be taken into consideration for the experiment.

Additionally, because fuzzing an Ada project requires a lot of work, it is essential to know how to approach an Ada project effectively and use the fuzzing tool accordingly. In Figure 3.2 there is a high-level representation of the architecture of unit-level fuzz testing for Ada [52].

3.4.6 Scripts

Some bash scripts were built to further automate all the fuzzing test procedures and to allow for some generalization. In this manner, future users might just modify a few variables to customize the fuzzer for a new project. The scripts are shown in detail in Chapter 4, together with a basic explanation of the main settings and the main tasks that they perform.

3.4.7 "Understand" Software

The stabbing part posed the biggest challenge to properly fuzz the CMC. This involved separating the CMC components from the NSS components and developing and implementing wrapper functions that could mimic this connection and maintain the CMC's normal behavior. Since it was not possible to automate this task for security reasons, it was possible to correctly stab a selected part of the CMC with the aid of the tool "Understand" [53]. This tool is an integrated development environment that can be customized, and it offers a variety of metric tools, documentation, and visualizations to facilitate static code analysis. This software allowed for the accurate understanding of program flows and the identification of the primary components (programs and libraries) that might be "cut" in order to replace them with a wrapper function, thereby freeing the program of its NSS and EQSW dependencies.

3.4.8 Environment Requirements

Given the abundance of tools and software present, this subsection will cover all of the primary requirements for each component, enabling replication of the experiment in the exact same settings. The Work Computer, which is the main machine that was used during the experiment, has the following specs:

- Processor: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
- Installed RAM: 32.0 GB (31.7 GB usable)
- Operating System: Windows 10 Enterprise 21H2 64-bit

Using Virtualbox, a virtual machine was installed on this machine to provide a fully customizable environment without requiring authorization to make changes to the files. The principal characteristics of the Virtual Machine are:

- Operating System: Ubuntu 20.04.6 LTS (Focal Fossa) 64-bit
- Base Memory (RAM): 16 GB
- Processors Allocated: 4
- Video Memory: 96 MB
- Graphics Controller: VMSVGA

- Network Adapter: Intel PRO/1000 MT Desktop (NAT)
- Shared Folder: 1, important to exchange data between the main working environment (so from Windows) to the Virtual Machine

Lastly, the AFL++ was configured with some characteristics that were used in other experiments and that were suggested by the company, thus the main specs used are:

- GCC version: 9.4.0
- GPRbuild version: GPRBUILD Community 2019 (20190517) (x86_64-pc-linux-gnu)
- Number of Threads used: 4 (one fuzzer per core)

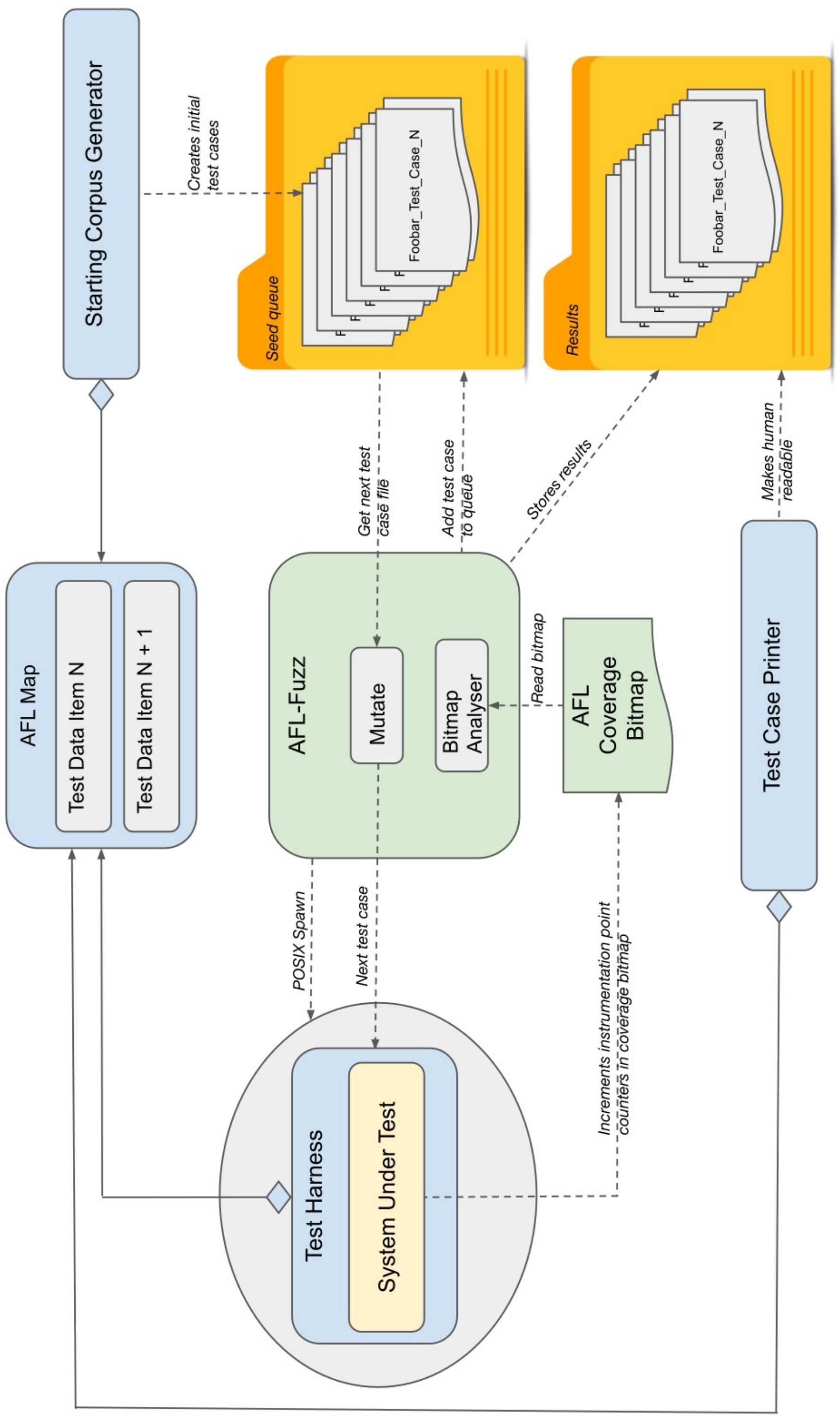


Figure 3.2: Unit Level Fuzz Testing Schema used for Fuzz an Ada Programs Architecture [52]

4 Implementation and Verification

The primary experiments, their methodical execution, and a brief discussion of the findings will all be covered in this chapter. In order to accurately comprehend the methodology and motivation behind some decisions, a few references to other works will be defined, accompanied by a relevant discussion on the individual choices. However, as the experiment was conducted on a code that is confidential and has some elements that cannot be shared, all of the missing portions will be described by utilizing aliases or by providing an abstract explanation of the part's principal purpose.

The chapter will begin with a thorough explanation of how the experiment was carried out, including how each of the necessary parts was assembled on the correct setup. After that, the test methodology will be clarified, along with the results obtained through the use of output screenshots and diagrams. Following that, there will be a discussion of the findings, with a particular emphasis on what the results are showing and how to interpret them. There will also be comments on how to potentially improve the results by speculating on what might work and what might not, which will provide readers with suggestions for improving future tests. In the end, an analysis of the successful and unsuccessful aspects of the experiment will be conducted, along with a reflection of what went wrong.

4.1 Implementation

This section will discuss the key steps that were taken to carry out the experiment, with an emphasis on the detailed process so as to provide a general idea of how to repeat the experiment in the future. Because the experiment was conducted using private code for an organization, the main process may be implemented differently depending on the tested code; nevertheless, the main environment and tools utilized remain fundamentally the same, allowing the experiment to be repeated and adapted to the chosen target.

4.1.1 Purpose and Objective

The purpose of using fuzz testing on the CMC, an avionics system component, is to identify any possible weaknesses and vulnerabilities that would enable a malicious attacker to provoke serious incidents or other attacks. Fuzzing avionic software is still relatively new as of right now, and those who have actually tried it haven't yet shared the techniques they used. Hence, another goal of this experiment is to provide an approach that would allow future projects to perform fuzz testing on their avionic software (and maybe other software) using open-source software (AFL fuzz). Airbus Helicopters, the firm that provided the test case, was granted access to the test's primary results, allowing them to repeat the test for other components in the future if they so choose.

4.1.2 Identifying the Target

In order to determine whether a fuzz method to find potential vulnerabilities to that component would be practical and produce positive findings without requiring a significant investment in resources, the company first picked the CMC component as the one component that potentially requires fuzz testing. Furthermore, the CMC was the part of the avionic system that would be easier to separate from the other parts in order to use the fuzzer, given that the system included multiple components.

The next challenge was figuring out which part could be fuzzed after the CMC was selected as the component to be fuzzed. Considering the extreme complexity of the code that constitutes the CMC, selecting only one library within the component was required. This is also partially because the fuzzer generates random inputs; however, the corporation felt that adding multiple random variables to large-scale projects like the CMC would be too dificult and would require too much time.

Therefore, rather than fuzzing the entire project, the initial approach to fuzzing was to find a suitable library inside the CMC that could meet particular conditions, which were chosen together with the company in order to produce results quicker:

- Number of dependencies: a library that is inside the CMC zone and has few dependencies outside of it is the one that has to be selected. The reason for this is that the library will be "disconnected" from its external dependencies so that the fuzzer may be attached to it, allowing it to analyze that specific component by introducing malicious inputs. It is the library with the fewest dependencies because many libraries have more than a hundred dependencies outside of the CMC (towards the NSS and EQSW).
- Number of variables: the library needed to have the fewest variables possible in order to reduce entropy when generating the inputs, which would enable the fuzzer to generate more potentially harmful inputs without wasting time or processing power on producing multiple incorrect inputs that might have been rejected earlier due to inadequate syntax. So, managing the entropy of the inputs' randomness was simpler when there were as few variables as possible.

- Complexity of the library: in order to avoid wasting a lot of computational power and time, a library that would not perform high computation performance cost was chosen after the complexity of the library was taken into account.

In the end, an OPF library within the OPS area was selected as the component; however, the purpose and name of the component cannot be disclosed, so the library will now be referred to as "FuzzedComponent" for the duration of this work.

4.1.3 Cutting FuzzedComponent

As a result of the FuzzedComponent's multiple dependencies on the EQSW and the NSS, it was necessary to disconnect it from both of them and take into consideration all of these dependencies. This was because, at a later time, these dependencies would be useful to create all of the wrapper functions that would mimic the dependencies and provide the program with a "normal" behavior.

Since the component could not be used in the Solaris environment (for clearance reasons), the only way to remove it from the dependencies without resorting to a script was to identify every one of them using the program Understand. This allowed for the identification of connections between components that could be severed and replaced with new parts that could mimic the behavior of the "cut" part. This made it possible to create new wrapper functions that would perform the same functions as the call to the dependency. This method made it easy to identify the changes that needed to be made inside the component in addition to isolating the OPF. After this, it was possible to move the selected FuzzedComponent to the VirtualBox environment, thus allowing us to modify it and continue with the experiment.

4.1.4 Wrapper Functions

The primary challenge, once the code has been chosen and the appropriate component has been found, lies in going further into the analysis of the primary dependencies and replacing these connections with variables or wrapper functions that can accomplish the same tasks without referencing the original component. This was the most crucial step since the component needs an entry point to receive data from the fuzzer, which will generate new inputs that have to be injected into it. Since the NSS and EQSW components in this case served as the entry points, all of the dependencies were changed with a wrapper function after the FuzzedComponent was isolated.

The term "wrapper functions" refers to functions that can actually receive a crafted data object containing the parameters that will be set inside the component. This allows the FuzzedComponent to function as normally as possible because it "believes" that it has received real data, even though it is actually a crafted packet created by the fuzzer. Thus, the following is the methodology for developing "wrapper functions" that are used to inject input data into the fuzzed software:

1. To begin with, you must accurately stab every external dependency from the library you wish to fuzz. It’s crucial to accurately identify every variable that might be exploited as a potential entry point to inject the input after carrying out the stabbing.

2. After having identified the variables, we can create a package used to store the data and its structure, which will be then referenced by the fuzzed library. To achieve this, we should create a file called "fuzzed_data.ads" that defines a package called "Fuzzed_Data", which contains the specification (so the variables) to create a record called "Input_Data_Record", that will simulate the data that has to be passed to the target. The package defines the type of vari-

ables and which ones are necessary to create a record, which will store the future inserted inputs; while the file "fuzzed_data.ads" is used to construct and build the package with the right values, by passing the "Input_Data_Record" as an argument for a new package called "Input_Data_File", which redefined the type "Sequential_Io". An example of this code is shown in 4.1.

3. Thanks to the package, is it now possible to also create an object of type "Input_Data_Record", which will contain some ad-hoc values inserted by the user. The file "generate_input_data_file.adb" will generate an object which will be used to initialize the values contained in "Input_Data_Record".

4. In conclusion, the last thing needed to do is to correctly bind these files from the libraries that need these variables. Inside there, all the variables that in reality should need input from the outside or from libraries that had to be deleted because of the stabbing, can be simply initialized by referring to the specific value stored in "Input_Data_Record".

5. Lastly, it is possible to correctly generate the record and pass the inputs to the right components in the main file, as shown in 4.2.

It was possible to replicate the receiving of the data object that the fuzzer would produce by feeding it with an object file as input for the FuzzedComponent. Because the FuzzedComponent required data from the NSS and EQSW, it would not have been possible for AFL to successfully create a malicious packet through random permutation of the inputs and inject it into the software (or at least it was not possible in short periods of time).

Listing 4.1: fuzzed_data.ads

```
with Sequential_Io;
package Fuzzed_Data is

    type Input_Data_Record is
        record
            -- list here as fields record all the input
        -- parameters of the procedure under test;
            Input1 : Boolean; -- originally from Library1
            Input2 : Boolean; -- from Library2
            Input3 : SomeType.Subtype; -- from Library3
            -- ... and so on
        end record;

    -- This is the container of the data which will be read
    -- by the procedure under test
    The_Data : Input_Data_Record;

    package Input_Data_File is new Sequential_Io
           (Element_Type => Input_Data_Record);

end Fuzzed_Data;
```

Listing 4.2: main.ads

```
with Fuzzed_Data;
with Text_Io;
with Generate_Input_Data_File;
procedure Main is
    Data_File : Fuzzed_Data.Input_Data_File.File_Type;
    Data_Record : Fuzzed_Data.Input_Data_Record;
begin
    -- procedure to create the fuzzed_data object
    -- (should be used only once to just create the object)
    Generate_Input_Data_File;

    -- Open the file
    Fuzzed_Data.Input_Data_File.Open
       (File => Data_File,
        Mode => Fuzzed_Data.Input_Data_File.In_File,
        Name => "prova",
        Form => "");

    -- Read The First (Only) record
    Fuzzed_Data.Input_Data_File.Read
        (File => Data_File, Item => Data_Record);

    -- Copy the record in the one which is used by
    -- the stubbed function
    Fuzzed_Data.The_Data := Data_Record;
end Main;
```

4.1.5 Scripts

It is necessary to make some decisions about the environment's parameters for the fuzz tool as well as any future dependencies or tools that may be needed (such as the compiler) in order to execute the fuzzer for the code. Some bash-programmed scripts were utilized for this project, allowing the virtual machine to have all of its primary functions implemented immediately. As a result, the environment was able to be made extremely modular. This is because altering certain variables inside the scripts files, rather than modifying other configuration files, was all that needed to be changed for fuzzer optimization. It was also necessary to change the files with the appropriate extension because there were other issues with the file extensions that were being used in the original environment (Solaris).

As a result, there were two main types of scripts that were required for either setting the operating system's functionalities or for fuzzing. These main categories are:

- Work Scripts: these scripts are the ones that are necessary in case of having older extensions for Ada files, such as ".1.ada" instead of ".ads". Since the former case does not work for fuzzing (because the fuzzer uses the new version of Ada), the script will change the extensions correctly, this can also be done for the body. Moreover, another script allows to change the name of the files that are constructed with a syntax such as "name.othername.1.ads". Since this could create a problem, it is mandatory to first change the extensions and then run this script. It is important to remember that this procedure is mandatory only in the case scenario that the avionic software is written with older configurations of Ada, and thus could create some problems for the fuzzer which uses a relatively new version of Ada. In case the system presents a more recent version of Ada, this step can be completely skipped.

- Fuzzing Scripts: contains all the scripts that are used to add a decent level of "modularity" to the fuzzing. In this way, it is possible to simplify the execution of all the necessary commands.

Because there are scripts that contain all the variables that were used to configure the AFL fuzzer, the "Fuzzin scripts" category plays a more significant role than the former, which can be mostly meaningless depending on the circumstance. Therefore, among all the scripts, the main one that was the most important was the "setup_fuzz.sh", which contains all the variables and all the functions used to correctly compile and launch the program, which is shown in 4.3.

Listing 4.3: setup_fuzz.sh

```
AFL_HOME=/home/user/AFLplusplus
AFL_FUZZ=$AFL_HOME/afl-fuzz

#will switch to a different schedule every time a
#cycle is finished
export AFL_CYCLE_SCHEDULES=1
#to disable the /proc/sys/kernel/core_pattern check
export AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
#disable the UI
export AFL_NO_UI=1
#afl-fuzz will terminate when all existing paths have been
#fuzzed and there were no new finds for a while
export AFL_EXIT_WHEN_DONE=1

#set the main GPR file from which it will refer
export GPR_PROJECT=fuzz
export EXEC_PATH=test
export TEST_PATH=fuzzing

#do the tests by passing the already created inputs
#and perform the fuzzing using 4 process
fuzz_test_program() {
    $AFL_FUZZ -D -x ./$TEST_PATH/dictionaries/$1.dict
    -i ./$TEST_PATH/input -o ./$TEST_PATH/output
    -L -1 -M fuzzer01 -- $EXEC_PATH/$1 @@
}

#compile the program with gprbuild
fuzz_compile_programs() {
    gprbuild -p --compiler-subst=Ada,afl-gcc
    -P $GPR_PROJECT
    --autoconf=fuzzing/fuzzing.cgpr
}

#debug the program by passing the same input that
#made the program crash
fuzz_debug_program() {
    gdb -ex "set args $2"
    -ex 'run' -ex 'bt' $EXEC_PATH/$1_test
}

#exectues all the crashed files on the program
#to see the errors
fuzz_triage_program(){
    for file in `ls -1 $1/fuzzer*/crashes*/*`; do
        if [ ! -d "$file" ] && [ "$(basename "$file")"
        != "README.txt" ]; then
            gdb -q -ex "set args $file"
            -ex 'set confirm off'
            -ex 'run' $EXEC_PATH/$2 -ex 'quit'
        fi
    done
}
```

There are four main functions in the script that are used to show and test the findings in addition to compiling and starting the fuzzer. These main tasks consist of:

- Compile: the primary goal of this function is to use the appropriate compiler and configurations to compile the application. The following subsection will go into more detail about this.
- Test: this function has the purpose of performing the test, which in other words means the execution of the AFL-fuzz tool, thus starting the real fuzz testing. As input it takes the AFL-fuzz compiler, thus it needs the relative or absolute path to it (in this case it uses the absolute since AFL-fuzz was not installed but only cloned from the GitHub repository). After this, it is possible to provide a dictionary path with the option "-x", which allows the fuzzer to automatically fetch all the content of the dictionary file. Consequently, there is the need to provide the input folder path, where all the initial cases that will be used as input to generate the starting corpus are stored, together with the output path folder, where all the outputs of eventual crashes found will be stored. Finally, there will be instructions regarding the execution of the

eventual "Master" and "Slave" process. This means that a Master process will start fuzzing the program, and the eventual Slave processes will explore different paths besides the ones that are already explored by the Master. Further information regarding this procedure will be further explained in a later subsection.

- Debug: this is a simple function that uses the same input that caused the program to crash to run GDB (GNU Debugger) on a specified executable file and gather extra debugging information.
- Triage: a function that runs through all of the program's crashed files to identify the problems that caused the crash and notify the fuzzer.

4.1.6 Compiler Specifications

As was previously mentioned, a few configurations need to be made before the application can be compiled using gprbuild. It is essential to use AFL-gcc to appropriately compile the program since the output object cannot be fuzzed since by default it is not correctly instrumented. This is because the code must be instrumented before applying the fuzz test. Furthermore, we need to explicitly tell GPRbuild that the project we really want to compile is an Ada project by using the line "–compiler-subst=Ada, afl-gcc," where afl-gcc is the primary compiler that will be used in this case. There is also the possibility of using previous configurations of the compilation process by using the "autoconf" parameter, but this will not be used for this experiment.

4.1.7 Parallel Fuzzing

Since each instance of afl-fuzz may operate on a single CPU core, it is possible to run many instances of the fuzzer for the same application simultaneously using the n-core resource, which will allow for faster results. When doing fuzz testing on software, the parallel fuzzing approach is a perfect choice because it uses the hardware to its fullest potential and rarely has performance issues.

Thanks to parallelization, it is possible to target several targets at the same time, but since this is not our case, we will concentrate on single-target parallelization. In order to do this, it is important to first create an output directory that will contain all the outputs of all the instances of AFL-fuzz. After that, when launching the instances at the same time, they must have a naming scheme for each instance (e.g., "fuzzer01"), where the first instance will assume the role of "master" (-M), while all the following instances will be "slaves" (-S). The main difference between the two modes is basically that the master will execute more deterministic checks, while the slaves will have a more random approach. In the case that the deterministic approach is not wanted, it is possible to use all the instances as slaves in order to have a more random approach. The primary difference is that each instance will routinely rescan the output folder, which is the top-level sync directory, to search for test cases found by other fuzzers. Each instance will run with a subfolder holding all of the outputs, just like a regular output folder. It will incorporate these into its own fuzzing if it determines that they are suficiently important.

Another option is to employ a multi-system technique, which enables us to use many machines to fuzz the same software, thus using additional cores. However, the multi-system parallelization won't be discussed because there aren't enough resources to implement this strategy.

4.1.8 Catching Errors with Exception

Since fuzz testing basically has the purpose of intercepting all the crashes that happen after the injection of a crafted input, it is important that the programs generate this crash by using exceptions, which allows us to catch and further understand what is the error and how it was generated.

Therefore, to catch the error, a simple top-level exception handler was used based on the work of [36], which is shown in 4.4.

Listing 4.4: setup_fuzz.sh

```
#The exception is needed to catch the rise,
#this should be put in the main file
#You can also filter unneeded exceptions
exception
    when Occurence : others  =>
    declare
      Text : constant String :=
      Ada.Exceptions.Exception_Information(Occurence);
    begin
      Put_Line ("exception occurred ["
      & Ada.Exceptions.Exception_Name(Occurence)
      & "] [" & Ada.Exceptions.Exception_Message(Occurence)
      & "] [" & Ada.Exceptions.Exception_Information(Occurence)
      & "]");
      GNAT.Exception_Actions.Core_Dump (Occurence);
    end;
```

For the exception to work properly with the environment, it is recommended to use the following libraries, which are used to catch the exception and to create the core dump: "Ada.Exceptions" and "GNAT.Exception_Actions". Furthermore, additional exception handlers were put in the dependencies in order to propagate possible problems.

4.1.9 Test Cases

The initial input must be properly constructed for the program to function properly; otherwise, the fuzzer won't be able to generate further cases because the first case has already crashed. Since the executable needs to receive a test case that functions properly and can produce accurate results, it is necessary to create one. Two files can be created to accomplish this:

- File for Input Data Record: this file contains a package containing a defined type, which is a record, that contains a list as a field record for all the input parameters of the procedure under test (in this case FuzzedComponent). At the beginning, each field will have a defined datatype and value, where the value will be given when generating the file. For example, in the record, there will be a variable called "Object_Available" of type "Boolean", which will replace the variable in the dependencies connected to the FuzzedComponent where the real value would be passed through other means, but since the component was isolated from all the dependencies, the value will be passed in this way.
- File for generating Input Data File: this file contains a procedure that defines the Record Type defined in the previous file, and here the input file will be created with defined values for all the parameters contained in the record. It is important to note that the defined values are created only in the first creation of the file (so for the initial input case, the so-called starting

corpus), while in all the later interactions of the fuzzer, all these values will be generated by the fuzzer.

Finally, by using the appropriate procedure, we can generate the first file (which will serve as the initial input case). Afterward, we can incorporate the procedure into the main file to open the input file and accurately parse its contents, which will then be passed to the appropriate dependencies and used as the value for the variables that were initially stabbed. This will allow the input file to mimic the proper data flow between the dependencies and the FuzzedComponent.

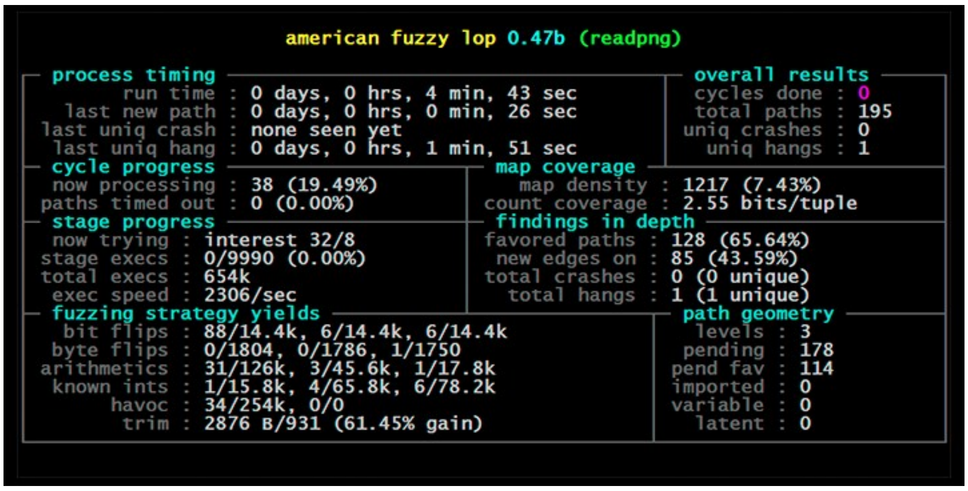
4.2 Verification

This section will describe the process of applying the fuzzer to the program. It will provide an explanation of the specific parameters that were used, along with a rough idea of the time required to confirm the existence of errors and problems. Moreover, there will be several graphs that will show the fuzzer's performances at runtime, together with additional information regarding the crashes found and the behavior of the fuzzer.

4.2.1 Using AFL++

After all the preparations for the fuzz testing are done, it is possible to start the real fuzz testing by executing the scripts and thus starting the process. First of all, the program must be compiled with gprbuild, by using afl-gcc as a compiler in order to instrument the code so that the executable can provide code coverage information. This is mandatory otherwise the fuzzer won't be able to work with the provided executable.

After that, it is possible to start AFL-fuzz with the obtained executable from the previous compilation, passing in input all the test cases, which in this case will be created by us by compiling the relevant file with the procedure for the file creation. By launching afl-fuzz, usually a UI of the fuzzing will be displayed, such as shown in Image 4.1. Here it will be shown several pieces of information, which span from the run-time of the fuzzer to the number of crashes found until that moment. Since the



```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec        │  total paths : 195    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec        │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)        │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)          │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8          │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)         │  new edges on : 85 (43.59%)            │
│ total execs : 654k                   │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec               │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k             │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750                 │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k              │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k              │  imported : 0          │
│       havoc : 34/254k, 0/0                           │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)               │    latent : 0          │
└──────────────────────────────────────────────────────┴────────────────────────┘
```

Figure 4.1: AFL standard GUI [26]

fuzzer is automated, the only thing left to do is to leave the fuzzer on its own and wait for possible findings. In the UI there are several information shown, which can be quite important to understand what is happening at the moment and to further understand how to possibly improve the fuzz session. Briefly, these information are:

- Process timing: in this section, there is the main information about the runtime of the fuzzer, which means the duration and how much time has passed since the last crash was found. There is also information about the possible findings of new paths and hangs, which allows us to have a clear idea about whether the fuzzer is correctly working or not. Usually, fuzzy testing can run for more than one day, and it also can arrive for months of runtime, but it is important to keep this into account if the fuzzer is finding new paths, otherwise it is considered a waste of time.

- Overall results: this section provides more data regarding the main results achieved by the fuzzer, meaning the number of cycles performed, how many paths were found, and, most importantly, the number of crashes achieved.
- Map coverage: here there is information about the coverage observed by the instrumentation embedded in the target binary. With the map density is possible to observe how many branches were already searched, by using a bitmap proportion to indicate this measure.
- Stage progress: gives more in-depth information about what the fuzzer is currently doing, which can be several types of operations (e.g., trim, bitflip, artih, etc.). The fuzzer will also warn you in case that the target is really slow.
- Findings in-depth : this shows the exact number and data regarding the main findings.

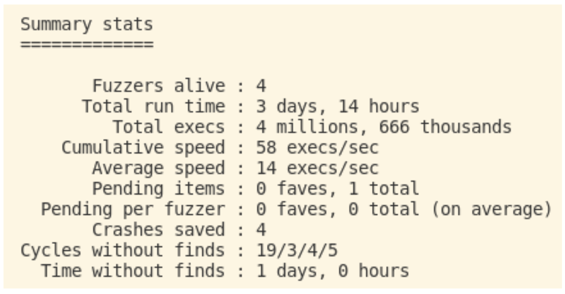
However, since the GUI puts quite a toll on the performance of the machine, it was disabled for this experiment, but it is important to understand all the information that AFL can provide. To further customize and thus improve the afl-fuzz, there is also the possibility to use environmental variables to find several functions that could be useful. Some of these variables were used inside the script for running the fuzzer and can be directly changed there.

4.3 Results and Discussion

This section will provide the most important graphs of the most successful tests conducted in order to illustrate the main findings from the experiment. To give a broad perspective of how to improve the work, more information about the experiment will be given together with an in-depth discussion of the most important choices made throughout the work's execution. Finally, a comprehensive analysis of the outcomes will be shown, aimed at summarizing the principal discoveries and the successes achieved from the study. This will be helpful for any future projects so that the work can be expanded even further and more settings may be optimized.

4.3.1 AFL results

A summary of the main results obtained at the end of the experiment is shown in Figure 4.2. The figure illustrates the main information regarding the achievements of the fuzzer, ranging from the number of crashes found to the total runtime of the fuzz testing.



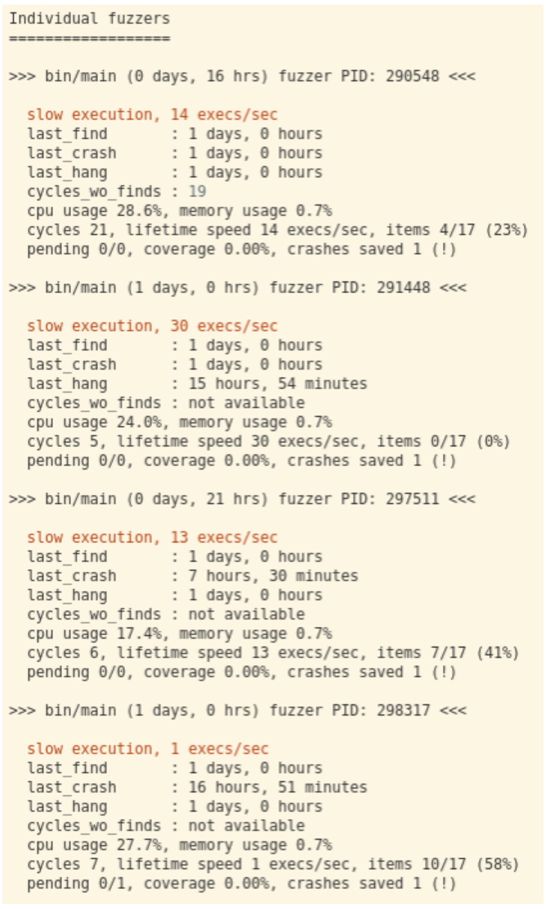
```
Summary stats
=============

       Fuzzers alive : 4
      Total run time : 3 days, 14 hours
         Total execs : 4 millions, 666 thousands
    Cumulative speed : 58 execs/sec
       Average speed : 14 execs/sec
       Pending items : 0 faves, 1 total
  Pending per fuzzer : 0 faves, 0 total (on average)
       Crashes saved : 4
Cycles without finds : 19/3/4/5
  Time without finds : 1 days, 0 hours
```

Figure 4.2: AFL Summary

The fuzzer was left active for several hours, waiting for it to find possible vulnerabilities thus creating a dump of the crash and further completing the queue of test cases generated by AFL. This resulted in a total run-time of more than three days (roughly 3 days and 14 hours), performing more than four million execution of the fuzz testing (4 million and 666 thousand executions circa), shared between the four main fuzzers alive (1 master and 3 slaves).

We achieved an average speed of 14 executions for seconds, which could be influenced by the computer performances and the several fuzzer actives, nonetheless, it was possible for each fuzzer to complete several cycles of the code and to find all the possible paths. The most important decision for the experiment was to decide when to stop the fuzzer, otherwise, it would continue endlessly despite the nonpresence of more things to discover. Therefore, it was decided to stop it after at least 24 hours of inactivity, which in this case corresponds to not finding any new crashes or possible paths.

In the end, the fuzzer found 4 crashes in total, which will be discussed later in the chapter. For further details about each fuzzer that was active for the experiment, we can see Figure 4.3. Here it is shown more details about the behavior of each fuzzer during runtime, in order to give an example of how it behaves while searching for possible crashes. Thanks to these results, it was possible to find some problems that



```
Individual fuzzers
==================

>>> bin/main (0 days, 16 hrs) fuzzer PID: 290548 <<<

  slow execution, 14 execs/sec
  last_find       : 1 days, 0 hours
  last_crash      : 1 days, 0 hours
  last_hang       : 1 days, 0 hours
  cycles_wo_finds : 19
  cpu usage 28.6%, memory usage 0.7%
  cycles 21, lifetime speed 14 execs/sec, items 4/17 (23%)
  pending 0/0, coverage 0.00%, crashes saved 1 (!)

>>> bin/main (1 days, 0 hrs) fuzzer PID: 291448 <<<

  slow execution, 30 execs/sec
  last_find       : 1 days, 0 hours
  last_crash      : 1 days, 0 hours
  last_hang       : 15 hours, 54 minutes
  cycles_wo_finds : not available
  cpu usage 24.0%, memory usage 0.7%
  cycles 5, lifetime speed 30 execs/sec, items 0/17 (0%)
  pending 0/0, coverage 0.00%, crashes saved 1 (!)

>>> bin/main (0 days, 21 hrs) fuzzer PID: 297511 <<<

  slow execution, 13 execs/sec
  last_find       : 1 days, 0 hours
  last_crash      : 7 hours, 30 minutes
  last_hang       : 1 days, 0 hours
  cycles_wo_finds : not available
  cpu usage 17.4%, memory usage 0.7%
  cycles 6, lifetime speed 13 execs/sec, items 7/17 (41%)
  pending 0/0, coverage 0.00%, crashes saved 1 (!)

>>> bin/main (1 days, 0 hrs) fuzzer PID: 298317 <<<

  slow execution, 1 execs/sec
  last_find       : 1 days, 0 hours
  last_crash      : 16 hours, 51 minutes
  last_hang       : 1 days, 0 hours
  cycles_wo_finds : not available
  cpu usage 27.7%, memory usage 0.7%
  cycles 7, lifetime speed 1 execs/sec, items 10/17 (58%)
  pending 0/1, coverage 0.00%, crashes saved 1 (!)
```

Figure 4.3: AFL Fuzz Details

could give a crucial insight into the behavior of an avionic software when performing under stress test caused by a fuzzer. This further confirms some hypotheses on the behavior of a fuzzer towards an Ada program and gives an idea of the expected performances.

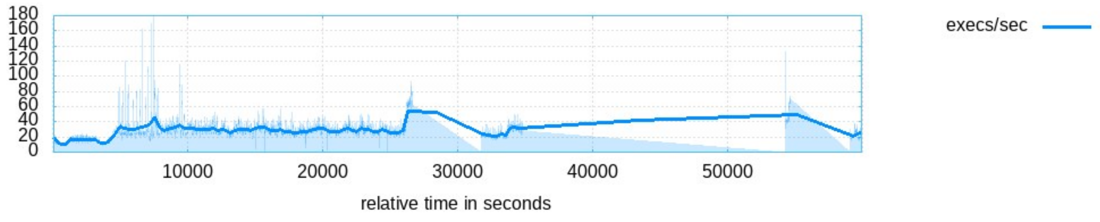
4.3.2 Graphical Analysis

These graphs were obtained by using afl-plot and it contains the main graphs regarding the performance of each fuzzer during the fuzz testing. There are 4 plots in total, and each one of them indicates the behavior of the fuzzer by giving details such as the edges discovered, the execution speed of the fuzzer, and the frequencies with which each fuzzer generated items and found possible crashes.

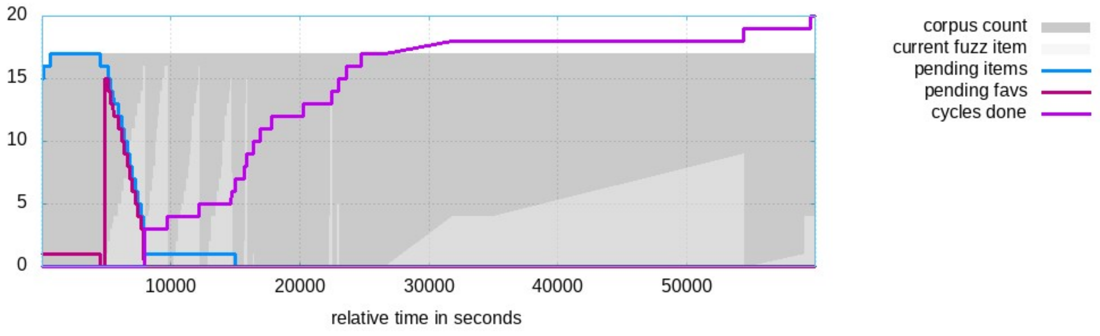
The plots are shown in Figure 4.4, 4.5, 4.6, and 4.7. It is noteworthy to mention that certain graphs might show sudden highs and lows at the end of the graph (e.g., 4.5b). It is unclear whether this is because the graph generator (afl-plot) made an error when it was being executed, or if the VirtualBox environment exhibited abnormal behavior that could have impacted the fuzzer's performance during run-time. More likely, the sudden spike was caused by the forced program interruption at the end of the experiment (essentially shutting down the fuzzer), which could explain why the spike is only present in the slave processes. Therefore, it might be beneficial to investigate graph generation in the future. The plots in this work are presented merely to provide a general notion of how the performances should appear; nevertheless, this may vary depending on the platforms and targets.

4.3.3 Discussion

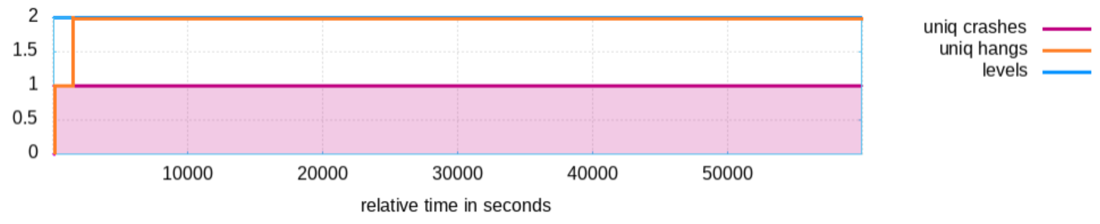
Few system crashes were discovered as a result of the fuzz testing, which made it possible to examine the component and system and determine whether or not a fuzz



(a) Execution Speed

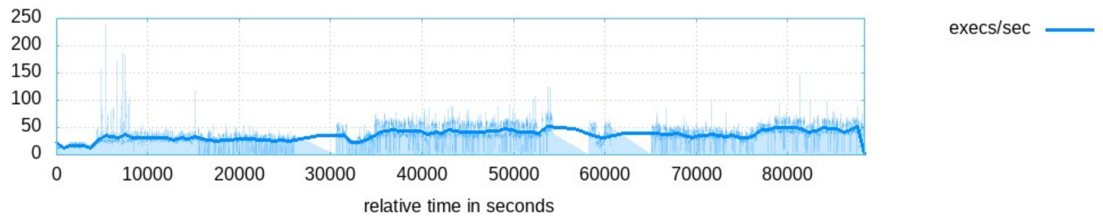


(b) Code coverage and pending items

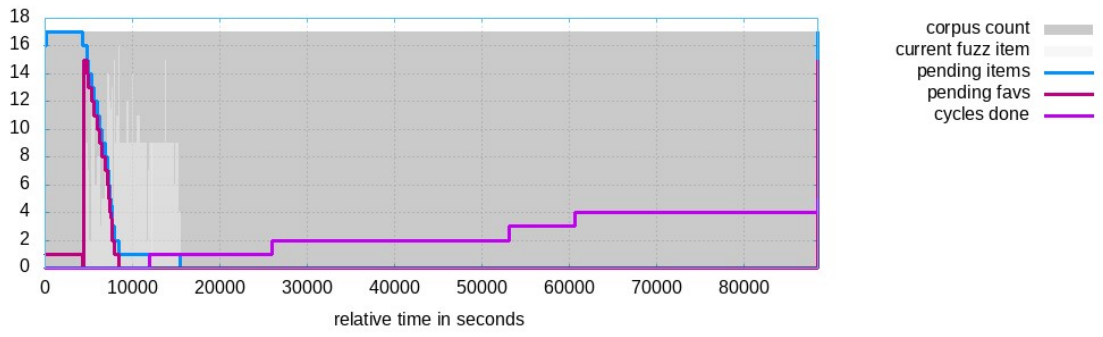


(c) Crashes and Hangs

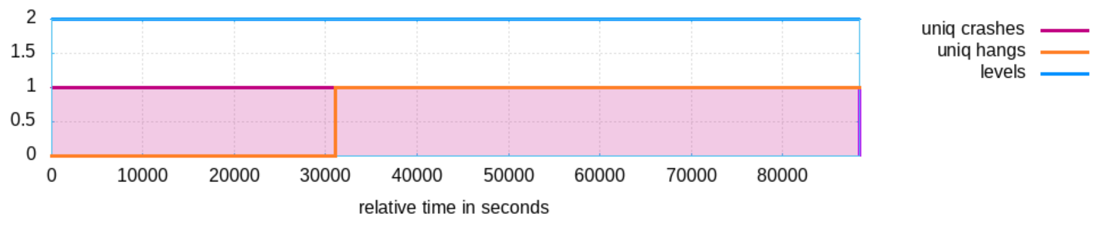
Figure 4.4: Plot of Master Process. a) Corresponds to the number of tests executed over time b) The plot shows the performance of the Master process during runtime while showing how many execution paths were found. The information shown is: the corpus count (set of inputs for a fuzz target, showing the number of test cases that will be used as a template to generate new inputs), the current fuzz item (shows the number of execution paths found inside the set of test cases, therefore the newly generated inputs), the pending items, and favored items (inputs that still did not get through the fuzzing, favored have more priority for the fuzzer), and cycles that are done. c) Crashes and hangs found during the fuzzing, where levels represent how much the inputs have derived from the initial test cases.



(a) Execution Speed

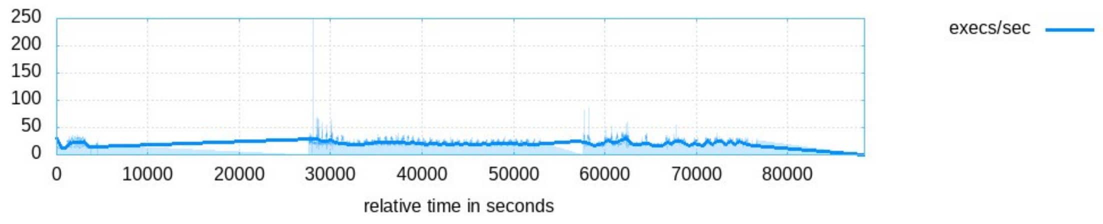


(b) Code coverage and pending items

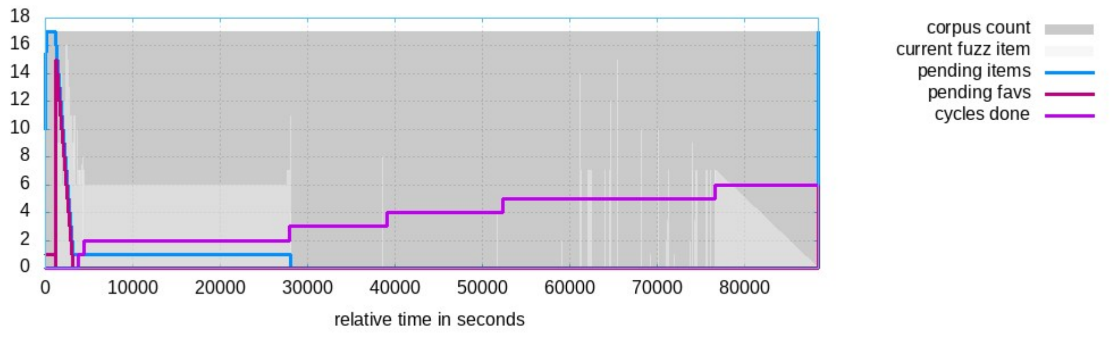


(c) Crashes and Hangs

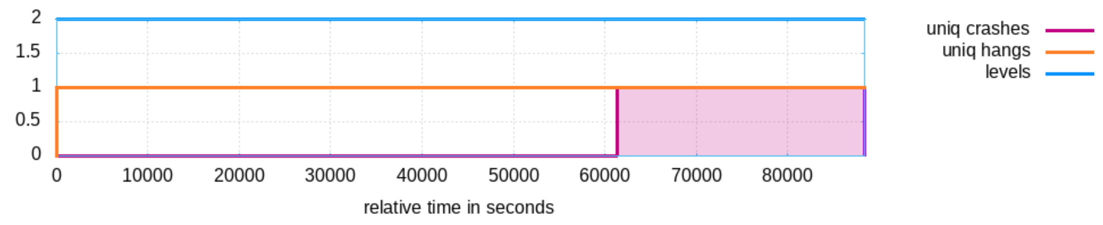
Figure 4.5: Plot of Slave Process 1: This plot shows the performance of the first slave process, which was initiated right after the execution of the Master process.



(a) Execution Speed

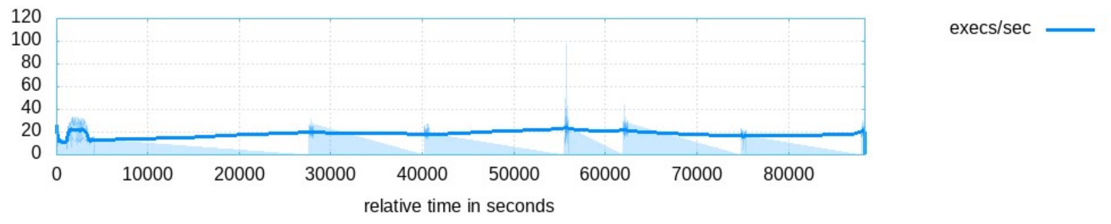


(b) Code coverage and pending items

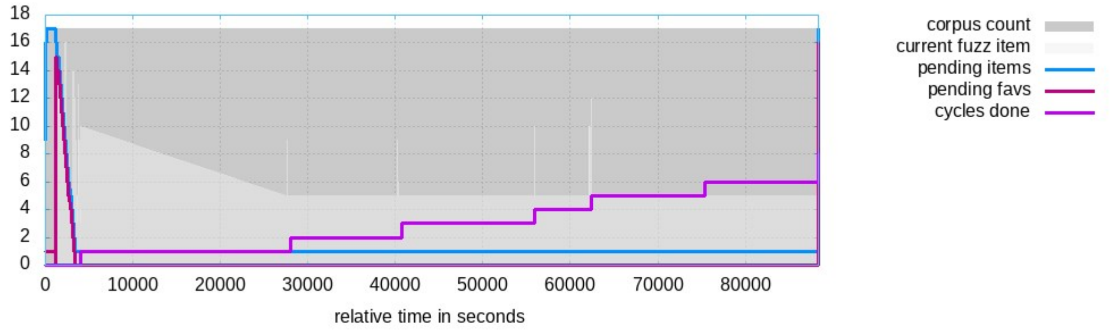


(c) Crashes and Hangs

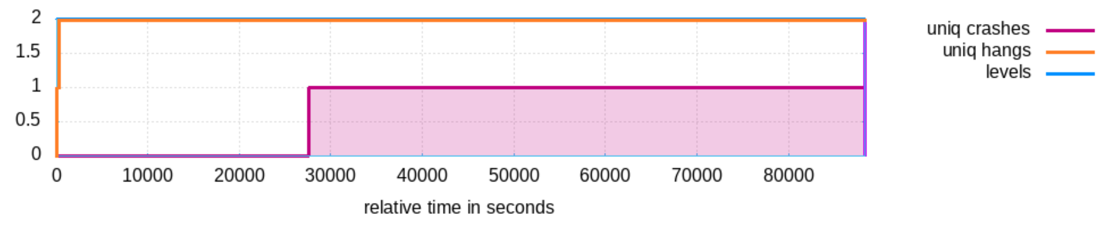
Figure 4.6: Plot of Slave Process 2: This plot shows the performance of the second slave process, which was initiated right after the execution of the first slave process.



(a) Execution Speed



(b) Code coverage and pending items



(c) Crashes and Hangs

Figure 4.7: Plot of Slave Process 3: This plot shows the performance of the first slave process, which was initiated right after the execution of the second slave process.

testing strategy may be effective. Very few system crashes were discovered in over three days of run-time and over four million executions; these crashes cannot be mentioned in this work due to security concerns.

However, the fuzzer only found problems that were mostly related to the input and how it was parsed. The primary reason for this is that Ada would not accept these kinds of parameters since the file created by the fuzzer did not adhere to the correct types specified by the record. This further confirms its strength against these types of attacks, as this behavior was quite expected given Ada's strong typing particularity.

Thus, the primary findings are not only that the open-source software AFL fuzz can be applied to avionic software successfully, but it can also search through all of the code's paths and identify potential vulnerabilities. This demonstrates that embedded avionic software can benefit from a fuzz testing approach that can identify potential issues while taking into account the time-consuming process of building the proper framework. In this particular case, improper input and its parsing were the main causes of the crashes but we were able to determine that the strong type of Ada prevented the program from crashing when given malicious random values as inputs. While it's possible that the fuzzer may have found more issues with more time, it was highly unlikely considering that there hadn't been any new findings for more than a day and that the coverage was nearly finished.

In the end, it is possible to confirm that, with some work, AFL-fuzz can be applied to the avionic system in general, and that Ada can provide an additional layer of protection against the kinds of attacks generated by the fuzzer. Even though they weren't considered for this work, there may be additional components (like sanitizers) that might be utilized in conjunction with the fuzzer and should be taken into account in the future. Furthermore, it shows the level of security that the software can offer in the event of a potential brute force attack, which can result in undesired behavior or the revealing of potentially sensitive information, although the real environment that protects the software could be considered secure enough.

5 Conclusion

5.1 Discussion

The basic principles and core notion of a potential approach to applying fuzz testing to embedded avionic software using open-source software have been shown in this work. Several tests were conducted on an actual embedded software component that is a part of the complex avionic embedded software of an Airbus NH90 helicopter to accomplish this goal.

Understanding the primary features and configurations that permit the fuzzing of a software component is what allowed the experiment to be conducted, and the guidelines that emerged from the work done on the component were important. As a result, it was feasible to construct a virtual workspace that could function similarly to an actual avionic system, enabling the creation of a fully customizable sandbox that would enable additional component testing. In order to determine whether a fuzz testing approach might be used to address the initial research questions, it was necessary to execute the experiment and see how the fuzzer and the tested system behaved. Moreover, it allowed us to come up with additional theories for future studies.

In the end, it was possible to demonstrate that it is possible to apply a fuzz testing approach when it comes to embedded avionic software, but it will need some work to ensure that the software will continue to function in this environment and to construct a viable test environment. Then, to demonstrate that testing could also be accomplished with non-proprietary software and to enable future researchers and workers to use this tool and maybe fine-tune it for their job, a potential configuration of an open-source fuzzing testing tool (AFL++) was shown. Depending on the code that is fuzzed and the language used, different performances and outcomes may occur since these factors may allow for vulnerabilities that the fuzzer is more likely to find. Consequently, the goal of this work is to at least offer some suggestions about the fundamental specifications needed to complete the fuzzing for embedded avionic software.

Several tools and libraries are required to fulfill the main requirements, which means that following best practices is necessary when performing fuzz testing. In addition to AFL++, which is the best open-source fuzzing testing tool in this situation, the appropriate compilers must also be used to produce a fuzzable binary, which in this case is also provided by AFL. Furthermore, the primary prerequisites of the virtual environment were demonstrated to accomplish the fuzzing, as AFL is limited to operating in a Linux environment. Additionally, there is an emphasis on modifying the code in advance of the fuzzing testing, which is necessary, or else it would be tough to apply this approach.

5.2 Future works

There may be various changes and improvements that would be useful and intriguing for study since this work might serve as a foundation for future studies whose goal is to apply the fuzzing testing to an embedded avionic program or a general embedded software.

Creating its genetic algorithm code, which AFL uses to create the data that will be injected into the binary, is one way to do this. Since each system would require a different set of inputs, it can be helpful to investigate whether a new generator for test cases can be made to test the software. Moreover, using sanitizers or more refined dictionaries would be a simple way to improve this experiment and enable the fuzzer to identify potential problems and vulnerabilities that it would not typically detect on its own.

Lastly, it would be interesting to observe the effects of applying fuzz testing to a larger portion of the software in terms of time and memory performance. This could allow for the discovery of more vulnerabilities, but it would need more time.

References

[1] C. Arnold, D. Kiel, C. Baccarella, K.-I. Voigt, and D. Hoffmann, “Technology adoption with reference to embedded systems”, in Proceedings of the 2nd International Conference on Advances in Management, Economics and Social Science, The Institute of Research Engineers and Doctors USA, 2015, pp. 119–127.

[2] C. Paar and A. Weimerskirch, “Embedded security in a pervasive world”, information security technical report, vol. 12, no. 3, pp. 155–161, 2007.

[3] E. Ukwandu, M. A. Ben-Farah, H. Hindy, M. Bures, R. Atkinson, C. Tachtatzis, I. Andonovic, and X. Bellekens, “Cyber-security challenges in aviation industry: A review of current and future trends”, Information, vol. 13, no. 3, p. 146, 2022.

[4] C. Baron and V. Louis, “Towards a continuous certification of safety-critical avionics software”, Computers in Industry, vol. 125, p. 103 382, 2021, issn: 0166-3615. doi: https://doi.org/10.1016/j.compind.2020.103382.

[5] P. Butcher, “Fuzz testing in international aerospace guidelines”, 2021. [Online]. Available: https://www.code-intelligence.com/blog/fuzz-testing-in-international-aerospace-guidelines (visited on 11/2023).

[6] AdaCore, “GNATfuzz”, 2021. [Online]. Available: https://www.adacore.com/dynamic-analysis/gnatfuzz (visited on 11/2023).

[7] M. Vai, D. Whelihan, N. Evancich, et al., “Systems design of cybersecurity in embedded systems”, in 2016 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2016, pp. 1–6.

[8] E. A. Lee, “Embedded software”, in Advances in computers, vol. 56, Elsevier, 2002, pp. 55–95.

[9] M. Nahas and A. Maaita, “Choosing appropriate programming language to implement software for real-time resource- constrained embedded systems”, in Embedded Systems, K. Tanaka, Ed., Rijeka: IntechOpen, Mar. 2012, ch. 15. doi: 10.5772/38167.

[10] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, “High-level programming of embedded hard real-time devices”, in Proceedings of the 5th European conference on Computer systems, 2010, pp. 69–82.

[11] Ipcserode. “Embedded systems: Powering the future of smart devices and beyond”. (May 10, 2023), [Online]. Available: https://medium.com/@ipcserddmtraining/embedded-systems-powering-the-future-of-smart-devices-and-beyond-ba7336bb60 (visited on 11/2023).

[12] A. B. Pandey, A. Tripathi, and P. C. Vashist, “A survey of cyber security trends, emerging technologies and threats”, in Cyber Security in Intelligent Computing and Communications, ser. Studies in Computational Intelligence, R. Agrawal, J. He, E. Shubhakar Pilli, and S. Kumar, Eds., Singapore: Springer, 2022, pp. 19–33. doi: 10.1007/978-981-16-8012-0_2.

[13] D. Papp, Z. Ma, and L. Buttyan, “Embedded systems security: Threats, vulnerabilities, and attack taxonomy”, in 2015 13th Annual Conference on Privacy, Security and Trust (PST), Izmir, Turkey: IEEE, Jul. 2015, pp. 145–152. doi: 10.1109/PST.2015.7232966.

[14] S. Das, W. Zhang, and Y. Liu, “A fine-grained control flow integrity approach against runtime memory attacks for embedded systems”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 11, pp. 3193–3207, 2016. doi: 10.1109/TVLSI.2016.2548561.

[15] L. Davi, P. Koeberl, and A.-R. Sadeghi, “Hardware-assisted fine-grained control-flow integrity: Towards eficient protection of embedded systems against software exploitation”, in Proceedings of the 51st Annual Design Automation Conference, ser. DAC ’14, New York, NY, USA: Association for Computing Machinery, Jun. 1, 2014, pp. 1–6. doi: 10.1145/2593069.2596656.

[16] L. Lopriore, “Memory protection in embedded systems”, Journal of Systems Architecture, vol. 63, pp. 61–69, Feb. 1, 2016, issn: 1383-7621. doi: 10.1016/j.sysarc.2016.01.006.

[17] Y. Wu, Y. Wang, S. Zhai, Z. Li, A. Li, J. Wang, and N. Zhang, “Work-in-progress: Measuring security protection in real-time embedded firmware”, in 2022 IEEE Real-Time Systems Symposium (RTSS), ISSN: 2576-3172, Dec. 2022, pp. 495–498. doi: 10.1109/RTSS55097.2022.00050.

[18] W. Youn and B. Yi, “Software and hardware certification of safety-critical avionic systems: A comparison study”, Computer Standards & Interfaces, vol. 36, no. 6, pp. 889–898, 2014, issn: 0920-5489. doi: https://doi.org/10.1016/j.csi.2014.02.005.

[19] A. Carbonari, “Avionic systems overview”, in Proceedings of the 17th Symposium on Integrated Circuits and System Design, ser. SBCCI ’04, Pernambuco, Brazil: Association for Computing Machinery, 2004, p. 6. doi: 10.1145/1016568.1016570.

[20] L. Zimmer, P. Yvars, and M. Lafaye, “Models of requirements for avionics architecture synthesis: Safety, capacity and security”, in Complex System Design and Management conference–CSD&M, Dec. 2020.

[21] A. Wölfl, N. Siegmund, S. Apel, H. Kosch, J. Krautlager, and G. Weber-Urbina, “Generating qualifiable avionics software: An experience report (E)”, in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov. 2015, pp. 726–736. doi: 10.1109/ASE.2015.35.

[22] AdaCore, “Guidelines and considerations around ED-203a / DO-356a security refutation objectives”, [Online]. Available: https://www.adacore.com/papers/guidelines-and-considerations-around-ed-203a-do-356a-security-refutation-objectives (visited on 10/2023).

[23] M. Smith, M. Strohmeier, J. Harman, V. Lenders, and I. Martinovic, Safety vs. security: Attacking avionic systems with humans in the loop, 2019. arXiv: 1905.08039 [cs.CR].

[24] A. Takanen, “Fuzzing: The past, the present and the future”, Actes du SSTIC, pp. 202–212, 2009.

[25] J. Fell, “A review of fuzzing tools and methods”, IEEE Transactions on Reliability, Mar. 2017, issn: 2084-1117.

[26] M. Zalewski, “American fuzzy lop”, [Online]. Available: https://lcamtuf.coredump.cx/afl/ (visited on 11/2023).

[27] M. Zalewski, “Fuzzing in depth”, [Online]. Available: https://aflplus.plus/docs/fuzzing_in_depth/ (visited on 11/2023).

[28] T. Huet, “AFL dictionaries”, [Online]. Available: https://github.com/mirrorer/afl/blob/master/dictionaries/README.dictionaries (visited on 11/2023).

[29] M. Zalewski. “Finding bugs in SQLite, the easy way”. (2015), [Online]. Available: https://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html (visited on 09/2023).

[30] HPC Wiki, “Compiler sanitizers”, [Online]. Available: https://hpc-wiki.info/hpc/Compiler_Sanitizers (visited on 11/2023).

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing”, in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS ’18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. doi: 10.1145/3243734.3243804.

[32] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, “Fuzzing: State of the art”, IEEE Trans. Rel., vol. 67, no. 3, pp. 1199–1218, Sep. 2018, issn: 0018-9529, 1558-1721. doi: 10.1109/TR.2018.2834476.

[33] J. Yun, F. Rustamov, J. Kim, and Y. Shin, “Fuzzing of embedded systems: A survey”, ACM Comput. Surv., vol. 55, no. 7, Dec. 2022, issn: 0360-0300. doi: 10.1145/3538644.

[34] L. J. Moukahal, M. Zulkernine, and M. Soukup, “Vulnerability-oriented fuzz testing for connected autonomous vehicle systems”, IEEE Transactions on Reliability, vol. 70, no. 4, pp. 1422–1437, Dec. 2021, issn: 1558-1721. doi: 10.1109/TR.2021.3112538.

[35] H. Turtiainen, A. Costin, S. Khandker, and T. Hämäläinen, “Gdl90fuzz: Fuzzing - gdl-90 data interface specification within aviation software and avionics devices–a cybersecurity pentesting perspective”, IEEE Access, vol. 10, pp. 21 554–21 562, 2022, issn: 2169-3536. doi: 10.1109/ACCESS.2022.3150840.

[36] L. Matias. “Leveraging ada run-time checks with fuzz testing in AFL”. (2017), [Online]. Available: https://blog.adacore.com/running-american-fuzzy-lop-on-your-ada-code (visited on 09/2023).

[37] Airbus, “Helicopters history”, 2023. [Online]. Available: https://www.airbus.com/en/who-we-are/our-history/helicopters-history (visited on 10/2023).

[38] Airbus, “NH90”, 2021. [Online]. Available: https://www.airbus.com/en/products-services/helicopters/military-helicopters/nh90 (visited on 10/2023).

[39] D. Perry, “Rotor club: Our top 10 most influential helicopters”, 2014. [Online]. Available: https://www.flightglobal.com/helicopters/rotor-club-our-top-10-most-influential-helicopters/115111.article (visited on 11/2023).

[40] F. Dordowsky and W. Hipp, “Implementing multi-variant avionic systems with software product lines”, 2010. [Online]. Available: https://dspace-erf.nlr.nl/server/api/core/bitstreams/893e867c-22dd-4ffe-a910-bc5f24ad4cc7/content.

[41] MILSTD-1553, “Complete online reference for MIL-STD-1553”, [Online]. Available: https://www.milstd1553.com/ (visited on 11/2023).

[42] United Electronic Industries, “ARINC-429 tutorial and reference”, [Online]. Available: https://www.ueidaq.com/arinc-429-tutorial-reference-guide (visited on 11/2023).

[43] J. Kelly, “RS-485 serial interface explained”, Aug. 14, 2020. [Online]. Available: https://www.cuidevices.com/blog/rs-485-serial-interface-explained (visited on 11/2023).

[44] T. Group, “DKU”, [Online]. Available: https://www.thalesgroup.com/en/dku (visited on 11/2023).

[45] M. Tooley, Aircraft Digital Electronic and Computer Systems, 2nd ed. Routledge, Jul. 18, 2013, 277 pp.

[46] EverySpec, "DOD-STD-2167 a defense system software development", [Online]. Available: http://everyspec.com/DoD/DoD-STD/DOD-STD-2167A_8470/ (visited on 11/2023).

[47] A. Srivastava, "Ada's vital role in new US air trafic control systems", Lecture Notes in Engineering and Computer Science, vol. 1, 2009.

[48] AdaCore, "Building with GPRbuild", [Online]. Available: https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/building_with_gprbuild.html#building-with-gprbuild (visited on 11/2023).

[49] M. Zalewski, "The AFL++ fuzzing framework", [Online]. Available: https://aflplus.plus/ (visited on 11/2023).

[50] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research", in Proceedings of the 14th USENIX Conference on Offensive Technologies, ser. WOOT'20, USA: USENIX Association, 2020.

[51] M. Zalewski, "AFL++ documentation", [Online]. Available: https://aflplus.plus/docs/ (visited on 11/2023).

[52] P. Butcher, "Finding vulnerabilities using advanced fuzz testing and AFLplusplus", [Online]. Available: https://blog.adacore.com/advanced-fuzz-testing-with-aflplusplus-3-00 (visited on 09/2023).

[53] Scientific Toolworks, Inc., "Understand: The software developer's multi-tool", [Online]. Available: https://scitools.com/ (visited on 11/2023).