Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 238 (2024) 273–280

Procedia Computer Science

www.elsevier.com/locate/procedia

The 15th International Conference on Ambient Systems, Networks and Technologies (ANT)
April 23-25, 2024, Hasselt, Belgium

A Fuzz Testing Approach for Embedded Avionic Software

Leonardo Xompero[a], Tahir Mohammad*[a], Jouni Isoaho[a], Jürgen Grossi[b]

[a]*Department of Computing, University of Turku, Vesilinnantie 5, Turku 20014, Finland*
[b]*Airbus Helicopters, Industriestrasse 4, Donauworth 86609, Germany*

Abstract

The objective of the research was to find if it was possible to apply fuzz testing, a technique that can be used to test software to discover vulnerabilities, on an embedded avionic software using an open-source fuzz tool. The open-source fuzz tool AFL++ was applied to an NH90 Airbus Helicopter embedded software component to find the vulnerabilities. The proposed setup was able to find a few crashes related to data parsing associated with Ada strong typing declaration requirement. Moreover, the experiments outline practical guidelines and considerations for implementing fuzz testing on embedded avionic software.

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

1. Introduction

In recent years, software, in general, has been employed within several kinds of different devices [1], eventually taking the name 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], which could be vulnerable to several attacks. The avionics field scenario represents a field in particular that may be vulnerable to these attacks [3], which threats could pose a serious risk for the aircraft. Numerous aircraft functions and operations are controlled by embedded software. As a result, it is critical to test the software and give adequate attention to the security. In this regard, it is important to determine whether any tools or techniques could help programmers quickly find possible vulnerabilities in the software.
To find answers to these questions, companies and certified authorities explore several possible solutions. This includes stricter standards to help developers produce robust software as well as software testing tools [4]. Researchers

* Corresponding author.
E-mail address: tahir.mohammad@utu.fi

1877-0509
10.1016/j.procs.2024.06.025

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 methods are 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 does not adequately provide research studies on using these techniques, particularly fuzz testing. 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 testing method. 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.

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. 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 do not guide 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. A potential strategy for using an open-source fuzz testing tool could provide further ideas for future projects, such as companies designing their own fuzz testing tool using an already-existing one.
This work aims to identify and specify a potential method for applying fuzz testing to proprietary embedded avionic software using an open-source fuzz testing tool. 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. Furthermore, the work will provide an overview necessary background to increase understanding of the fuzz testing approach and suggest ways to improve it. This includes 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.

3. Related Works on Fuzz Testing

Several works in the scientific literature attempt to describe how and what kind of results fuzz testing can achieve when applied to different applications. However, no research has been done to investigate using fuzz testing in the context of avionic software. The only fuzz testing approach that is applied to avionic software is provided by AdaCore, which offers the tool GNATfuzz [6], a tool for implementing an automated testing technique to detect abnormal and faulty behaviour. However, it doesn't explain how the fuzz works or how it can be replicated to try to perform the fuzz without relying on the private solution. Other works, such as [7], 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).
Other more specific works regarding fuzzing on embedded systems scenarios are [8] and [9]. 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 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 primarily focuses on possible DoS attacks against a particular protocol, it provides a clear understanding of the fuzz testing process. Finally, the only work that attempted a similar strategy was [10], 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 in an Ada software scenario.

4. Fuzz testing with AFL++

Fuzz testing is 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 behaviour, 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. Of all the possible fuzzing tools that can be used, American Fuzzy Lop (AFL) is the most the leading candidate due to its simplicity and efficiency. Many vulnerabilities and flaws discovered in various open-source libraries and tools have contributed to its success and rich history [11].
Specifically, our work utilizes AFL++ for experiments, which is an improved fork of the original AFL that offers faster execution times, more and better mutations, better instrumentation, and support for new modules. AFL++ was chosen in order to find possible configurations for it to produce a better fuzzing approach for Ada. AFL++ can provide 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, mutation, and persistent mode. Other features that are frequently included in fuzzer tools to improve efficiency 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 are 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 [12], allowing for more potential findings and performances. Some of these features are seed scheduling to enable more enhanced schedules and mutators. In addition to the standard ones, it is possible to use a customized mutator, and instrumentation, offering support for a wide range of code-instrumentation tools to test more components and platform support to enable the fuzz in different environments.

5. Target Architecture

The NHIndustries NH90, or NH90, is a modern multi-role rotorcraft that is one of Airbus Helicopter's primary products. It is built to meet and adhere to NATO specifications [13]. 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 NH90 comprises 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, the CORE system and the MISSION system. The subsystem researched in this work is the CORE system, which 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. 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. 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. Figure 1 shows the software architecture comprising several parts that perform different roles.
Operational Processing Functions (OPFs), which mainly include the mission functions and control for the avionic equipment, are crucial components of the software architecture. Also, in Figure 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.

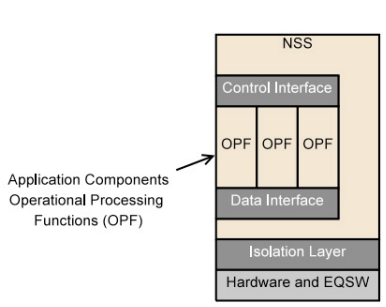

Fig. 1. NH90 Main computers software architecture, which represent how the main components inside the CMC interfaces with each other. [14]

This will play a significant role in the fuzz testing approach.
The 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 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. 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.
Given the target, fuzz testing resulted in a more suitable choice to test the vulnerabilities. There are minimal chances that this target system may be targeted by external threats since there are a lot of constraints and protections behind the embedded system. Since a hostile insider is the only type of attacker that may have access to the test tools used to test this particular kind of software, which is the only method in which input can be injected, they emerge as one of the primary threats in this case study.

6. Test environment

For the test environment, there were two essential components: the main computer, which was the work computer, where the test was performed, and the virtual environment, where the real tests were performed. The test computer was equipped with intel 11th Gen Intel(R) Core i7-1185G7 3GHz with 32GB of RAM. The virtual environment was created in Virtual-Box and running Ubuntu 20.04.6 with 4 processors and 16GB RAM. Regarding the compiler used for the test, the main compiler used was GCC (version 9.4.0), but for compiling the component that would be fuzzed, the AFL compiler ”afl-gcc” was used. To compile the Ada project, ”gprbuild” was used. It is essential to use AFL-gcc to appropriately compile and instrument the program since the code must be instrumented before applying the fuzzer, which can be done with GPRbuild by using the line ”–compiler-subst=Ada, afl-gcc,”. 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, allowing for faster execution.

7. Implementation and Verification

In order to determine whether a fuzz method to find potential vulnerabilities was possible, the CMC component of the NH90 was chosen as the component for fuzz testing. Considering the extreme complexity of the CMC code, selecting only one library within the component was required. This is also partially because the fuzzer generates random inputs. Also. adding multiple random variables to large-scale projects like the CMC would be too difficult and would require too much time. Therefore, rather than fuzzing the entire CMC, 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, like the number of dependencies of the library, the number of variables, and the

overall complexity of the library.
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” from here onwards. 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, which later would be used to create all of the wrapper functions. To properly isolate the OPF, which would undergo the fuzz testing, it was necessary to separate and create the appropriate wrapper functions, which would simulate the data exchange and thus ensure its correct functioning. This was possible by using the tool called ”Understand” [15] to identify all the dependencies of the OPF. However, other tools can also be used for the same purpose. After this, it was possible to move the selected FuzzedComponent from the Solaris environment to the VirtualBox environment, allowing to perform the fuzz testing. Since the NSS and EQSW components, in this case, served as the entry points to inject the fuzzer generated input, all of the dependencies were changed with a wrapper function after the FuzzedComponent was isolated. The ”wrapper functions” can receive a crafted data object containing the parameters that will be set inside the component, allowing FuzzedComponent to function correctly.
Therefore, the basic methodology to create the wrapper functions resides in working with the cut dependencies. To begin with, you must accurately separate every external dependency from the library you wish to fuzz. It is crucial to accurately identify every variable that might be exploited as a potential entry point to inject the input after isolating the component. After having identified the variables, the package can be created to store the data and its structure, which will be then referenced by the fuzzed library. To achieve this, a file called ”fuzzed_data.ads” was created that defines a package called ”Fuzzed_Data”. This file 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 variables 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 Figure 3(a). Using the package allows to 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”. 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”. Lastly, it is possible to correctly generate the record and pass the inputs to the right components in the main file, as shown in Figure 3(b).
Some bash-programmed scripts were utilized for this project (e.g., compile the target), allowing the virtual machine to have all of its primary functions implemented immediately and to have all the environment variables easily modifiable. An example of the main functions written in the script used to compile the target and start the fuzzer is shown in Figure 2(a). It is also necessary to change the files to the appropriate extension because there were other issues with the file extensions that were being used in the original environment (Solaris), such as ”.1.ada” instead of ”.ads”.
Since fuzz testing aims to intercept all the crashes that happen after the injection of a crafted input, the programs must warn the fuzzer of this crash by using exceptions. Therefore, to catch the error, a simple top-level exception handler was used based on the work of [10], which is shown in Figure 2(b). The following libraries are used to create the exception and to create the core dump: ”Ada.Exceptions” and ”GNAT.Exception_Actions”.
The initial input must be properly constructed for the program to function properly; otherwise, the fuzzer will not be able to generate further cases because the first case has already crashed. To create it, a file was constructed to correctly create the initial test case, where the initial inputs were possible values that could be accepted by the FuzzedComponent. This file would create an object that would be used as the input for the fuzzer, which after being inserted in the executable, would be parsed by a constructed function to give value to the appropriate variables.
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, using afl-gcc as a compiler to instrument the code so that the executable can provide code coverage information. 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.

```
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 @@
}

fuzz_compile_programs() {
    gprbuild -p --compiler-subst=Ada,afl-gcc
    -P $GPR_PROJECT
    --autoconf=fuzzing/fuzzing.cgpr
}
```

(a)

```
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;
```

(b)

Fig. 2. Main examples of code components needed to correctly apply the fuzz testing. Figure 2(a) represents the script's functions needed to compile and start the fuzz testing. Figure 2(b) represents the Exception's code needed to catch the crashes.

```
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;
```

(a)

```
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;
```

(b)

Fig. 3. Different examples of codes that allow the correct exchange of data between the components. Figure 3(a) represents fuzzed_data.ads to create the data record type which will be used as input. Figure 3(a) represents how the main.ads should be constructed to receive the correct data type from the fuzzer.

8. Results and Discussion

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) as shown in Figure 4. The experiment 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, as shown in the summary of the fuzzing session in Figure 5. More information regarding the

behaviour of the fuzzer at runtime can be seen in Figure 6, 7, and 8. The proposed fuzz testing approach found some

```
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 (!)
```

Fig. 4. Main information regarding each fuzzer that was running at runtime.

```
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
```

Fig. 5. Summary of the main results of all the fuzzers.

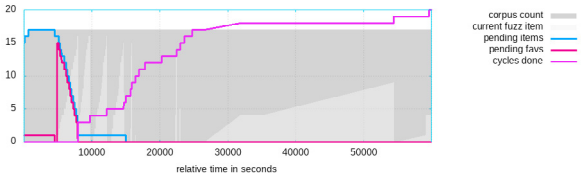

Fig. 6. Graph that shows the behavior of the fuzzer at runtime.

Fig. 7. Graph which shows the main crashes and hangs found at runtime.

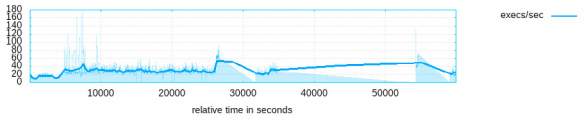

Fig. 8. Graph that shows the execution speed.

problems that could give crucial insight into the behaviour of an avionic software when under stress test by the fuzzer. This further confirms some hypotheses on the behaviour of a fuzzer towards an Ada program and gives an idea of the expected performances. 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 testing strategy may be effective. These crashes were mostly due to the type constraints imposed by the strong typing of Ada, which would throw a runtime error when a variable would have assigned a wrong input type. 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 behaviour was quite expected given Ada's strong typing particularity. While it's possible that the fuzzer may have found more issues with more time, it was highly unlikely, considering that there have been no new findings for more than a day and that the coverage was nearly finished.

In summary, 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 were not considered for this work, there may be additional components (like sanitisers) that might be utilized in conjunction with the fuzzer and should be taken into account in the future.

In the end, the experiment was successful since it provided some crashes as a result of the random inputs injected by the fuzzer. Since the fuzzer was able to find some errors, it is then possible to apply the fuzz testing to avionic software by using an open-source tool, which could also work with different programming languages apart from Ada. The errors were mostly due to the type constraints imposed by Ada, thus further demonstrating its resistance against mistyping.

9. Conclusion

The basic principles and core notion of a potential approach to applying fuzz testing to embedded avionic software using open-source software were presented 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. We demonstrated the applicability of fuzz testing in embedded avionic software using an open-source tool and to enable future researchers to use this tool and fine-tune it for their test cases. A potential use of an open-source fuzzing testing tool (AFL++) was shown, which allowed to find some crashes in the targeted component. From this application, an approach was outlined, which could be used as a reference for future works.

Additionally, there is an emphasis on modifying the code in advance of the fuzz testing, which is necessary to identify potential crashes or vulnerabilities. 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. Understanding the primary features and configurations that permit the fuzzing of a software component allowed the experiment to be conducted, and the guidelines that emerged from the work done on the component were important.

References

[1] C. Arnold, D. Kiel, C. Baccarella, K.-I. Voigt, 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, A. Weimerskirch, Embedded security in a pervasive world, information security technical report 12 (3) (2007) 155–161.

[3] E. Ukwandu, M. A. Ben-Farah, H. Hindy, M. Bures, R. Atkinson, C. Tachtatzis, I. Andonovic, X. Bellekens, Cyber-security challenges in aviation industry: A review of current and future trends, Information 13 (3) (2022) 146.

[4] C. Baron, V. Louis, Towards a continuous certification of safety-critical avionics software 125 (2021) 103382. doi:https://doi.org/10.1016/j.compind.2020.103382.

[5] P. Butcher, Fuzz testing in international aerospace guidelines (2021).
URL https://www.code-intelligence.com/blog/fuzz-testing-in-international-aerospace-guidelines

[6] AdaCore, GNATfuzz (2021).
URL https://www.adacore.com/dynamic-analysis/gnatfuzz

[7] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang, Fuzzing: State of the art, IEEE Transactions on Reliability 67 (3) (2018) 1199–1218.

[8] L. J. Moukahal, M. Zulkernine, M. Soukup, Vulnerability-oriented fuzz testing for connected autonomous vehicle systems, IEEE Transactions on Reliability 70 (4) (2021) 1422–1437. doi:10.1109/TR.2021.3112538.

[9] H. Turtiainen, A. Costin, S. Khandker, T. Hämäläinen, Gdl90fuzz: Fuzzing - gdl-90 data interface specification within aviation software and avionics devices–a cybersecurity pentesting perspective, IEEE Access 10 (2022) 21554–21562. doi:10.1109/ACCESS.2022.3150840.

[10] L. Matias, Leveraging ada run-time checks with fuzz testing in AFL (2017).
URL https://blog.adacore.com/running-american-fuzzy-lop-on-your-ada-code

[11] AFLPlusPlus, The AFL++ fuzzing framework.
URL https://aflplus.plus/

[12] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, AFL++ : Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies (WOOT 20), USENIX Association, 2020.

[13] Airbus, NH90 | airbus.
URL https://www.airbus.com/en/products-services/helicopters/military-helicopters/nh90

[14] F. Dordowsky, W. Hipp, Implementing multi-variant avionic systems with software product lines (2010).
URL https://dspace-erf.nlr.nl/server/api/core/bitstreams/893e867c-22dd-4ffe-a910-bc5f24ad4cc7/content

[15] Understand: The software developer's multi-tool.
URL https://scitools.com/