Fuzz testing for design assurance levels

Marcus Gustafsson
Oscar Holm

Tutor, Eric Elfving
Examiner, Jonas Wallgren
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida
http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page:
http://www.ep.liu.se/.

© Marcus Gustafsson & Oscar Holm
Fuzz testing for design assurance levels

Oscar Holm
Linkoping, Sweden
oscho707@student.liu.se

Marcus Gustafsson
Linkoping, Sweden
margu478@student.liu.se

ABSTRACT
With safety critical software, it is important that the application is safe and stable. While this software can be quality tested with manual testing, automated testing has the potential to catch errors that manual testing will not. In addition there is also the possibility to save time and cost by automating the testing process. This matters when it comes to avionics components, as much time and cost is spent testing and ensuring the software does not crash or behave faulty. This research paper will focus on exploring the usefulness of automated testing when combining it with fuzz testing. It will also focus on how to fuzzy test applications classified into DAL-classifications.

INTRODUCTION
In modern airplanes there are parts that need to communicate with each other. In this environment, data needs to be able to flow securely from one part to another. ARINC 653 is a specification for avionics real-time operating systems (RTOS) that explains how data can flow between these parts [1].

Applications resides inside partitions. When a partition is executing, it does not need to finish executing before sending data to a component. The components are parts of a interface which links the partitions with the hardware and operating system. The partitions, interface, operating system and hardware makes up a module. Modules has a component for monitoring and handling software errors and crashes, this is provided by system partitions. This is done to prevent component failures from propagating to other partitions or modules. Since it is important that the airplane can be controlled in a secure and comfortable way, identifying critical sectors is very important. This can be done by using Design Assurance Level (DAL).

DAL is used to classify applications in avionics systems [2]. Classification is based on a risk analysis of the application. There are different DAL-classifications. Higher DAL-classifications has stricter requirements on software reliability, an error in the highest DAL-classification can cause wreckage. Contrary to the lowest DAL-classifications where a software failure will only have a minimal affect to the safety, control over the airplane nor does it strain the crew [3].

Software that is being tested will be referred to as the Software Under Test (SUT). While studies suggests that manual testing is widely used today [4, 5, 6], fuzz testing can make it possible to do more automated and broader testing of the SUT.

Fuzz testing is usually not specified to a certain part of the SUT, but instead tests random paths throughout the whole SUT which might give unexpected results [7]. This kind of testing is achieved by using a variety of strategies and algorithms to mutate the input data fed to the SUT. American Fuzzy Lop (AFL)\(^1\) is a tool which uses a deterministic, whitebox variant of fuzz testing. This variant of fuzz testing tries to understand how the SUT works by using certain strategies and a feedback loop. It can then target specific paths in the software. Non-deterministic fuzzers\(^2\) will not use deterministic steps during mutation of input data but instead just randomize input.

If a fuzzer can recognize important bytes in input data, it categorizes as a smart fuzzer [8]. Likewise, if it is aware of the application structure it will categorize as a whitebox fuzzer. AFL for example learns the application structure when instrumenting the SUT during compilation. This separates AFL from other fuzzers as it is categorized as a smart, deterministic and whitebox fuzzer. When the deterministic strategies are exhausted, AFL will dive into completely non-deterministic, random behaviour. Since deterministic strategies are employed in AFL, the input will be random to a lesser extent than what it would be for a non-deterministic fuzzer.

Purpose
The goal of this research paper is to evaluate what fuzz testing can contribute in testing of different DAL-classifications and outcome in amounts of errors, and implement a solution that can:

1. Generate code from C++-header files and XML-files.
2. Create a layer which links the SUT with AFL where data can flow continuously.
3. Test the generated code with AFL.
4. Log eventual errors found in the SUT.

We are also evaluating how the architecture of the SUT affects the usefulness of fuzz testing and draw conclusions from the correlation between code coverage and outcome.

Research questions
Our main research question is if fuzz testing can be useful for finding errors in software that is written and tested according to different DAL-classifications, and can be broken down into following:

- What is the correlation between amount of errors found with fuzz testing and code coverage?
- What is the relation between amount of faults found and DAL?

\(^1\)http://lcamtuf.coredump.cx/afl/
\(^2\)A fuzzer is a program which employs fuzz testing
Limitation
For this research paper we are using AFL because it is a well-established tool in fuzz testing. AFL has found vulnerabilities in for example Mozilla Firefox\(^3\), OpenSSL\(^4\) and SQLite\(^5\). For the code generation we will only use Dextool and its libraries.

Delimitation
This study will not be about the code generation of ARINC 653 interfaces, it will be about the security of applications that has undergone testing and verification towards their DAL-classification.

THEORY
In this chapter, we will go through the terminology related to our testing environment such as DAL-classifications, ARINC 653, fuzz testing and code generation.

Design Assurance Level
In avionics systems applications can be classified into Design Assurance Levels (DAL). If the system is using DAL, the classification depends on several factors. One of them is the possible result of a hardware failure, where DAL-classification depends on severity of failure, as seen in figure 1. A risk analysis determines what DAL-classification a application should have.

<table>
<thead>
<tr>
<th>DAL-classification</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous/Severe</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
</tr>
<tr>
<td>D</td>
<td>Minor</td>
</tr>
<tr>
<td>E</td>
<td>No Safety Effect</td>
</tr>
</tbody>
</table>

Figure 1. DAL-classifications and their severities.

Severity can for example be based on a failure that would affect handling of the aircraft, cause crew members discomfort or both. A failure with severity of minor or no safety could for example be when a number is incorrectly displayed on the instrument board \([2]\). DAL A has the highest requirement for the application to be stable at all times. If a application gets classified into the DAL A it needs to be tested, documented and verified thoroughly. The requirements on testing and verification usually gets lower depending on the DAL classification, and therefore the severity of a fault in the application. DAL E is for example tested and verified towards different requirements than DAL A. In theory, if the DAL classification has been done correctly it ensures that no unnecessary testing has been done \([2]\).

ARINC 653
The ARINC 653 specification defines the structure of software partitioning in avionics real-time operating systems.

\(^4\)https://www.openssl.org/news/secadv_20150611.txt
\(^5\)https://www.sqlite.org/src/info/9e6eae660a0230t

Partitions
All partitions are classified as either a application partition or a system partition. When a partition is executing, it does not need to finish executing before sending data to a component in the connected interface. Applications are partitioned (application partitions) to isolate their execution from the rest of the system. System partitions can provide system functions such as fault handling and device drivers that the interface does not have \([1]\).

In order to isolate the partitions, they do not have any information on where their data is being sent or when they receive data. Partitions have ports for receiving and sending data. It is possible for several partitions to receive data from the same port, data sent to this port is received by all partitions listening to that port. In short, data flows in or out of these ports in order for the partitions to communicate with the interface \([1]\).

Modules
A module is made up of partitions, a interface with components, a operating system and the associated hardware. The architecture for a module in ARINC 653 is specified in such a way, that applications should be portable between modules without any modification to the interface, operating system or the hardware \([1]\).

Application/EXecutive
The interface which acts as a link between hardware, operating system and partitions is called APplication/EXecutive (APEX). The purpose of APEX is to make it possible for everything in the module to communicate, and to handle important system tasks such as process scheduling and health monitoring of partitions \([9]\). This work is done by several components residing in the APEX interface \([1]\).

- Partition management for starting partitions or changing partition modes \([1]\).

![Diagram](https://www.example.com/diagram.png)

Figure 2. Module structure following ARINC 653 specification.
• Process management for killing, starting, stopping processes. Synchronization for processes within a application is achieved with semaphores. This component in short provides important functions for managing processes, it does however not tell APEX how these processes should be handled [1].

• Time management for scheduling processes into correct time frames. A fixed priority preemptive scheduling strategy is used [10]. Processes are associated with a time frame cost for executing and a period specifying how often the process should execute. Processes also have priorities which tells the scheduler how important it is to finish them. The scheduler will need to ask the process management component whenever it wants processes interrupted.

Lastly, the deadline associated to a process tells the scheduler what to do when a process does not finish in time. Deadlines not met will inform the operating system to take action. Example of such action would be killing the process or restarting the partition associated with the process. The time and process management components is what makes APEX able to handle context switching and interrupts of processes.

• Memory management for assigning memory to partitions. It is assumed memory is statically allocated to partitions and processes before they run. For this reason, dynamic memory allocation is not allowed during execution of partitions [1].

• Interpartition communication for allowing partitions to communicate with other partitions [1].

• Intrapartition communication for allowing applications in a partition to communicate with each other [1].

• Health monitoring for the status of the partitions.

Code generation
Genuine avionic real-time operating systems for testing is impractical, as that would require real avionics hardware. For this reason and to be able to inject the middle layer, code is generated to simulate the testing environment. In order to simulate the partitions, code needs to be generated from the XML-records and the code used to describe the ports. The APEX interface can be represented with a generated interface following the ARINC 653 structure, and additionally with the capability of connecting, reading and writing to specified ports as this is a implementation requirement for the software we are testing.

The generated interfaces needs to handle the data generated from AFL and use it to update the data in the ports. A middle layer needs to be implemented to handle the connection between the interfaces and AFL.

Whitebox fuzz testing and blackbox fuzz testing
Whitebox fuzz testing is a variant where the fuzzer has access to the source code and then theoretically knows everything about the SUT. This can be achieved by using instrumentation when compiling the SUT. AFL is able to instrument binaries with both GCC ⁶ and Clang⁷. The blackbox fuzzer does not know anything about the application and will randomly generate or mutate the input data. This approach is not very smart in the sense that it does not know any paths or if it has been able to reach a certain point in the code. The whitebox fuzzer will learn from the output from the SUT, you could say that it knows what code has been tested [11, 12].

Smart and dumb fuzzers
When a fuzzer is aware of the input structure and can understand important bytes in inputs⁸ it is categorized as a smart fuzzer. This is a must if the fuzzer wants to mutate existing input in a smart way as it needs to understand which bytes in the input that is important for the application. It also makes it possible to sort out unnecessary inputs which would yield the same result as an input already tested. When the fuzzer is not able to understand the input structure and just randomly generates or mutates inputs, it is categorized as a dumb fuzzer [8].

Generation or mutation-based fuzzers
There are fuzzers that constantly generates its input from scratch, these are called generated-based fuzzers. Mutation-based fuzzers will however mutate the input data instead of generating completely new data. Usually, a smart fuzzer will also be mutation based, as it is capable of understanding the input structure it can also mutate inputs in a smart way [13, 14, 15].

RELATED WORKS
Fuzz testing can discover vulnerabilities in a application, if given adequate amount of time. In a real world example, there are time constraints which makes it necessary for us to establish a solid understanding of the correlations between the time a fuzzer spent learning and errors found in the SUT. Resources are limited, and we want to test the SUT quickly while still finding most of the vulnerabilities and errors in general. Early data-mining approaches has shown promising results for using feedback based testing to increase the frequency at which errors are discovered when testing software [16].

Patrice Godefroid et. al. explains that whitebox fuzz testing has been successful for automated testing at Microsoft. It was able to find several security vulnerabilities in different kinds of software, covering most kinds of software when testing. A blackbox fuzzer was more efficient if the software had not been tested by a whitebox fuzzer, mainly because it was better at finding simple errors [12].

Once it came to finding the complex bugs, whitebox fuzz testing was found to be the better choice. It was more intelligent because it could find paths in the code in order to provide higher code coverage. This is possible since the whitebox fuzzer knows the software it is testing. Blackbox fuzz testing is testing the software with random input data to see if it crashes. If there is no crash, new input data is generated

---

⁶https://gcc.gnu.org/
⁷https://clang.llvm.org/
⁸For example, image files or plain text
or mutated. It is a common consensus in fuzz testing that it is best to start by looking for the things that are actually wrong. The fuzzer may report that it found errors, which might just be the intended way the software should work and not an actual error [11, 12]. The interesting part comes to investigating if the said error is a real error which would result in a faulty system or if it was able to handle that error in a good way.

Covering all code is a complex task for the fuzzer if it is not supplied with relevant data for the SUT. Input data is very subjective to the SUT, not all input data can be used with every type of the SUT. Generating usable input data for the SUT could therefore take a long time if it is not supplied with relevant input data from the beginning [11, 17]. For one to be able to generate as good test cases as possible one could generate code coverage reports based on the data generated by AFL. Those can then be analyzed to be able to generate better test cases which would cover the entire application as stated in the work by S. Bekrar et. al. [11]. This would reduce the time the fuzzer would need to run before finding any faults in the SUT.

To reach all paths in the application it might be necessary to implement some kind of system that makes the application go through a specific path. Gerlich et al. presented Fault Injection (FI) where they implemented test inputs with NULL in them. This made the branch coverage better as they were able to reach error states in the code [18].

**METHOD**

This chapter will go through our approach for fuzz testing the generated interface following the ARINC 653 specification. It will also describe how we research the correlation between amount of errors and code coverage, and the relation between amount of faults found and DAL.

**AFL**

American Fuzzy Lop (AFL) generates binary data which the interface we want to generate will use. AFL data is not usable in its "normal" form, so we will need to pick bytes of data to determine the type of the random generator, amount of cycles and seed. The type of the random generator will be any of static or Mersenne Twister [19]. The static variant needs something that tells it which port variable should have what value, and which cycles in the execution of binary (Not the same thing as a AFL cycle) this should be set, so we can control and guide the behaviour in AFL.

For this purpose, a configuration file with simple formatting (see figure 3) is used. It defines static values and which cycles the port variable should have the static value and the third part is the value.

<table>
<thead>
<tr>
<th>port1 . var1</th>
<th>100–200</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>port1 . var2</td>
<td>100–200</td>
<td>7</td>
</tr>
<tr>
<td>port3 . var1</td>
<td>201–300</td>
<td>10</td>
</tr>
<tr>
<td>port4 . var4</td>
<td>201–300</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 3. Space delimited file for setting static variables in AFL. First part is the port variable name, second part is which cycles the port variable should have the static value and the third part is the value.

The amount of cycles that was read from the AFL input will be used to determine when the application should quit, i.e. stop generating data to the ports and quit the application. That will be one full execution in AFL.

As there can be many different data types in one port, we need to handle type casts for different types properly to ensure a port receives valid data. The random generator in each port will then be initialized with the seed. Then it can generate new data for every item in the port based on the requirements on every item. Requirements will be specific types and their ranges. For example an item can be of type `int` and have the range 1 to 99. The random generator for the port will then generate an `int` in the range 1 to 99. These requirements are specified in the XML-files given. Since the entire input data from AFL can be smaller than the bytes needed, that input data will be ignored and the Software Under Test (SUT) will be restarted with new input data from AFL.

For us to be able to realistically simulate an avionics system, we must have changing values. For that reason random generation will be used to get the most out of every cycle the SUT will run. The random generator will update the values with newly randomly generated data every cycle. When the data updates, the SUT can have a different behaviour than the one before, this will make us able to reach even more branches than it would if the data would be static.

The byte range from AFL binary and what each byte represents can be seen in figure 4.

<table>
<thead>
<tr>
<th>Size of input</th>
<th>Randtype</th>
<th>Cycles</th>
<th>Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Variable</td>
<td>Variable</td>
</tr>
</tbody>
</table>

Figure 4. Expected bytes in AFL input.

The testcases for AFL will be small sized binary data, and if possible it will be in a format that looks like the format that we will use (random generator type, cycles and seed).

Testcases are at start loaded into a FIFO-queue. The first testcase from the queue is selected and minimized as much as possible. AFL will employ deterministic strategies such as sequential bit/byte shifting and arithmetics with known integers to mutate data. When the strategies are exhausted, it will mutate data using random behaviour. When a mutation reaches a new state in the SUT, the testcase is saved into the back of the queue and the algorithm is repeated for next case in the queue.

https://lcamtuf.blogspot.se/2014/08/binary-fuzzing-strategies-what-works.html
Code generation

Code generation is a vital part for this research paper since it is needed for simulating the parts of a avionics system we do not have access too, mainly APEX and the underlying system. We use the generated code for injecting our middle layer into the avionics code. The code generation will be somewhat basic. Dextool\(^\text{10}\) is a tool written by Joakim Brännström which makes it possible to parse C++-header files and generate C++ interfaces.

Dextool uses LLVM/Clang\(^\text{11}\) for static analysis of the source code and then parses the data given by LLVM/Clang and provides a easy-to-use interface to handle the code.

![Dextool plugin flowchart.](image)

The basic functionality of code generation will not be sufficient for using with our simulation. We must write a plugin for Dextool in order for it to output usable code. A compilation database \(^\text{12}\) with the rules used to compile the complete application will be taken as an input parameter. The compilation database will both be used to find all relevant header files and to generate a Makefile. To make sure that only relevant interfaces are generated, one or more directories with XML-files will be taken as another input parameter to the plugin. These XML-files contains representations of the interfaces.

The header files found from the compilation database and all the XML-files will be analyzed. When all the files has been analyzed a filter process will occur, this will remove unnecessary code that does not need to be in the output code. The unnecessary code will be determined by what the XML-files contains and is therefore the relevant interfaces. After the filter process is done it will start to generate code for the interfaces. It is important to note that the generated interfaces are what will represent the APEX interface in each test run. When the generation of the interfaces has been finished, we will use the compilation database to generate a Makefile. This Makefile will be used to merge the middle layer with the interfaces. The interfaces will be compiled to object files which will then link to our middle layer.

A header file from ARINC 653 usually contains nested namespaces and within them classes. A class is representative to a port and will contain getters and setters for port variables and a function for creating the port. What is going to be generated:

- Basic implementation of the function defined in the files.
- Empty constructors and destructors.
- Functions for the port classes which can generate and set new values for the port variables.

While code generation is a vital part of the research paper, we also need to create a middle layer between AFL and the simulated application. The middle layer will be a library which has functions that can generate values for the variables in the ports, create a structure from the AFL data and handle the random generation of data. AFL will therefore only be a small portion of the entire flow of the application, it will only be used to start the application and feed it with data that is used in the SUT. The middle layer will take over the process of running the SUT and feeding it with new data based on the AFL input data. AFL will still be used to handle crashes and hangs.

Sanitizers

GCC and Clang has an option to be able to detect memory errors and undefined behaviour during runtime. This can be used by AFL to detect potential crashes since AddressSanitizer (ASAN)\(^\text{13}\) and UndefinedBehaviorSanitizer (UBSAN)\(^\text{14}\) will send abort signals, if compiled with the AFL compilers, when finding behaviours that are undefined or memory errors. Since AFL registers abort signals as crashes, it is possible to debug these crashes just like any normal crash in AFL. As we use GCC 4.9.4 when compiling with sanitizers the following checks are made when enabling UBSAN: shift, integer-divide-by-zero, unreachable, vla-bound, null, return, signed-integer-overflow. The LeakSanitizer is enabled by default when using ASAN \(^\text{15}\). Using sanitizers will slow down execution speed and increase memory usage but in general allows more errors to be found \([\text{20}]\).

Program flow

Normal applications written for partitions in avionic systems have predefined functions that will not be generated, these functions are specified in the ARINC 653 specification. These are used for initialization, execution and sometimes termination. The initialize function runs at the start of the

---

\(^{10}\)https://github.com/joakim-brannstrom/dextool  
\(^{11}\)https://clang.llvm.org/  
\(^{12}\)https://clang.llvm.org/docs/JSONCompilationDatabase.html  
\(^{13}\)https://clang.llvm.org/docs/AddressSanitizer.html  
\(^{14}\)https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html  
\(^{15}\)https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer
application and the execute function will run in a loop (See figure 6). After the execution function is finished, the program will update all the port variables. The port variables will in most cases, when not simulated, use APEX instead of doing it programmatically. This means that port variables will be updated between each execution. If a function for termination exists this should be run after the loop is finished.

1. The simulation will start by reading from input. As AFL sends its data through standard input.
2. Our AFL-parser will parse that data and return the needed parts.
3. The application will initialize, create new ports etc.
   (a) The ports created by the application will be saved to a list for easy access.
   (b) If a port already exists, it will be reused.
4. Now it will loop as many times as AFL has selected (one of the parts derived from input bytes, also known as "for n cycles").
   (a) Update all ports with new values.
   (b) Execute the application with the the new values.
5. When the loop is finished it will terminate if a function for that exists.

Workflow when testing an application
To be able to feed an application with AFL input data we need to merge our own middle layer with the Software Under Test (SUT). This is done by using a Dextool plugin. This plugin takes XML directories, a compilation database and an application name as parameters. These are used to produce a fully functioning middle layer. The Dextool plugin does also generate a Makefile, this is used to compile everything to a executable file. AFL then needs to be fed with input data, for it to be correct we created our own script that generates input data that follows our data structure. When the executable file has been compiled it can be started with AFL.

For the coverage data to be generated we will run AFL-cov. AFL-cov needs to have a binary that has code coverage enabled. That binary and its dependent files are compiled with GCC and its code coverage option. AFL-cov will run either at the same time as AFL or after, this does not matter in our case since AFL-cov is pretty fast when running the test cases.

16https://github.com/mrash/afl-cov

AFL-cov uses the input data that is generated by AFL when a new path has been found. This ensures that the coverage data is correct and so that it does not miss any testcases that AFL has generated.

Code coverage
Coverage for applications are measured in lines executed, functions executed and the number of paths found by AFL. If a line or functions in the code is called at any point during fuzz testing it will be added into the code coverage. This is done with GCC builtin code coverage system (gcov). The number of paths found is something that AFL keeps track off and is reported to the user through its graphical interface.

Fault and code coverage correlation
To determine how strong the relation is between amount of faults and code coverage, we must compute the correlation coefficient. There are several methods to do this [21], but given our sample size we will use Pearson correlation coefficient, see figure 8.

\[
\begin{align*}
\Large r &= \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}
\end{align*}
\]

Figure 8. Formula for calculating the fault and code coverage correlation coefficient.

As sample, we will use applications corresponding to DAL A, C and E where we denote x as amount of faults and y as code coverage in percentage. The choosing of DAL-classification A, C and E is because of that it is difficult to find any relevant and interesting applications to fuzz test in DAL-classification B and D. Neither is all applications big enough to have faults in them. We will denote n as amount of applications tested. We aim to fuzz test each application with our implementation for 4 days on identical environments to get the amount of errors and the code coverage. The 4 day goal is an estimation of how long time it will take to find crashes in higher DAL-classifications. We suspect that it will be much harder to find errors in higher DAL-classifications, for that reason the
time needed to fuzzy test and get result from a application of higher DAL-classification sets precedence for the fuzz testing duration of all programs. Since we are testing a few selected applications we can fuzzy test them much longer.

Fault and DAL relation
In order to find a relation between amount of faults and DAL-classifications, we will fuzz test each application in every DAL-classification. We will compare the resulting faults found between DAL-classifications. The computation for this will be very basic, we will take the sum of all faults $F_n$ in every application $P_n$ divided by amount of applications $n$ and then summarize average faults for each DAL-classification $D_n$. See figure 9.

$$D_n = \frac{\sum F_n}{n}$$

Figure 9. Formula for calculating average faults for each DAL-classification.

Logging faults
AFL catches abort signals from the SUT, and will save the data that caused the fault. Since the amount of faults is the main interest of this research paper, we will use the command line interface to collect statistics about amount of faults. In order to verify that the cause of faults is indeed in the SUT and not in our own middle layer, we will check cause of crashes with global debugger and AFL crash triage on all the unique crashes found. Unique crashes will be found in the sync directory specified when starting AFL. AFL does not by default catch ASAN or UBSAN errors.

Maturity level
Every application we are testing is associated with a maturity level. Maturity level can be either "done" or "not done". If an application is "done", it means it has undergone enough testing to fulfill all required security standards for its DAL-classification. Which maturity level an application has, is taken into consideration when evaluating the results we get from fuzz testing.

Test cases

<table>
<thead>
<tr>
<th>Application</th>
<th>DAL-classification</th>
<th>Maturity level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A</td>
<td>Done</td>
</tr>
<tr>
<td>A2</td>
<td>A</td>
<td>Done</td>
</tr>
<tr>
<td>C1</td>
<td>C</td>
<td>Done</td>
</tr>
<tr>
<td>E1</td>
<td>E</td>
<td>Done</td>
</tr>
</tbody>
</table>

Figure 10. Test cases for fuzz testing.

As shown in figure 10, we have four applications being tested. The two applications that has DAL-classification A and is considered "done", is not expected to crash. A crash in any of these applications would indicate something went wrong during the development, testing and verification process. Faults on that level should not remain undetected for that long. For the application that has DAL-classification C (C1), we expect there to be a possibility of faults, although very unlikely.

Application C1 is also considered "done", something that may further deny chances of finding crashes. Lastly, we have a DAL-classification E application (E1) which we expect to find crashes in. E1 is also considered "done". To our understanding DAL-E applications is tested with the least requirements of all DAL-classifications, and for that reason we think the biggest chance we find errors is in application.

RESULTS
We are presenting our results derived from the method and answering our research questions in this chapter.

Fault and code coverage correlation
Each application has been fuzzy tested for 4 days. AFL did not find any crash in any of the applications. The expected result was to have at least one crash in DAL-classification E and maybe one in DAL-classification C. This was not the case.

<table>
<thead>
<tr>
<th>Application</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0</td>
</tr>
<tr>
<td>A2</td>
<td>0</td>
</tr>
<tr>
<td>C1</td>
<td>0</td>
</tr>
<tr>
<td>E1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 11. Fault data retrieved from testing of applications.

As shown in figure 11, no faults were found in any of the applications. Therefore it is unnecessary to calculate the Pearson correlation coefficient.

<table>
<thead>
<tr>
<th>Application</th>
<th>Line coverage</th>
<th>Function coverage</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>91.7%</td>
<td>90.0%</td>
<td>256</td>
</tr>
<tr>
<td>A2</td>
<td>73.4%</td>
<td>77.7%</td>
<td>532</td>
</tr>
<tr>
<td>C1</td>
<td>63.8%</td>
<td>48.5%</td>
<td>314</td>
</tr>
<tr>
<td>E1</td>
<td>41.7%</td>
<td>72.4%</td>
<td>416</td>
</tr>
</tbody>
</table>

Figure 12. Coverage and path data retrieved from testing of applications.

The code coverage fluctuates a lot, this mainly depends on the total amount of source code in the applications. For example, application A1 has the highest coverage and does also have the least amount of source code (around 200 lines of code). Application E1 is a special case since it seems to have functions with many lines of code, quite the opposite of how the coverage for C1 is.

Fault and DAL relation
No faults were found in neither of the applications and their respective DAL-classifications. The results was expected for the DAL A applications, but not for DAL C and DAL E.

 Application | Faults |
-------------|--------|
 A            | 0      |
 C            | 0      |
 E            | 0      |

Figure 13. Data retrieved from the applications

DISCUSSION
In this chapter we evaluate the findings presented in the results, the factors influencing these and how well this answers our research questions.
Threats to validity
Given by the method presented in this paper, there is several flaws that can influence the result. There is a bias issue in selecting test cases, as we chose test cases we expected was worth fuzz testing. Fuzz testing can be a very lengthy process because no real exit conditions exists. For that reason we chose to focus on providing longer fuzz testing sessions as data to the research and less time on compiling many applications. Fuzz testing for a longer time will solve the issue of no exit conditions. We will be able to ascertain that no new paths has been found for a long time, and that further testing is unlikely to yield any result.

There is also an issue of how much testing the applications previously has undergone. If an application is considered "done", it should be very hard to find errors. It might even be the case that AFL needs guiding to reach the state machines that may cause errors. These kind of applications is not desirable to fuzz test if finding crashes is the intention. In theory, our method could be repeated with applications considered "not done" and a bigger sample to yield results answering our research questions better. Considering the amount of crashes found, the test cases chosen was detrimental to our result and there is also the possibility that AFL can not find errors in these kind of applications.

As we started fuzz testing we got some faults, these faults were so called false positives. False positive is, in our case, a fault that did not happen in the application itself but in our middle layer. These false positives are registered as a crash in AFL. As false positives appear in our code, they were easy to fix so that they do not happen again. When a crash appeared we would need to debug the application with the same AFL input data. Multiple crashes appeared when adding UBSAN and ASAN, these were not found when running AFL without sanitizers.

DAL-classifications that are final could have been changed from a higher DAL-classification to a lower classification during development which means that some applications has been tested towards higher requirements than their corresponding DAL-classification. This is a problem as the DAL C or DAL E application could have been tested and verified towards a DAL A-classification, this can lead to misrepresented results on faults on those levels.

Method
The method chosen would have been better if we had found any faults. Now, the method is more of a problem since we could not calculate the Pearson correlation coefficient nor could we calculate the relation between faults and DAL-classification. Some things could have been changed to better fit both cases; no faults found and faults found.

More iterations
AFL mutated and tested the same input data that we fed it at the beginning of the testing. It would be interesting to see what AFL would have done if we restarted the AFL testing with new test cases after an amount of time. We could have setup a Cron job which restarted the session with new test cases.

Debugging test cases
Since the code coverage were not that good on all applications, it would have been interesting to see what could have been done to reach more states in the code. This could have been done by looking through the code coverage data. Then we could have used our configuration file to set static variables which would have guided AFL into those states.

Less time on more applications
Since most of the paths in the code was found at the beginning of the fuzz testing session, it would have been better to test more applications for a shorter time span (say a few hours). The issue with this was that many of the applications had dependencies that needed to be added into compilation flags and these were time consuming to find. It took some time to get a application ready for fuzz testing.

Changing research questions
Our research questions were not good in the sense that they presumed that faults would happen. If those were changed to something that were less quantitative, we would have gotten results that probably would have been more interesting.

Repetitive tasks
A lot of time were put on getting applications to compile. Every application had its own set of dependencies. This was a problem since it took very long time to get the applications to compile the very first time. When the first compilation was done, the Makefile was complete which made the following compilations easier.

Limitations in the developed system
Our implementation has some flaws that could not be fixed at this moment. There is a problem with how AFL mutates the input data. A small change in the input data will usually not be a small change in the SUT. This is because we are using a Mersenne twister random generator seeded by input data to set variables in the SUT. We tried to make it so that not all input data would be random by implementing the configuration file that would be read if the correct byte was set. This allowed some paths to be explored without waiting for the random generator to reach that path.

Memory safety
AFL without any sanitizers tests the SUT for memory safety. The applications has passed this category without any problems. When the programmer follows a code standard that prevents problems with memory safety, that will propagate to the lower DAL-classifications as well. This could mean that applications with lower DAL-classifications in general have less faults than expected.

Results
No errors were found in the applications and the results and that may make it seem like either our implementation was lacking or the tools we used was not good enough for this purpose. There are some interesting aspects of the results though, such as how untested applications would need to be for us to start finding errors.
No faults

Our research questions were aimed at answering whether fuzz testing was a good option or not for testing DAL-classified applications. While our results indicated that it would be hard to find errors in applications that had undergone a lot of testing, it would be interesting to investigate the results from a bigger sample size with more untested applications. Promising code coverage was shown for some applications as well, such as A1. Fuzz testing could perhaps be a viable option for stability testing these applications.

Code generation could be improved

If the code generation in our Dextool plugin is improved further along with our middle layer, more paths could be targeted in the applications thus increasing code coverage. We spent some time implementing the possibility to target certain state machines in the SUT by setting static values on variables, something we did not use for the actual tests. It would not be unreasonable to spend some time debugging the paths AFL can not hit in the SUT, and target these by setting static values on variables that makes us reach these paths. We believe there is great possibilities here, not only for setting up a full test suite for automated testing but also performing extensive crash and stability testing. Everything as a complete process for quality assurance for software with DAL-classification.

Sanitizers

Sanitizers gave us more interesting results as we started finding errors when we used them. These errors turned out to be false positives. We had tested our middle layer and the underlying applications for 4 days before using sanitizers. At that time no errors was found, but when using sanitizers ASAN would start picking up errors. These errors were found in our own middle layer. Our middle layer has not been unit tested at all, and since AFL had not found any errors we have always assumed that the middle layer was stable enough. This would never had caused a crash immediately without ASAN enabled, making the bug avoiding detection until either the SUT would crash by coincidence or maybe it would never get discovered at all.

Trusting the DAL-classifications

DAL-classifications can change. Some applications can therefore have been tested before the final DAL-classification had been selected. This can be the case for the lower DAL-classifications if the developers had expectations for getting a application into a DAL-classification that required more testing than the final DAL-classification. This makes it uncertain towards what requirements the application has been tested. For example E1 could have been tested towards requirements for DAL C while it is actually a DAL E application.

Interfaces

The code generated for the interfaces are from our own interpretation of what each interface should implement and what it should do. This can of course be a misunderstanding of how it actually should work. This can lead to not enough paths being reached by AFL and that would also lead to bad code coverage. While this is probably not an issue, it could make a big difference when trying to reach many of the smaller code blocks.

CONCLUSION

Fuzz testing is a promising alternative to automate testing within avionics software development. It scales well, is fairly smart when testing and can be directed to certain code states if needed, so with knowledge of the SUT it is possible to target certain code paths. It is possible to use AFL within avionics system software testing as well, proven by our implementation. We do not think the result we got is a good indicator how useful fuzz testing is for avionics software as the amount of applications tested and the knowledge of the applications was low.

All the applications we tested had a maturity level considered "done", which means that finding error is unexpected. For our research questions, we would have needed to test many applications which were not considered "done". If we did that, it is more likely we would have found crashes in for example DAL C and DAL E applications making it relevant to calculate the Pearson correlation coefficient.

Another interesting thought is at what stage in the development of these applications would fuzz testing start finding crashes? Which gives further interesting questions to ask in new studies, such as in what stage of software development is fuzz testing most efficient and useful. We suspect application E1 was tested and verified for a higher DAL-classification. Taking that into consideration, it was not out of expectations that we did not find any crashes. The sample size and selection was not ideal for our chosen method, we would have needed more applications to test and also applications that had undergone less testing.

The use of sanitizers was great for finding errors in our code and found errors that AFL could not find directly. When we tested our code we did a 4 day run with AFL without ASAN and UBSAN. We found no errors at all. As we did not found any errors without sanitizers, we compiled the SUT with ASAN and UBSAN support. Then we got some errors, but these were false positives since the errors was in our middle layer. In that matter fuzz testing can be used early in the development of the application to find problems with undefined behaviour and memory problems as it does not require any engineer hours to be allocated looking for bad code.

Considering that we got a decent code coverage in our tests and that it could be improved by having a better understanding of the applications that are tested. The implementation shows that fuzz testing could be used to perform stability tests, not in purpose of finding errors, but simply to ensure a application is stable. The line coverage in application E1 is questionable, further debugging and understanding would be needed to understand why AFL can not target more code paths in this case. It is quite possible that it has many state machines which is hard to reach without guiding AFL, something that is possible with our implementation as long as the user has knowledge about the application.

The results was quite disappointing since we did not find any errors, there is more to be done. One problem we had was that we did not have enough knowledge about the applications we
tested. The applications we tested have a large amount of lines of code (some up to 5000 lines of code) and it would take a lot of time and effort to find state machines that AFL could not get into. If one would have known the structure of the application, it would have been easier to test specific parts known to be problematic. As we only looked into pure crashes of the SUT, another interesting aspect would be to look at what the actual output from the SUT is.

REFERENCES
USER MANUAL

We will go through dependencies and how to build and use our work to fuzz test avionic applications.

Dependencies

- Dextool and its dependencies\(^{17}\)
- GCC 4.9.4+
- American Fuzzy Lop (AFL)
- afl-cov
- lcov

Note: Developed and tested only on Linux and macOS.

Building

To use our implementation, first build all dependencies and make sure they work. The Dextool fork contains our plugin, middle-layer and some examples to test on.

**Building Dextool with fuzz plugin**

Before using commands, make sure you are in Dextool root directory. Then run:

```
$ mkdir build && cd build
$ cmake ..
$ make
```

![Figure 14. Commands needed to compile Dextool and its plugins](image)

**Fuzzing the SUT**

Once again, make sure you are in Dextool root directory. To generate the SUT code, run:

```
$ ./build/dextool fuzz --xml-dir yourapp/namespaces
   --compile-db compile_commands.json --app-name APP_Name
```

![Figure 15. Commands to generate code for the interfaces](image)

This command will generate a Makefile, fuzz.cpp, fuzz.hpp, main.cpp and main.hpp. The fuzz-files will contain the interfaces that it found. Main.cpp and main.hpp will contain the scheduler that will run the complete application including our middle layer. The Makefile can be used to compile the complete application without code coverage support.

Compile with AFL and start fuzzing:

```
$ make -f Makefile_fuzz
$ afl-fuzz --i input/ --o output/ ./a.out
```

![Figure 16. Commands to compile and to start AFL](image)

**Sanitizers**

When you are using sanitizers, such as ASAN or UBSAN, you need to use AFL_USE_ASAN=1 to the Makefile_fuzz rules, just like you would do for the sanitizers. This setting is useful to find memory errors that do not count as a crash.

**AFL variables**

If you are not using any sanitizers, you can add AFL_HARDEN=1 to the Makefile_fuzz rules, just like you would do for the sanitizers. This setting is useful to find memory errors that do not count as a crash.

**Utilities**

AFL has some useful commands for minimizing amount of test files for input and to remove unnecessary data in the files. This doesn’t mean AFL is missing to test things, just that the removed files and bytes were tested by some other test case or byte previously. The commands are:

- afl-cmin: Use to remove unnecessary files from the corpus.
- afl-tmin: Remove unnecessary bytes in corpus file.

Ideally, you run AFL for some hours and then stop. Run cmin and tmin on all files in the queue and then restart AFL with the new corpus as input. This process can be repeated several times to improve corpus, yielding a more exhaustive result each iteration.

**Debugging faults**

When a crash or hang has been found by AFL it just shows up as a number in the interface. To be able to investigate more in the crash, AFL saves the input data to a file which can be found in the specified output directory (-o flag). To debug this further you could either use gdb or AFL’s own script called crash-triage. Crash-triage prints the basic information about all the crashes found in the crashes directory. Gdb can only use one input file at a time but gives better information on what is happening in the application. Gdb is in that case better when some more investigation is needed.

**Guiding AFL**

Guiding AFL can be necessary when AFL has been running for a while but without any new paths found. This can be the case if there is a lot of state machines that are hard to get into. What state machines not reached can be found out by using AFL-cov for better information about what lines which has been reached.

To ease this problem a configuration file can be written. This file can set variables in ports and in what cycles this variable should have this value.

**Configuration file**

```
fun.V0 100--200 1337
fum.V0 100--200 1337
```

![Figure 17. Configuration file with two example variables](image)

The configuration file is straightforward. It is space delimited which means that there is a space between each of the fields. The first field is which variable that should have the value chosen. The second field is during which cycles the variable should have this value.

\(^{17}\)https://github.com/ploq/dextool
should have the value chosen. The third field is the value that the variable should have during the cycles.

The example configuration file says that fun.V0 should have the value 1337 between 100 to 200 cycles. The same file also says that fun.V0 should have the same value and that the variable should be set between 100 to 200 cycles.

Example

```xml
<Interface name="Bar">
  <SubType name="MyInt" type="IntT" min="1" max="2000" unit="km"/>
</Types>

<ContinuousInterface name="Fun"
  direction="From_Provider">
  <DataItem name="V0" type="IntT"/>
  <DataItem name="V1" type="IntT"/>
</ContinuousInterface>

<ContinuousInterface name="Fum"
  direction="To_Provider">
  <DataItem name="V0" type="IntT"/>
</ContinuousInterface>
</Interface>
```

Figure 18. A XML file which is representative of an ARINC 653 interface and is used by the following code example.

The XML file contains two interfaces, Fun and Fum. Fun and Fum then contains two variables or dataitems, V0. The XML file also contains a type with a range specification, it says that the values it handles are between 1 and 2000.

Consider a example application with the following execute function:

```c
void APP_Name_Execute()
{
  if (comp_y->Get_Port().Get_Fun().V0 == 1337)
    if (comp_x->Get_Port().Get_Fun().V0 == 1337)
      port_z->Get_Fun().V1 = comp_x->Get_Port().Get_Fun().V1;

  // Division by zero state machine
  if (comp_x->Get_Port().Get_Fun().V0 /
      comp_y->Get_Port().Get_Fun().V0) {
    comp_y->Get_Port().Put_Fun(uni(rng));
    //random number between 0–8000
  }
}
```

Figure 19. An example application which uses the interface defined in the example XML file.

We have component x, y and z. They are interfaces for the ports residing within them. The ports have all the functions used for modification of any variables. This application will risk crashing on the last if-statement due to a division by zero when the fun.V0 variable in component y is 0. However, the conditions for that to happen is very specific:

- Fun.V0 of component y must be 1337.
- Fun.V0 of component x must be 1337.
- The uniform random generator must generate a 0, and it randomly generates integers between 0 and 8000.

AFL would have a hard time to reach this state many times for it to crash. So in order to reach this state machine consistently with AFL we will have to set some static variables. We want to set component x fun.V0 to 1337 and component y fun.V0 to 1337. This can be done by creating a "config.txt" file in the root directory of Dextool. See the "Configuration file"-section for an example of how the configuration file can look for this example application. The result of doing this for this example is that the application would crash quite often and AFL would register these as crashes.

Code Coverage

gcov

For code coverage support you will need to compile the application with gcc and with the flags -fprofile-arcs -ftest-coverage. This will generate a new executable file and .gcno files. Next time you start your program it will collect code coverage data that you can read with either gcov or lcov. This is used by AFL-cov to get code coverage data from your AFL run.

AFL-cov

AFL-cov is written as a wrapper for lcov. AFL-cov can be used to get coverage data during or after an AFL run. It takes all the input data that AFL has saved and then runs your program with those files.

To use AFL-cov you either need to start it before you start AFL, during AFL runs or after. If you are running AFL-cov as the two first cases you’ll need to have the –live flag. If you are not you can omit that flag.

```bash
$ afl --coverage-cmd "/a.out < AFL_FILE" --code-dir .
```

This command will look for AFL input data files in output/. The command it will run to start your application is taken as a parameter called coverage-cmd, it is important that you do not omit the AFL_FILE as that is replaced with the correct path to an AFL input data file.

If you already have ran AFL on your program and want to get code coverage data for that run, you’ll need to recompile your program with the flags covered in the gcov-section. After that you can start AFL-cov with the –live flag removed.