UNIVERSITÀ DEGLI STUDI DI MILANO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in
Sicurezza dei Sistemi e delle Reti Informatiche

Security aspects of avionics protocols

RELATORE
Prof. Valerio Bellandi

TESI DI LAUREA DI
Giulio Ginesi
Matr. 872795

CORRELATORE
Prof. Ernesto Damiani

Anno Accademico 2017/2018

Ai miei genitori

Acknowledgements

I would like to thank my family for everything they have done for me, especially my mother and my father and Sarah for always being there.

I would like to thank Andrei Costin for all the help and support he gave me during this research, for the high standards of his teachings and last but not least for his friendship. I am proud of calling him a friend more than a teacher.

My gratitude also goes to Valerio Bellandi for his valuable suggestions, and to my flatmate Cem that made my staying in Finland easier, happier and funnier.

I would also like to thank all my friends and the people I met during the Erasmus period who made me feel at home.

To conclude I would like to thank all my friends that always supported me and believed in me: Francesco, Giorgio, Emanuele, Michele, Sergio, Giulia, Costanza, Pietro, Davide and all the others that have always been there for me.

Contents

1 Introduction **1**

2 Scenario **5**
2.1 OldGen . . . 8
2.2 NextGen and SESAR . . . 11

3 Testing and Validation **17**
3.1 Software testing . . . 17
3.1.1 Black Box . . . 17
3.1.2 White Box . . . 18
3.1.3 Gray Box . . . 18
3.2 The fuzzing world . . . 19
3.3 State of the art . . . 20
3.3.1 American Fuzzy Lop . . . 20
3.3.2 Peach . . . 24

4 Analysis **25**
4.1 Hardware Setup . . . 26
4.2 Software Setup . . . 26
4.2.1 Datasets . . . 27
4.2.2 Tools . . . 29
4.3 Proceedings . . . 30

5 Results **37**
5.1 Results and Caveats . . . 37

6 Conclusions and Future Work **41**

Acronyms

ACARS Aircraft Communications Addressing and Reporting System.

ACAS Airborne Collision Avoidance System.

ADS-B Automated Dependent Surveillance - Broadcast.

ADS-C Automatic Dependent Surveillance - Contract.

AFL American Fuzzy Lop.

AOC Air and Space Operations Center.

ATC Air Traffic Control.

ATM Air Traffic Management.

ATS Air Traffic Service.

CRC Cyclic Redundancy Check.

DF Downlink Format.

EASA European Aviation Safety Agency.

EFB Electronic Flight Bag.

EHS Enhanced Surveillance.

ELS Elementary Surveillance.

ENAC Ente Nazionale Aviazione Civile.

ENAV Ente Nazionale Assistenza Volo.

ETSO European Technical Standard Order.

EUROCAE European Organization for Civil Aviation Equipment.

FAA Federal Aviation Administration.

FEC Forward Error Correction.

FIS-B Flight Information Services - Broadcast.

GA	General Aviation.
HF	High Frequency.
IC	Interrogator Code.
ICAO	International Civil Aviation Organization.
IFF	Identification Friend or Foe.
IFR	Instrument Flight Rules.
IoT	Internet of Things.
MTOW	Maximum Takeoff Weight.
NOTAM	Notice To AirMen.
RA	Resolution Advisory.
RTCA	Radio Technical Commission for Aeronautics.
SAR	Search and Rescue.
SESAR	Single European Sky ATM Research.
SPI	Special Identification Pulse.
SSR	Second Surveillance Radar.
TIS-B	Traffic Information Services - Broadcast.
UAT	Universal Access Transceiver.
UAV	Unmanned Aerial Vehicle.
VHF	Very High Frequency.

Chapter 1

Introduction

Over these last 15 years the introduction of more and more automated systems on board of aircrafts and inside the Air Traffic Control and Air Traffic Management has seen an incredibly fast growth compared to the past. Speaking of the General Aviation sector, which includes all the non-military aircrafts different from scheduled air services and non scheduled salaried air transport, this transformation is even more evident. In particular more and more systems are designed as IoT devices with a strong decision-making autonomy, the Garmin G1000[1] is a perfect example as it is a Glass Flight Deck which integrates all the avionics and can also be connected with an Electronic Flight Bag that lets the pilot perform some operations directly on an iPad.

Since new air carriers must undergo long processes of research and development, which take decades before a "new generation" airplane can go into service, the avionics must at some extent follow this timing and therefore a too rapid or sudden development, in the commercial aviation is not possible. But, instead, a balance has to be kept between avionics and aircraft development.

New avionics protocols, having been designed with an IoT paradigm in mind, are now deeply integrated inside organization networks and are at the base of services like Flightradar24. Moreover, such protocols are used, not only by aircrafts, but also by drones, gliders and similar; while it is true that this enhances the safety of all the flying objects and allows almost all of them to be connected, on the other hand a possible vulnerability inside those protocols can lead to dangerous and catastrophic consequences.

The protocols examined are: **ADS-B**, **FIS-B**, **ModeS** that are messaging protocols and **1090ES**, **UAT** which are lower level transport protocols. Previous researches had already demonstrated the insecurity of such protocols in different ways and more recently the Department of Homeland Security successfully conducted a *"non-cooperative RF hack"* on a Boeing 757 which gave them complete control over the aircraft [1].

The innovative approach of my research is the use of Fuzzing to test avionics protocols. This is a software testing method based on the generation of completely random or mutated inputs to trigger crashes in the target binary. Such method has already been proven effectively in finding important bugs such as Heartbleed, OSX Kernel memory corruptions, libjpgturbo corruptions and many more. The specific tool used during the research is American Fuzzy Lop, a feature-rich and open source fuzzer developed by a Google employee and widely used inside the organization. A particular fork of this tool, *afl-unicorn*, has been used as it was developed specifically to deep test programs interacting with Radio Frequencies and it allows to selectively emulate and test programs for a different architecture. The very nature of such programs, which are specifically designed to be noise resistant, employ different strategies in order to correct possible errors, which bring some challenges in the Fuzzing attempt.

---

[1] `https://buy.garmin.com/en-US/US/p/6420`

This work has a dual importance because the tests are conducted on open source implementations of such protocols, which allows to test the IoT device as well as the protocols themselves. Although the main focus was on finding vulnerabilities on the protocol side a potential presence of software specific errors might give hints on most problematic points. Moreover, if a vulnerability is present in the open source software there is a high possibility that it would be present in the commercial version, nevertheless a vulnerability might also be present only on the commercial side and vice versa.

The fast growing progress in the avionics hardware and related protocols must go along with security researches; therefore it is paramount to develop knowledge and tools that can help discover vulnerabilities in **NextGen** and avionics as easy and as early as possible.

The main body of my dissertation is organized as follows:

In *Chapter 2: Scenario* there is an overview of the present situation and an analysis of the current state of the art of the avionics protocols. A clear description of the two families of protocols, **OldGen** and **NextGen**, is given in order to show the possible critical points. While there is an overview of the most relevant avionics protocols, particular attention is given to the ones analyzed during my research.

In *Chapter 3: Testing and Validation* some basic information on the most common software testing methods is given with particular focus on the fuzzing. Two of the state of the art tools in term of binary fuzzing are then described: **AFL** and **Peach**. I focused on **AFL**, since this is the tool used during the research, giving detailed information on the techniques used by this fuzzer and on how to read its interface. Moreover a description of a modified version of **AFL**: *afl-unicorn* is provided.

In *Chapter 4: Analysis* the experimental setup of the research is illustrated, dividing the chapter into **Hardware Setup** and **Software Setup**. The first part is about the hardware used while the second one describes the binaries tested, the datasets used and how they have been acquired and, lastly, the software and scripts specifically created for this research.

In *Chapter 5: Results* the results and caveats encountered during the research are described. In particular detailing the problems that have been encountered with the flavored version of **AFL**: *afl-unicorn* and with the underlying emulator, the Unicorn Engine. Also there are some considerations on the datasets and on the programs tested.

In *Chapter 6: Conclusions and Future Work* there are the conclusions and an overall analysis of what has been done. More importantly some ideas and proposals to further continue and expand this work are discussed.

Chapter 2

Scenario

The technological explosion of the last years influenced also the aviation sector. Better planning, more efficient aircrafts and many other management improvements brought the price of airline tickets down making traveling by plane accessible to anyone. More recently the drone market rapidly grew to the point where every professional and amateur video maker must have at least one Unmanned Aerial Vehicle (**UAV**) to capture great aerial shots that in the past could only be done by using helicopters. Moreover companies like *Amazon* and *DHL* successfully tested a parcel delivery service operated by drones [2] while other companies like *Facebook* and *Google* started testing an affordable and reliable method to bring internet connection to remote areas of the planet [3]. Furthermore, companies like PrecisionHawk [4] developed accurate sensors to be used with different drones that allow the user to perform many different measurements. **UAVs** are spreading also in the 4.0 agriculture, Search and Rescue (**SAR**) and general emergency fields [5]. Nowadays Military operations are more and more drone-centered, therefore armed forces, from infantry to flying squadrons, are equipped with different kinds of **UAV**. All this sudden progress has led to overcrowded skies in which the innovation in the Air Traffic Management (**ATM**) field must be continuously updated. Moreover new systems have been built with a strong push of implementing IoT paradigm in respect with the **ATM**, **UAV** and avionics systems.

Airplanes can no longer be seen as the main users of the sky. A wider spectrum of flying objects should be considered given the different nature of their functions. Aircrafts are divided into different categories depending on the maximum altitude, their weight and other parameters. Commercial drones, such as *Amazon* or cinematography ones are now developing capabilities similar to small aircrafts in terms of performance. Similarly *Facebook* Aquila drones have the same wingspan as a Boeing 737 and they can fly at a considerably high altitude[6]. Also Project Loon internet balloons (*Google*) travel at high altitude and through busy airspace. In addition, there are many other objects like gas balloons, helicopters and gliders operating in specific areas and airspace class that must be tracked, managed and monitored efficiently to ensure high safety standards.

The development of a new technology brings a lot of challenges, some of them in the cybersecurity realm. In particular, Costin and Francillon [7] demonstrated in 2012 that it is easy and cheap to perform various operational attacks on ADS-

B protocols in particular, and **ATM** and **ATC** systems in general. Subsequently, various weaknesses were demonstrated in various Air Traffic Control (**ATC**) protocols [8, 9] and avionics sub-systems. More recently, in November 2017 the news broke that back in September 2016 a team from Department of Homeland Security (DHS) was able to hack a Boeing 757 parked at the airport. The hack was described as *"remote, non-cooperative, penetration"* [1]. Though the details were scarce because of the classified label, it was acknowledged that the team, consisting of industry experts and academics, accomplished the hack by accessing the aircraft systems through radio frequency communications.

Aiming at more security and in order to overcome some limitations of the old generation of protocols, like the absence of radar coverage in certain areas of the planet (e.g. over the ocean), a modernization of the radar and communication system was launched starting from the 80s up to the present days with the development of a new generation of protocols. Standards and guidance documents for aeronautics are defined by different cooperative organizations, the major of which are Radio Technical Commission for Aeronautics (**RTCA**) in the United States, European Organization for Civil Aviation Equipment (**EUROCAE**) in Europe and International Civil Aviation Organization (**ICAO**) which is a UN organization. The above mentioned organizations are non governative therefore they produce standards, guidelines or rules that will have to be adopted by the Federal Aviation Administration (**FAA**) in the United States, by the **EUROCONTROL**[1] and the single national aviation authority of the various European countries (*ENAC/ENAV* in Italy). This fragmentation of the establishment, the absence of a globally shared guidance on the standardization of the aviation sector (which is partially an ICAO task[2]) and the development of different standards to accomplish the same task have created an unclear situation about standards and regulations. In particular **NextGen** has shown problems in the definition of a globally accepted family of protocols as it can be seen in the relevant section.

Aircrafts can be tracked either in a **cooperative** way and a **non-cooperative** one:

- The classic radar (or Primary Radar), based on electromagnetic waves is able to give information only on the position of an object in the sky. Such system gives a real-time image of the portion of sky it is observing, including aircrafts, birds, clouds, and any other object in its visual range making it a basic and fairly unreliable system. Since this method requires no interaction with the aircraft that is being tracked, it is called **non-cooperative** system.

- Second Surveillance Radar (**SSR**) is an evolution of the above system. In addition to the spatial position of the aircraft, it gives further information depending on its mode of operation. This system requires an active cooperation from the aircraft which must reply to the interrogations received making it a **cooperative** system.

---

[1] The organization that provides unique centralized platform for civil and military aviation coordination in Europe.

[2] Art 1. The contracting States recognize that every State has complete and exclusive sovereignty over the airspace above its territory.[10]

The cooperative systems allow to have a bidirectional data flow between aircraft and ground as well as aircraft and aircraft. This system stems from the military Identification Friend or Foe (**IFF**) one which was developed during the Second World War. The actual civil system uses a 4 digit code called "Transponder Code" or **"Squawk"** to identify the aircraft. This communicates via a **transponder** which handles the incoming interrogations and the responses. In Figure 2.1 is an overview of all the protocols and their division between the old generation and the new one which is still in deployment.

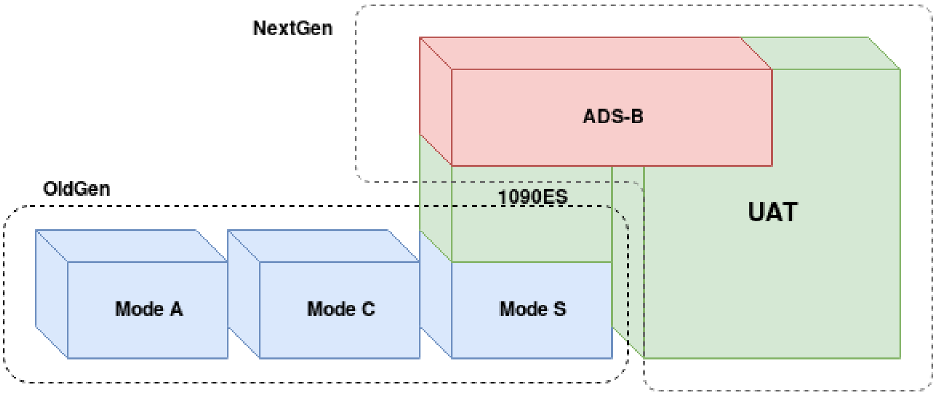


Figure 2.1: All generation of avionics protocols

Having a continuous data flow between the aircraft and the ground is beneficial not only for the **ATM/ATC** but also for the airline. As a matter of fact:

- Flight controllers can obtain much more information about a single aircraft, his intentions and the status of the flight. Moreover satellite based ADS-B allow to trace planes in areas where no radar coverage is available.
- The airline can keep track of its fleet in real time thus allowing a better planning of the turnover and, if needed, quick deployment of a substitute airplane.
- The maintenance branch of the airline can track in real time any anomalies reported by the instruments and sensors performing a remote analysis which will help pilots make better decisions in the troubleshooting process.

As previously mentioned the current avionics protocols can be divided in two families: **OldGen** and **NextGen**. The **OldGen** is currently deployed and used globally while the **NextGen** is standardized and set to be fully deployed by 2020. For this reason, to comply with regulations and to enhance safety Project Loon balloons are equipped with both OldGen and NextGen hardware [11] while some DJI drones are compatible with NextGen protocols [12].

In addition to their use in aircrafts both **OldGen** and **NextGen** protocols are now integrated in a big network of IoT sensors. It is currently estimated there are no less than 32000 air-traffic sensors/receivers integrated into crowd sourced projects, in particular about 17000 in FlightRadar24 [13], around 15000 in Flightaware [14], and at least 700 in OpenSky-Network [15]. These sensors/receivers can be either general purpose devices such as RaspberryPi (RPi) and routers plugged with USB RTL-SDR dongles, or can be special-purpose ADS-B receivers using specialized FPGA implementations. Moreover, it is expected the number of aircraft, including air carrier and General Aviation (GA), that are equipped or upgraded with avionics embedded devices supporting ADS-B and other NextGen protocols will surpass 100000 [16]. These numbers do not account for the considerable number of devices that are deployed at various ATC towers and that process avionics RF protocols. Therefore, given the number and the Critical Infrastructure (CI) functions of those devices, it is important to secure those systems and to be sure that no flaws exist in the protocols.

2.1 OldGen

The first generation of protocols is fairly simple; it uses the 1030MHz frequency for interrogation and the 1090MHz frequency for replies. The first two protocols are **Mode A** and **Mode C**.

Mode A is the simplest because it responds to an interrogation request just by broadcasting the **squawk** code which was previously assigned to the pilot by the controller. In this way the controller can identify the aircraft on his screen by such code. In addition to this, the pilot can manually generate a special response called *"Ident"* or *"SPI"* (Special Identification Pulse) which is used to highlight the aircraft on the controller screen.

Mode C is an extension of the previous protocol which, in addition to the transponder code, sends information about the altitude and the pressure. This kind of transponders are often referred to as **Mode A/C**.

Mode S is the newest protocol of this family and it is an hybrid between the two generations. It was born as part of the **OldGen** family but at the same time it is also part of the **NextGen** as it can be seen in Figure 2.1. As clearly explained further in the text, **Mode S** in the years has undergone various enhancements and refinements in order to be used as an easy-to-deploy and affordable **NextGen** protocol. **Mode S** ground stations and transponders support both all call interrogations and selective interrogations, in particular each aircraft is identified by a unique 24-bit address which is part of the aircraft registration documents and should never be changed. The address is transmitted with every **Mode S** reply, which allows a SSR station to perform an all call interrogation in order to acquire the address of each aircraft which can then be used to selectively interrogate them. Each ground station is identified by a 4-bit[3] Interrogator Code (IC). In this way single aircraft interrogation as well as lock-out of the aircraft can be performed in order to avoid multiple pickups of the same aircraft by different SSR stations. **Mode S** messages can be of 56 or 112 bits and they are structured as in Figure 2.2.

In **Mode S** messages the first 5 bits identify the type of message called Downlink Format (DF), then the other bits, except for the last 24, are message dependent. **Mode S** comes in two versions: Elementary Surveillance (**ELS**) and Enhanced Surveillance (**EHS**). **ELS** is the basic version that has been described

---

[3] For aircraft complying with ICAO Annex 10, Volume IV Amendment 73 a 6-bits code can be used.

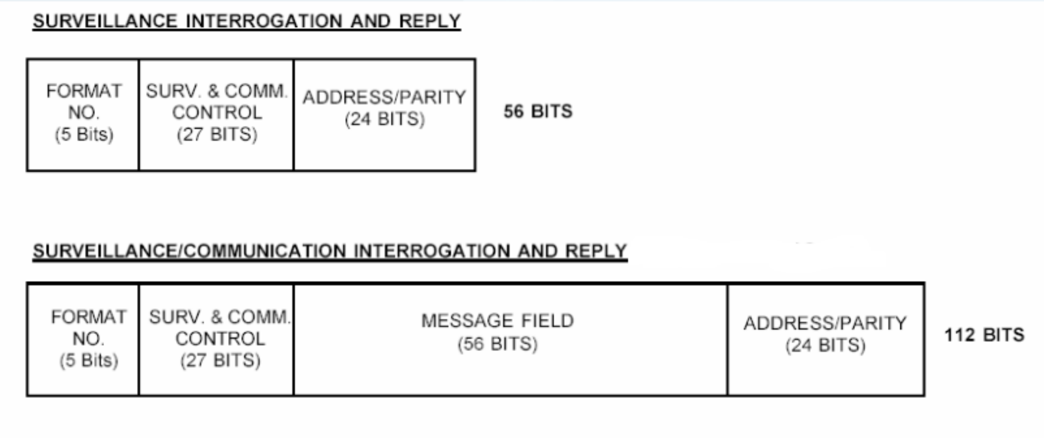

SURVEILLANCE INTERROGATION AND REPLY
FORMAT NO. (5 Bits)
SURV. & COMM. CONTROL (27 BITS)
ADDRESS/PARITY (24 BITS)
56 BITS
SURVEILLANCE/COMMUNICATION INTERROGATION AND REPLY
FORMAT NO. (5 Bits)
SURV. & COMM. CONTROL (27 BITS)
MESSAGE FIELD (56 BITS)
ADDRESS/PARITY (24 BITS)
112 BITS

Figure 2.2: Mode S messages

above and is the minimum standard required in European Mode S airspace. **EHS** introduces some improvements:

- Reduced workload for the pilots and the controller, since some radio communications can be avoided when the controller has access to information such as the Magnetic Heading or the Selected Altitude.

- Improved Situation Awareness, since the controller can have access to a greater amount of data like Vertical Climb Rate, Magnetic Heading, Selected Altitude and similar.

- Safety Enhancements brought by data such as Selected Altitude which permits the controller to check if the aircraft is following the instructions and avoid potential collisions or level bust in advance [17].

Mode S EHS is also a minimum requirement for some categories of aircraft flying in European Mode S airspace.

All the above protocols are at the base of the Airborne Collision Avoidance System (**ACAS**). This system requires all flying objects to be equipped with **Mode A/C** or **Mode S** transponders. In this way it can keep track of the surrounding airplanes and, in case of colliding traffic, propose vertical Resolution Advisory (**RA**). **NextGen** systems have enhanced **RA** capabilities. It is clear that this is a critical part of the system and, if compromised, can lead to dramatic results such as mid air collision.

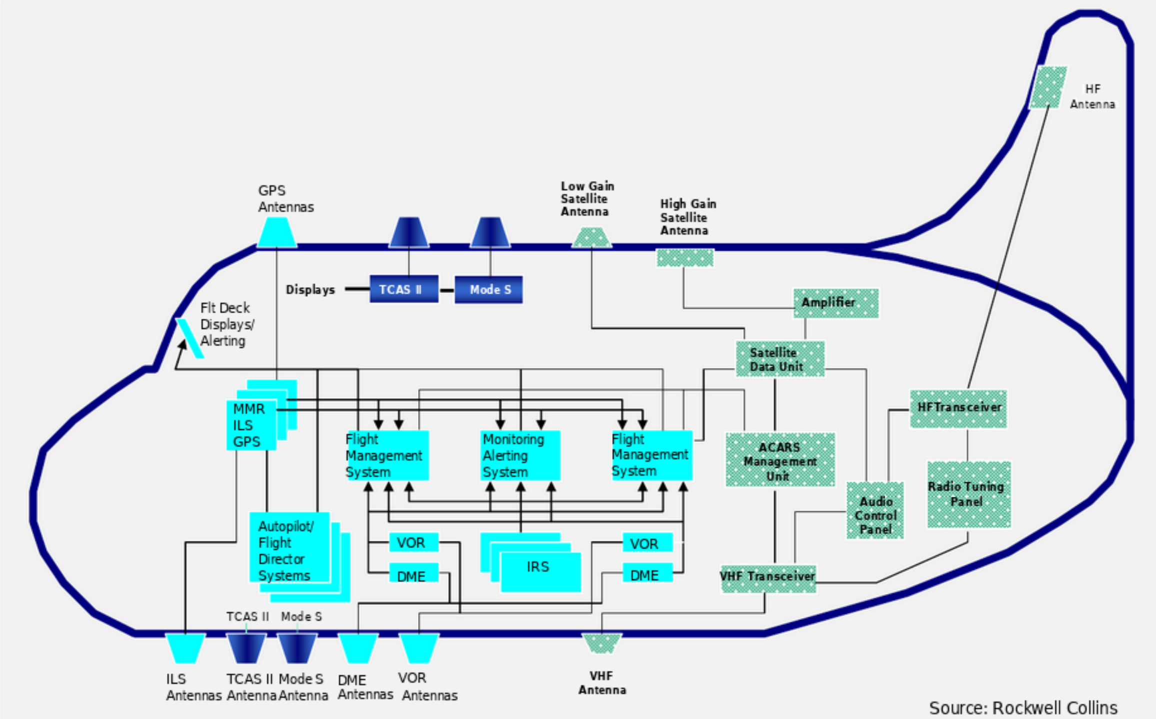
Mode A/C and **S** are the current standards and they are widely used, notably according to the Italian regulation all aircrafts must be equipped with at least a **Mode A/C** transponder and in some particular cases with a **Mode S** transponder as it can be seen in GEN 1.5-3.1 of AIP Italia [18].

Beside the basic protocols described here, a more sophisticated one lays between the two generations. The Aircraft Communications Addressing and Reporting System (**ACARS**) which has been in use since 1978. At first it relied only on VHF radio channels but in the years it has been improved to add other transmission means to expand coverage. It is also now deeply integrated into the aircraft systems giving it access to a large number of data and the ability to operate autonomously. **ACARS** messages can be delivered via 3 different transmission means: VHF Data Link, HF Data Link and satellite. Depending on the position of the aircraft, one method can be better than the other, specifically VHF works only in line of sight while satellite communication (SATCOM) is not available at the poles. **ACARS** messages can be of 3 different types:

- Air Traffic Control messages. Used in some busy airports as an alternative to the radio. And in routes where no radio contact is possible (e.g oceanic routes).
- Aeronautical Operational Control (AOC) and Airline Administrative Control (AAC) used to send documents to the aircraft as well as to receive error messages or information on the status of the flight.
- Free Text Message.

Messages from the aircraft can be pre-configured so that they are automatically delivered to the appropriate recipient based on the message type; in the same way ground-originated messages can be configured to reach the correct aircraft. This messages cover a fundamental part of the bidirectional aircraft-ground data link and their content can be of utmost importance. For example the Air France flight 447 which disappeared from the radars on 1 June 2009 sent in his final moments 24 **ACARS** messages, some of them indicating anomalies and errors. In the first moments those messages were the only clue to understand what had happened and to locate the aircraft [19].

Current minimum system requirements of an airliner can be seen in Figure 2.3. However by 2020 all the **Mode S** transponders must be upgraded to support **1090ES NextGen** protocol.


HF Antenna
GPS Antennas
Low Gain Satellite Antenna
High Gain Satellite Antenna
Displays
TCAS II
Mode S
Amplifier
Flt Deck Displays/ Alerting
Satellite Data Unit
MMR ILS GPS
Flight Management System
Monitoring Alerting System
Flight Management System
ACARS Management Unit
HFTransceiver
Audio Control Panel
Radio Tuning Panel
Autopilot/ Flight Director Systems
VOR
DME
IRS
VOR
DME
VHF Transceiver
TCAS II
Mode S
ILS Antennas
TCAS II Antenna
Mode S Antenna
DME Antennas
VOR Antennas
VHF Antenna

Source: Rockwell Collins

Figure 2.3: Current avionics minimum requirements

Although the **OldGen** provides some benefits in term of position accuracy and details for the controllers, there are still some problems:

- The information provided is still low and, except for the enhancements introduced with **Mode S**, it only provides a small aid during the flight.

- Using such protocols is still better than the radar alone; however, the accuracy level is still low and usually not much information can be carried so a intense verbal interaction with the controller is still required.

- There is no effective way to confirm the identity of an aircraft and the validity of the data that it is sending, thus allowing for no effort Man in the Middle and other attacks.

Different approaches to secure such protocols (both **Old** an **NextGen**) have been proposed [20]. Unfortunately, none of them will be adopted in the imminent transition to the **NextGen** but hopefully some may be used in the near future as there are studies and official documents about that [21].

2.2 NextGen and SESAR

Both **NextGen** and **SESAR** (Single European Sky ATM Research) are efforts to modernize the air transportation system respectively from the United States government and from the European Union along with other private parties. Both programs aim to achieve a higher degree of safety enhancing communications, navigation, surveillance technologies thus enabling all the users of the sky, even the future ones, to virtually see each other. The introduction of those new standards will also bring a reduction of costs through better **ATM** and enabling low visibility operations. The efforts are coordinated in order to assure a globally accepted common standard. For this reason from now on I will make no distinction and refer to both protocols as **NextGen**. However, many different protocols go under the **NextGen** family and some of them are not shared between Europe and America even if there are plans to do that.

The main component of this system is Automated Dependent Surveillance - Broadcast (**ADS-B**). *"The American Federal Aviation Administration (FAA) as well as its European counterpart EUROCONTROL named ADS-B as the satellite-based successor of radar."*[22].

ADS-B is an automatic system which broadcasts aircraft sensors information to the outside world. It is divided in "ADS -B Out" which requires just a transponder able to properly encode messages and "ADS -B In" which requires a receiver, a computer and an interface to display the data. **ADS-B** messages are also picked up by ground stations which feed the data to a central system where they are used, in combination with other data (e.g radars), to create a Traffic Situation Picture.

Two main protocols were proposed to deliver **ADS-B** messages:

- **1090ES** (Extended Squitter)
- **UAT** (Universal Access Transceiver)

1090ES it is the current global standard for **ADS-B** in commercial aviation. In Europe **1090ES** is required for IFR aircrafts with a MTOW exceeding 12,566 pounds or maximum cruise airspeed faster than 250 KTAS. All aircrafts must comply with this regulation by 2020 [23].

In USA **1090ES** will be mandatory for all aircrafts after June 2020 and is the only technology that should be used when flying above 18,000 feet [24]. An overview of the USA supported technologies in the various airspace can be seen in Figure 2.4.

This protocol is backward compatible with the **OldGen** since it uses the same frequencies and, in particular, a transponder supporting **Mode S** messages can be easily updated to support **ADS-B**. Therefore **ADS-B** information is simply carried inside a 112 bit **Mode S** message as it can be seen in Figure 2.5. The message structure is as follows:

- The first 8 bits are used to identify the message and in particular the first 5 bits identify the type of message (10001 and 10010 for an **ADS-B** message) while the next 3 bits represent the capabilities which are different for every message type.

- The next 24 bits are the ICAO address of the aircraft. Then there are 56 bits containing data and the last 24 bits of CRC code. These last bits are particularly important as they allow to correct several errors in one message.

All these characteristics are in common with the **Mode S** messages so **ADS-B** messages encode their type and information in the 56 bits reserved to data. Data segments contains the first 5 bits indicating a type code for the **ADS-B** message and then other bits encode specific information depending on each message [17].

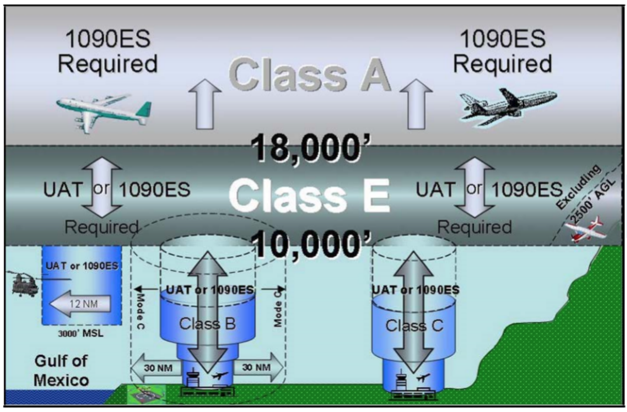

1090ES
Required
Class A
1090ES
Required
18,000'
UAT or 1090ES
Required
Class E
UAT or 1090ES
Required
Excluding 2500' AGL
10,000'
UAT or 1090ES
12 NM
3000' MSL
Gulf of Mexico
UAT or 1090ES
Mode C
Class B
Mode C
30 NM
30 NM
UAT or 1090ES
Class C

Figure 2.4: USA airspace

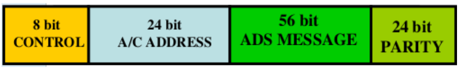

8 bit
CONTROL
24 bit
A/C ADDRESS
56 bit
ADS MESSAGE
24 bit
PARITY

Figure 2.5: 1090ES Message

UAT (Universal Access Transceiver) is not actually Universal as it is not yet widespread like **1090ES**. This protocol is mainly deployed in North America and it is starting to be deployed also in China. It is entirely part of the **NextGen** and it was developed specifically to be used with **ADS-B**. It uses the 978MHz frequency with a 1 MHz bandwidth and requires new dedicated hardware to function properly. The structure of a **UAT** message can be seen in Figure 2.6, it is particularly interesting how the CRC (Cyclic Redundancy Check) is combined with standard FEC (Forward Error Correction) resulting in an extremely low rate of undetected errors and an improved noise and interference resistance.

Moreover, since it is a newly designed protocol built to be future proof it allows to carry more information compared to **1090ES**, such as weather reports,

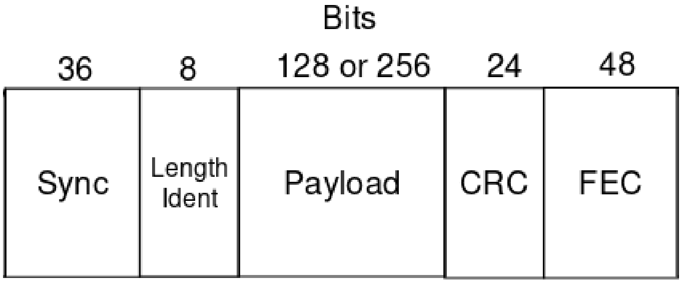


Figure 2.6: UAT message

pilots reports and other messages (like NOTAMs) using **FIS-B** (Flight Information Services - Broadcast) and **TIS-B** (Traffic Information Services - Broadcast) services, which are ground based and broadcast to "ADS -B In" equipped aircrafts information on weather and traffic detected by ground radars as well as coming from "ADS -B Out" equipped planes. **FIS-B** service is also capable of delivering real time weather images captured from the United States Weather Surveillance Radar (**NexRad**), these messages have the following format:
<type> *<hour>:<minute>* *<scale>* *<north>* *<west>* *<height>* *<width>* *<data>*.

<type> can be Regional (value 63) if the image represents only a specific region of the map. Conus (value 64) if the image represents the whole country (USA).
<scale> can be 0, 1 or 2 and indicates the resolution of the image. A higher value means lower resolution.
<north> *<west>* *<height>* *<width>* the first two elements refer to the respective edge of the block, their value is in arcminutes. The last two elements are the height and width of the block expressed in arcminutes of latitude and longitude.
The *<data>* portion can be encoded in different ways:

- Run Length Encoding (**RLE**), particularly powerful as it has a high decompression speed.
- Empty Block representation, representing one or more blocks that are completely empty of data.
- Huffman Encoding, a common compression algorithm used in jpg images. It is based on a binary tree conveniently constructed using the occurrence of each character or element [25].

US government is encouraging General Aviation (GA) aircrafts, not flying in Class A airspace, to use a **UAT** transponder. In this way there is less pollution of the 1090MHz frequency and they can receive more information such as weather data.

Multilateration is a technology that uses different signals to accurately locate an aircraft. It was initially developed for military purposes but was then adopted in the civilian world and used to confirm the position transmitted by **ADS-B**. It employs strategically placed antennas that listen to different replies (**Mode A**,**C**,**S**, military **IFF** and **ADS-B**) transmitted from an aircraft. Since a single aircraft will be at a different distance from every ground station, the replies to each station will have a different time of arrival; this difference can then be used to compute the precise position of an aircraft. A graphical example of how the multilateration works can be seen in Figure 2.7. This is the inverse of triangulation and requires no additional hardware since the ground stations are able to acquire replies from a multitude of different protocols. Which makes **multilateration** a faster method to locate an airplane since information can be acquired at a considerable higher rate. However, it has been empirically demonstrated by Bran Haines in [26] that such a system is not mandatory and rarely used.

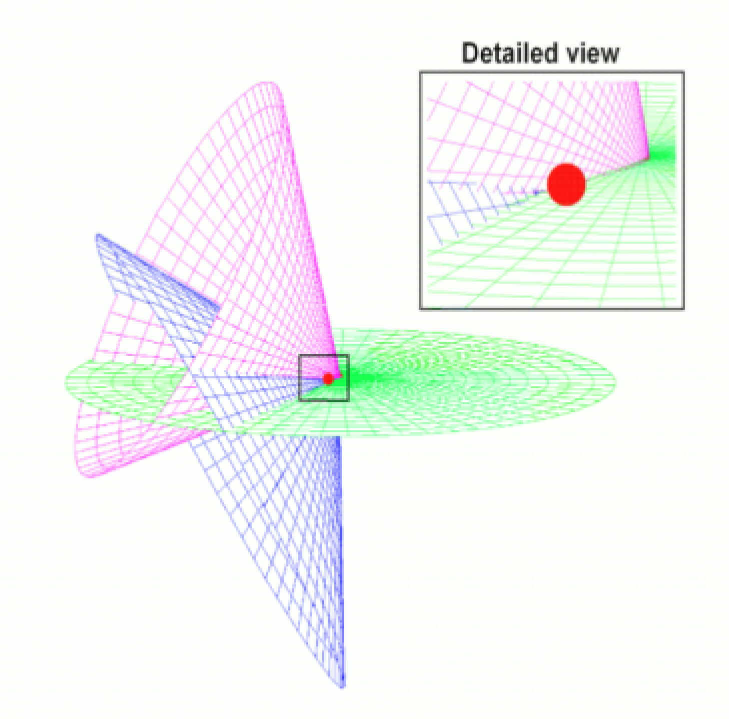


Figure 2.7: Example of multilateration from [27]

NextGen protocols will also replace a big part of the voice communications not only between pilots and air traffic control but also between air traffic controllers themselves using the **UAT** data link and the previously mentioned **ACARS** messages. In particular **ADS-C** (Automatic Dependent Surveillance - Contract) automatically aggregate data such as aircraft position, altitude, speed, intent and meteorological data from on-board sensors. The generated report can then be sent to an **ATS** (Air Traffic Service) unit or **AOC** (Air and Space Operations Center) facility ground system for surveillance and route conformance monitoring [28]. These ground stations must authenticate themselves on the aircraft system to require a contract.

Contracts can be of 4 types:

- Periodic: the **ATS** specify the time interval at which the aircraft sends the report.
- Demand: a single report requested from the ground station. This request does not affect any other contracts that might be present.

- Emergency: these reports are tagged as "emergency" reports and will be highlighted to the **ATC**. Emergency reports can be generated manually by the crew or as a consequence of triggering another type of emergency system.
- Event: the **ATS** unit can specify the event (limited to 1 per aircraft) at which the report will be sent. The contract can contain multiple event types (e.g lateral deviation, vertical rate change ecc).

Security researchers in this field are focusing on **ADS-B** and various researches [7, 8] have already demonstrated the insecurity of such protocols in the **ATC/ATM** world. Few researches have been done on the aircraft side and it is now public and clear that this is a feasible entry point to an aircraft system [1].

Chapter 3

Testing and Validation

3.1 Software testing

Software testing can be conceptually divided into two main groups: **Static Testing** and **Dynamic Testing**. While **Static Testing** is based on a (human) review of the source code, without executing the program, the **Dynamic Testing** is mainly accomplished with the use of automated tools at run time. In particular, many different tests can be conducted on a single piece of software by an autonomous program. The results of such tests are then used to correct bugs in production (e.g. agile development) or to exploit such bugs if the test is conducted by a malicious counterpart.

The research was carried out mainly using **Dynamic Testing** and in particular Fuzz testing mixed with other methods such as source code review and reverse engineering. A **Dynamic Test** can be of three types: **Black Box**, **White Box** and **Gray Box** depending on the level of knowledge and access to the source code required.

3.1.1 Black Box

For this methodology no access to the source code or knowledge of the design specifications and internal mechanisms of the program is required. The test is based only on what can be observed, which means to give the target different inputs and monitor its behavior. This definition reflects exactly the interaction that a typical user might have with the program. Which are exactly the cases that this kind of testing wants to examine and it should test typical and atypical user behavior. Black Box testing is widely used and it includes all the techniques where the only available information comes from the user interaction with the application. One example among all is SQL injection, where the test is conducted against a plain web page without any previous knowledge of the logic of the servers and the various scripts. Only interacting with them allows the user to get an idea of the inner working mechanisms. The test can be performed directly by the user (manual testing) or using ad hoc programs (automated testing) such as SQLmap[1] in this case. Black Box testing has many advantages: mainly, it does not require access to the source code so it is always applicable; moreover, even in the presence of source code, it can still be effective. In addition to this, since it is not source code dependent, a successful test case against one program (for example a zip extractor) can be easily reused to test any programs that implement the same functionality.

---

[1] `http://sqlmap.org/`

However, due to its simplicity this approach can only scratch the surface of the application that is being tested, achieving an extremely low code coverage. Moreover complex vulnerabilities, which requires multiple attack vectors, will never be triggered and can only be discovered with a White Box or static approach.

3.1.2 White Box

This methodology is nothing more than a source code review or analysis and it can be done either by a human being or, more often, by automated tools. Human analysis in not always feasible and might be inaccurate, especially in large projects, because it is easy to miss one line or make simple mistakes. For this reasons automated source code analysis is the means used in a White Box approach. This method has many advantages mainly regarding code coverage since it is possible to test all the paths for vulnerabilities. However, source code is rarely available for projects outside the open source community and this approach still requires a human interaction to review the results and identify the parts that are actually related to security problems. Moreover, reviewing a large project, even with the use of automated tools, might require significant time to complete, effectively bringing the performance down to the level of the Black Box approach.

3.1.3 Gray Box

The definition of this technique is quite trivial because several different methods can be seen as Gray Box approach. This takes the Black Box method as a starting point augmenting it with the use of common reverse engineering and binary analysis techniques using disassemblers, debuggers and similar tools. The majority of these tools still requires human interaction and, reverse engineering a program can often be a tedious and hard job. However, there are some automated tools that aim to facilitate this analysis, such as Ropper[2] which provides information about a binary and automatically searches for particular vulnerabilities (ROP and JOP) or Frida[3] which allows the user to inject snippets of code inside a Black Box binary. For these reasons the Gray Box technique can efficiently improve the coverage produced by the classic Black Box approach still without requiring access to the source code. Moreover this approach becomes very powerful if the source code is available as it can modify the program during the compilation phase to efficiently trace the execution flow and the internal state. The information gathered with this method can then be used directly by the user or by another program to better conduct further tests.

---

[2] `https://scoding.de/ropper/` - `https://github.com/sashs/Ropper`

[3] `https://www.frida.re` - `https://github.com/frida/frida`

3.2 The fuzzing world

"Fuzzing is the process of sending intentionally invalid data to a product in the hopes of triggering an error condition or fault." [29]

Fuzz Testing dates back to 1989 when it started as a project of the Operating Systems course at the University of Wisconsin-Madison. At his first stages it was nothing more than a simple Black Box technique to test the robustness of some UNIX utilities. The first two fuzzers *fuzz* and *ptyjig*, developed by two students, where incredibly successful in finding previously unknown vulnerabilities: they where able to crash around 30% of the considered utilities using random inputs [30]. It quickly became clear that this was a powerful and strong technique to test the robustness of a piece of software.

During the years fuzz testing evolved and developed gaining popularity and expanding to more and more fields (networks, files, wireless and more) until it became a proper domain of research and engineering. As for today it is quite vast and, using this kind of tools in the software testing phase would be the best practice. Nevertheless it is still uncommon to find fuzzing outside the security world, so it is hard to give a precise and unique definition for this technique. I will try to give an overview of the actual fuzzing world focusing mainly on the binary fuzzing and explaining how the previously mentioned techniques are applied to the fuzzing field.

Fuzzers can be at first divided into two main categories:

- **Mutation Based (or Dumb)**: it applies random mutations to the given input without any knowledge of the data that are being manipulated. Common mutations include bitflipping, shortening or expanding the input and similar. This method is a really simple and pure brute force approach and it can be applied to user provided inputs as well as to automatically generated ones to progressively create a valid input.

- **Generation Based (or Smart)**: it requires some knowledge of the data or protocol that is being fuzzed and this information is then used to generate proper inputs. Such information is either provided by the user who defines a specific set of rules or autonomously extracted from a sample by the fuzzer and can be used to generate valid CRC codes for mutated strings or keep the correct structure for particular files (jpg). Obviously this approach has some advantages but in some cases a mutation based approach can give better results as it can test programs even outside their functioning boundaries.

Predictably a fuzzer that combines the two methods is the one that can give the most comprehensive results.

Fuzzers can be associated with the previously mentioned testing methods. In this way fuzzers can be further differentiated based on the approach that they have to the problem and the information they require to test the binary. This divides the fuzzing world in three theoretical categories: **BlackBox Fuzzing**, **WhiteBox Fuzzing** and **GrayBox Fuzzing**. **BlackBox Fuzzing** is the pure Black Box approach, discussed earlier. It can be both mutation and generation based and it can usually only scratch the surface of a program as its behavior is based only on the output produced by the tested program. Therefore it is unlikely that this method will produce a test case able to deeply explore all the code and, for example, trigger nested if conditions. **WhiteBox Fuzzing** requires access to the source code and is a really sophisticated approach which is rarely used since it uses code analysis to generate test cases that will produce full code coverage. For this reason this approach is usually resource and time intensive. **GrayBox Fuzzing** is halfway between the two and it usually works with or without access to the source code. This method consists in injecting special code during the compilation phase or sandboxing the binary inside an emulator in order to extract information on the execution status and the code coverage achieved. With knowledge on the internal status it is also easier to test the program for particular conditions or specific cases.

However, fuzzing is not an exact science and in the real world choosing the right approach can be trivial and usually there is not a clear distinction between the different categories.

3.3 State of the art

There is much software available for many different fuzzing jobs. The choice is vast and goes from the simple hobby project to the much more complex and advanced commercial software. Choosing a good fuzzer is a crucial step for the success of the task: a fuzzer, which is too basic and relays solely on a pure brute force approach (like *ptyjig*) can require ages to find a simple bug on a modern program, whereas a "smarter" fuzzer, will take just a few minutes or hours.

This section is focused on binary fuzzers since the main objective of this research is to test binary tools. American Fuzzy Lop, which was used in this research as it offers many features and it is extensively documented, and Peach represent the actual state of the art in term of binary fuzzers.

In the following two sections there is an overview of this two tools.

3.3.1 American Fuzzy Lop

American Fuzzy Lop or **AFL** [4] represents the state of the art among binary fuzzers and it has a high rate of vulnerability discovery as stated in the "bug-o-rama trophy case" section of the website. Moreover it is widely used and in active development.

AFL has many different features that makes it a quite sophisticated software. It is mainly based on a Gray Box approach, which require access to the source code in order to perform what it calls "instrumentation". This process consists in adding some custom code to the program that is being compiled; this automatic operation, performed by the dedicated compilers *afl-gcc* and *afl-clang-fast*, adds small code snippets in critical positions without weighing down the program but allowing the fuzzer to obtain information on the execution flow at run time. It also uses a **Mutation Based** approach which relies on some statistical methods to optimize the generation making it efficient and fast to setup.

---

[4] `http://lcamtuf.coredump.cx/afl/`

In addition to instrumentation, **AFL** can fuzz Black Box binaries leveraging the QEMU emulator to obtain information about the execution state of the binary and adjust the generation method accordingly. Adopting a Black Box approach can be useful also when the source code is available since it can highlight vulnerabilities introduced in the compilation/optimization phase. However this can result in reducing the execution speed which considerably slows down the testing.

The information gathered by the fuzzer at execution time will then be used to meaningfully mutate the input. **AFL** has two different input generation strategies depending on the information supplied: if a file containing a valid input for the program is provided, it is used as a starting point to the next stages. Otherwise, if a blank file is supplied, **AFL** starts a "0-day" input generation. This last technique has been proven to be particularly successful, even if more time consuming than the other one. Interestingly it enabled **AFL** to generate valid URLs, JPG headers, XML syntax and more [31]. More specifically the fuzzer uses a genetic algorithm approach which keeps a queue of interesting inputs that will then be mutated. It will append a file to the queue if it triggers a previously unknown internal state. Three different deterministic mutation strategies are applied:

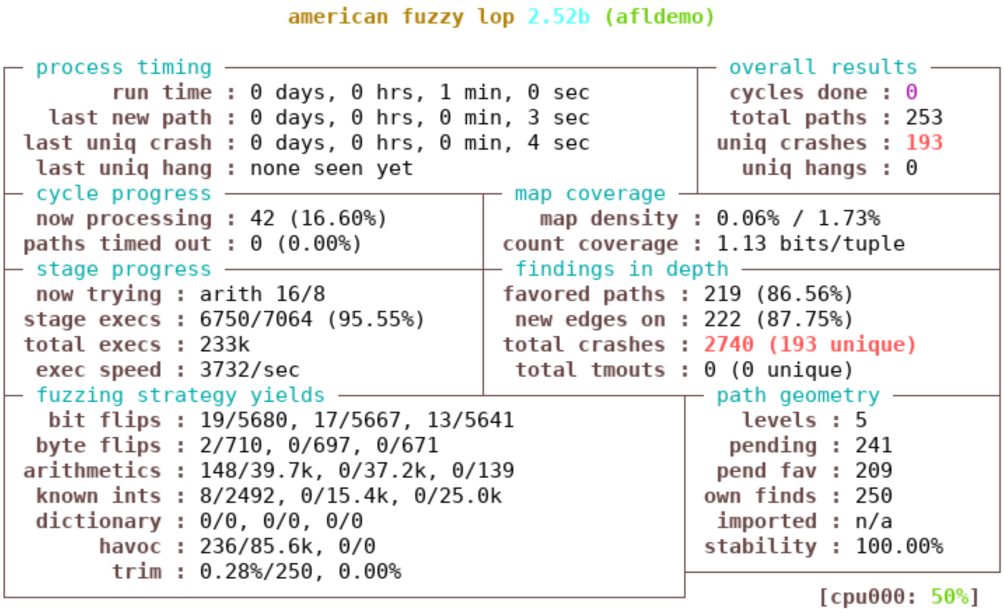
1. *Sequential bitflips with varying lengths and stepovers*: this is a slow but very effective way of mutating the input as it has been observed that, with different length of bitflipping, it is possible to generate around 130 new paths per million input.

2. *Sequential addition and subtraction of small integers*: this method consists in incrementing or decrementing existing integer values in the input file. The operation is performed in 3 different stages: firstly on 8-bit values, then on 16-bit values and lastly on 32-bit values all in both endianess. These last two methods are performed only if during the operation the most significant byte is changed; otherwise the value has already been tested in one of the previous cases.

3. *Sequential insertion of known interesting integers*: this method uses particular values that are known for their ability to trigger interesting cases, for example -1, MAX_INT, MAX_INT-1 and so on. The values are tested in both endianess and in 8,16,32 bits representation.

There are also non deterministic strategies applied in a never ending loop which include insertions, deletions, arithmetic, and splicing of different test cases [32].

Moreover **AFL** can be parallelized on many cores, in this way there will be a Master process that coordinates other Slave processes. All the orchestration and synchronization of the processes is handled automatically by **AFL**. The Master process will perform all the deterministic mutations while the Slaves will perform the random mutations, since the random mutations are faster to perform the Slaves will progress faster than the Master greatly speeding up the testing. In addition to this the fuzzer is also able to resume previously interrupted jobs without losing the progresses and the found crashing inputs.



```
                     american fuzzy lop 2.52b (afldemo)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 1 min, 0 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 3 sec        │  total paths : 253    │
│ last uniq crash : 0 days, 0 hrs, 0 min, 4 sec        │ uniq crashes : 193    │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 42 (16.60%)       │    map density : 0.06% / 1.73%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.13 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 16/8            │ favored paths : 219 (86.56%)          │
│ stage execs : 6750/7064 (95.55%)    │  new edges on : 222 (87.75%)          │
│ total execs : 233k                  │ total crashes : 2740 (193 unique)     │
│  exec speed : 3732/sec              │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 19/5680, 17/5667, 13/5641             │    levels : 5          │
│  byte flips : 2/710, 0/697, 0/671                   │   pending : 241        │
│ arithmetics : 148/39.7k, 0/37.2k, 0/139             │  pend fav : 209        │
│  known ints : 8/2492, 0/15.4k, 0/25.0k              │ own finds : 250        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 236/85.6k, 0/0                        │ stability : 100.00%    │
│        trim : 0.28%/250, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 50%]
```

Figure 3.1: AFL interface

AFL comes with a very informative user interface as it can be seen in Figure 3.1.

The most interesting and relevant parts are described here:

- *Overall Results*. The first field in this section shows the count of queue cycles done so far. This is the number of times the fuzzer consumed all the interesting test cases in the queue, fuzzing them, and started over from the very beginning.

- *Map Coverage*. This section gives insights about the results of the instrumentation and the internal state of the program. The first line shows how many branch tuples have already been hit, in proportion to how many are expected for this binary; this value is stored in a bitmap file. The number on the left describes the current input and the one on the right is the value for the entire input corpus.

  The second line shows the hit counts for the branch tuples of the binary. If every branch is taken for a fixed number of times for all the inputs, this value will be 1.00. As other hit counts are triggered for every branch the value will move towards 8.00 (every bit in the 8-bit map hit).

- *Stage Progress*. This section gives an image of what the fuzzer is actually doing. The first line indicates the strategy that is being applied to mutate the inputs. The following two lines show the number of executions for the current stage of the fuzzing and for the entire test procedure. The last line shows the current execution speed of the target to give an idea of the performance of the binary.

- *Path Geometry*. This section gives an overview of the inputs generation and discovery. The first line is the "path depth": user supplied inputs are of level 1, then the inputs derived from this are of level 2 and so on following this logic. The second line is the number of inputs not used yet for testing, the third line is the number of entries that are considered really promising by the fuzzer and will be tested in this cycle. Then the other two lines are the number of discovered new paths and of paths imported from other fuzzers during parallelized jobs. At the end, the last line measures the consistency of observed traces: if a program always behaves the same for the same input data, this value will be of 100%. A too low value indicates that **AFL** will have difficulty discerning between meaningful and "phantom" effects of tweaking the input file.

One of the best features of **AFL** is that it is an open source, meaning that every person can improve the code and modify it to meet different needs. For example one really active researcher in the fuzzing field, Dr. Marcel Böhme[5], created some interesting forks of **AFL** trying to achieve better code coverage [33, 34] and better path discovery [35].

afl-unicorn

afl-unicorn is a fork of **AFL** that was built to leverage the power of **AFL** to fuzz trivial binaries.

"For example, maybe you want to fuzz a embedded system that receives input via RF and isn't easily debugged. Maybe the code you're interested in is buried deep within a complex, slow program that you can't easily fuzz through any traditional tools."[36].

In particular it uses the Unicorn Engine [37], which is a CPU emulator, to run a program using the extracted context from an instance of the same application making it architecture independent and easy to selectively fuzz. As a matter of fact the emulator supports a wide range of CPUs: ARM, AArch64, M68K, Mips, Sparc and X86 thus permitting to easy leverage a different platform for the tests and to directly interact with the virtual processor at execution time. The Unicorn Engine allows to fuzz a program in the exact same way as before but, in addition to this, the user has now control over the memory, the CPU registers and the code of the program. This is useful in many different situations, for example when there is no access to the source code or when the sources are available but it can be more interesting to fuzz a precompiled version of the binary for many different reasons (to test the behavior of a compiler, to test a specific version for a specific architecture and so on). Moreover *afl-unicorn* allows the user to specify portions of code to be fuzzed, i.e. a single function. In addition to this the fuzzing takes place directly inside the emulated memory manipulating a user defined region. It is also possible to write custom "hooks" when certain functions are called or particular addresses are reached so as to skip specific portions of code or to trigger a defined behavior, such as a crash, if the program reaches a defined location in the code.

---

[5] `https://comp.nus.edu.sg/~mboehme/`

While it is true that this approach is very powerful it requires a template to be hand written for the Unicorn Engine. This requires specific information like the heap addresses, the register values and the content as well as the addresses of all the memory regions. The extraction of such information is a trivial process since a binary must be running inside a debugger and, usually, the relevant portion of code must be identified. Only then the template for the emulator can be written. This process is time consuming and requires very specific knowledge of the hardware. As a matter of fact the context of the process must be dumped from the memory and, at the same time, the binary must be disassembled to find the boundaries inside which the emulation must take place.

For those tasks some tools like *unicorn_dumper.py* and a basic template are provided with the tool; other programs like *unicorn_template_generator.py* and *extract_from_memory.py* have been written specifically for this task and will be illustrated in Chapter 4.

3.3.2 Peach

Peach is another very popular fuzzer and is the commercial counterpart of **AFL**. However, the two programs are different since **Peach** is capable of fuzzing different targets such as network protocols, device drivers, file consumers, embedded devices, external devices natively, while in **AFL** all these capabilities are added by specific forks. Another difference comes in the generation of inputs. As a matter of fact **Peach** is a **BlackBox** and **Generation Based** fuzzer that requires no access to the source code but, instead, a *"Peach Pit"* must be supplied. A *Pit* is a template, defined by the user, that the fuzzer uses to generate the input data, monitor the target and display the status and results. It contains the following information:

- A description of the data layout to test.
- The agent, monitors and I/O adapter to use in the fuzzing session.
- Setup parameters, such as Port or Log Folder, to use in the fuzzing session.

As an added factor *Pits* are open source and shared inside the **Peach** community which makes finding the perfect *Peach Pit* most likely. Moreover from the Whitepaper it is possible to understand that there are *"over 50 mutation algorithms"* but there is no explanation on what these algorithms are doing or how [38].

Peach was open source but from version 3 the policy changed as it is stated on the website: *"Peach is not open source software, it's FREE software. Peach is licensed under the MIT License which places no limitations on it's use. This software license was selected to guarantee that companies and individuals do not have to worry about license tainting issues."* [39]. This license is ambiguous because on the company website a Community Edition of **Peach 3** is available and it is stated to be open source (source code is also available). This is probably due to the fact that the company sells fuzzing services using **Peach** so the commercial version might have more features. However an in development fork of **Peach 2.7** is maintained by Mozilla.[6]

---

[6] `https://github.com/MozillaSecurity/peach`

Chapter 4

Analysis

Accessing real avionics hardware and software was out of my reach for this research, therefore the analysis was carried out on open source implementations of **ADS-B**, **FIS-B** and **NexRad** protocols as found in some popular source tree, binary package and device firmware releases such as Stratux, FlightRadar24 and FlightAware. Experimenting on those software has also secondary relevant implications as it allows to test at the same time avionics protocols as well as IoT devices. As a matter of fact sensors using such programs are commonly used and often connected to company networks or user personal devices (i.e smartphone or tablet). Even though avionics software must comply with strict regulations and security standards as described in DO-178 (ED-12 in Europe) and DO-278 (ED-109), it is most likely that using one of the previously mentioned protocols as attack vector will have the same effect on all the implementations. While it is true that the systems in which such protocols are used are a crucial component of an aircraft, as they are connected to the main control systems and can take autonomous decisions, in the same way similar problems arise in ground stations and in the network of IoT sensors as those can be used as an entry point to an organization network.

Stratux project is particularly interesting as it uses inexpensive hardware (a RaspberryPi and a RTL-SDR dongle) to provide "ADS -B In" services. Moreover Stratux software is compatible with every major Electronic Flight Bags (EFB) and, as far as it can be understood from its community, it is largely used by pilots making it a proper avionics component at least in the General Aviation sector.

Fuzzing was chosen as test method since, based on the knowledge acquired before the research, no previous attempt of applying this test method to avionics protocols had been carried out. This research gains even more importance if we examine the latest news about the hacking of a Boeing 757 and the Cyber Grand Challenge (CGC) 2016 [40]. In particular, the CGC contained a specific challenge (`FSK_Messaging_Service`) [41] tailored to identify techniques and systems able to discover vulnerabilities in the Radio Frequency (RF) software using processed data after the RF front-ends. Aircraft and avionic radio-communication interfaces are a perfect example of this type of design. *afl-unicron* has been proved particularly effective on solving this specific challenge of the CGC, therefore it was used as an additional test method.

Even though there is no official linking between the CGC and the Boeing hack the circumstances indicate that there is a search and a need for knowledge and tools that can exploit (and therefore eventually protect) RF facing software, embedded devices in general and avionics and **NextGen** devices in particular. Researches in this field are moving fast and some tools, such as beSTORM [42] and TumbleRF [43] that are now public, allow for direct RF fuzzing.

4.1 Hardware Setup

The hardware used during the research was the following:

- *DVB-T (RTL-SDR) dongle:* a simple and cheap TV receiver equipped with the *RTL2832U* or compatible chip that can be easily tuned on a very wide range of frequencies, not only on the TVs, using *librtlsdr*.
- *RaspberryPi (RPi) 3 Model B:* was used to acquire the data using the RTL-SDR dongle, and also to run some tests on the precompiled binaries. The RaspberryPi was running the latest version of Raspbian, or the particular flavor/version of Linux or Raspbian bundled as part of Stratux and FlighTRadar24 SD-card firmware image files.
- *Standard laptop:* was used for dry-run testing, for initial code testing and for initial fuzzing experiments monitoring. The laptop has the following specifications: Intel Core i7 5500 (4 cores), 12 GB of RAM. Since the fuzzing is time and resource consuming, when the running experiments were considered promising or really important, they were moved to run on a multi-core server.
- *Multi-core server:* with the following specifications: Intel Xeon CPU E7-8837 @ 2.67GHz (64 cores, 32 cores used for a single afl-fuzz experiment), 1 TB of RAM, running Centos7.4 Linux.

4.2 Software Setup

The focus was on two of the most widespread and common implementations of the protocols: *dump1090* and *dump978*. The first one decodes **OldGen** and **1090ES** messages while the second one decodes **UAT** messages.

Those software have two main components:

1. A *demodulator* that converts the raw signal into a proper binary string.
2. A *decoder* that extracts the information from a packet provided by the demodulator.

dump1090 handles directly the communication with the RTL-SDR dongle using *librtlsdr*. This library has been developed to work with the *RTL2832U* Radio Frequency (RF) chip providing access to the raw signal. It is possible to define other sources for the raw data such as a file or the standard input with the option `"--ifile"`. *dump1090* is a big monolithic software which has undergone major rework and edits from different authors resulting in many different forks. For these reasons testing each component individually is a trivial task which can be accomplished only by using *afl-unicorn*.

dump978 does not acquire directly the raw data which, instead, has to be provided on the standard input. *librtlsdr* comes with a convenient command line utility, called *rtl-sdr*, which can be used to modify all the parameters of the dongle and in particular to tune it on a specific frequency and dump the raw signal. The *dump978* software has a very modular design; in its basic version it is composed by at least 3 different programs: *rtl-sdr* used to communicate with the dongle and acquire the raw data. *dump978* which is the demodulator and takes raw data as input giving encoded messages as output. *uat2text* which is the decoder and converts the output of the demodulator into a readable form. In addition to these *extract_nexrad* decodes **NexRad** packets from the demodulated output and *plot_nexrad.py* creates png images from those data.

While *dump978* is a compact and organized software with almost no forks, *dump1090* has many different versions so testing all of them is impossible. Therefore, three of the most widespread versions were selected for fuzzing. The authors are *Salvatore Sanfilippo (Antirez), MalcommRobb* and the *Stratux community*, for each author the very first and the latest release were considered. The main **AFL** experiments were run on the latest version of the tool from the Stratux repository as it seemed the most updated and in active development. Later **AFL** tests were also carried out on the first stable release from *Antirez* (commit id 8236319) since he is the original author of this tool and this commit keeps the basic features without the clogging created by the later introduction of accessory tools such as a server for real time view of the traffic. Moreover the testing wanted also to consider precompiled binaries which are shipped inside the previously mentioned Linux images and that are integrated in specific receivers such as the proprietary ones from FlightRadar24. Therefore *dump1090* was extracted from the latest available release (1.4r4), from the first working release (0.8r2) and from the very first release (0.1) of Stratux, as well as from the latest release (1.0.18-9) of the FlighRadar24 Linux image (which is a free but not open source). The precompiled version of *dump1090* analyzed with *afl-unicorn* was the one contained in the first Stratux image and it seems to come directly from the *Antirez* repository and from the commit id 8236319.

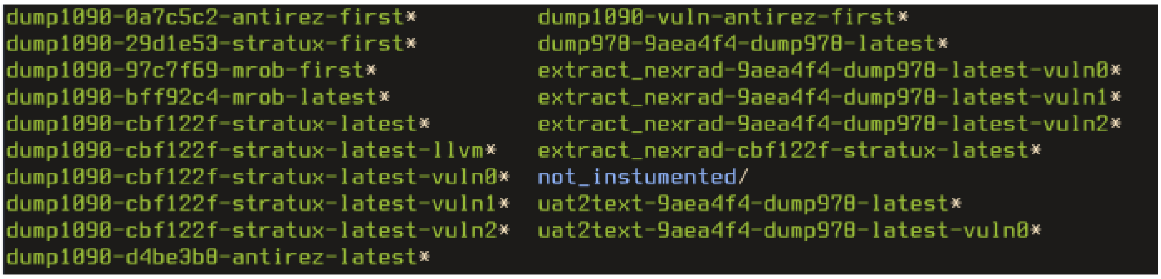
All the collected sources were then compiled using *afl-gcc* and conveniently named using the following rule:

binary_name–short_commit_id–repository_name–version–vulnerability

Where the presence of the last value indicates that the binary contains an artificially introduced vulnerability for testing purposes. The final lineup of the binaries for testing can be seen in Figure 4.1.

4.2.1 Datasets

Every considered software requires its specific format of data, moreover, the input given to **AFL** must be carefully chosen as using a too narrow input (i.e. that triggers only a specific function) can lead to a waste of time before the fuzzer will actually be able to syntetize a meaningful input and, as a consequence, the



```
dump1090-0a7c5c2-antirez-first*         dump1090-vuln-antirez-first*
dump1090-29d1e53-stratux-first*         dump978-9aea4f4-dump978-latest*
dump1090-97c7f69-mrob-first*            extract_nexrad-9aea4f4-dump978-latest-vuln0*
dump1090-bff92c4-mrob-latest*           extract_nexrad-9aea4f4-dump978-latest-vuln1*
dump1090-cbf122f-stratux-latest*        extract_nexrad-9aea4f4-dump978-latest-vuln2*
dump1090-cbf122f-stratux-latest-llvm*   extract_nexrad-cbf122f-stratux-latest*
dump1090-cbf122f-stratux-latest-vuln0*  not_instumented/
dump1090-cbf122f-stratux-latest-vuln1*  uat2text-9aea4f4-dump978-latest*
dump1090-cbf122f-stratux-latest-vuln2*  uat2text-9aea4f4-dump978-latest-vuln0*
dump1090-d4be3b0-antirez-latest*
```

Figure 4.1: Binaries to be tested

vulnerable point might never be hit.

Some suggestions from the README of **AFL** and the ThalesIgnite afl-training[1] are:

- Find some real inputs that exercise as much of the target as possible.
- Keep the input file small (Under 1kb is ideal).
- Use multiple test cases only if they are functionally different from each other.

Complying with those rules, multiple test cases were created. A sample for *dump1090* was captured using the RaspberryPi placed in the university laboratory and resulted in two files: *1090_small.bin* which contains a valid message, some noise and some invalid messages and *1090_smaller.bin* which only contains one valid message. In addition to this, *modes1.bin* is a sample file contained in the *dump1090* repository. However, despite containing some sample messages, it was never used during the research. Capturing some 978MHz data is a hard task in Europe since, as already said, this frequency and the **UAT** protocol is deployed mainly in North America and China. Moreover no raw samples come with the *dump978* bundle but only some demodulated messages are provided. It was therefore necessary to find another way of acquiring such data. One option was to ask on Reddit which successfully resulted in more than half a GB (*978_big.bin*) of raw capture containing almost any kind of **UAT** messages[2]. Such a big file cannot be used as an input for **AFL** either because the fuzzer will complain about the size and because using a such large amount of data will have a significant impact on the fuzzing process and on the execution speed of the binary. Therefore a smaller and more convenient file (*978_random.bin*) was created, starting from the previous one and taking some random data with the objective of simply reducing the size but still keep at least one message. Creating files such as *1090_small.bin* and *978_random.bin* was a trivial process since there is no easy way to access and manage the raw data contained in the bigger files particularly regarding the content and his meaning, however the creation of a tool to interact and manage raw samples is left as future work.

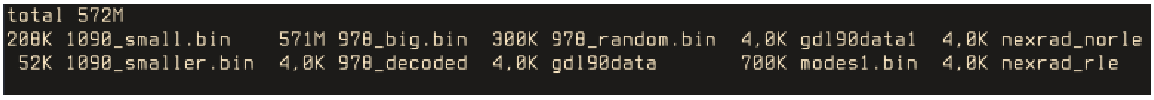
To test *uat2text* and *extract_nexrad* tools, some files were created containing specific demodulated messages from *978_big.bin*. In addition to this, two files (*gdl90data* and *gdl90data1*) containing **FIS-B/NexRad** samples from the documentation of the Garmin first certified **ADS-B** datalink transceiver (GDL90 [44]) were used in the tests. In particular, for **FIS-B** services, RTCA provides a document (DO-358: *"Minimum Operational Performance Standards (MOPS) for Flight Information Services Broadcast (FIS-B) with Universal Access Transceiver (UAT)"*) and the respective supplement containing many different test files [45] that, after a few modifications, can be used with **AFL**.

---

[1] `https://github.com/ThalesIgnite/afl-training`

[2] `https://www.reddit.com/r/stratux/comments/7zdoaa/raw_dump_of_978_data`

The final content of the dataset directory can be seen in Figure 4.2.



```
total 572M
200K 1090_small.bin    571M 978_big.bin  300K 978_random.bin  4,0K gdl90data1  4,0K nexrad_norle
 52K 1090_smaller.bin  4,0K 978_decoded  4,0K gdl90data       700K modes1.bin  4,0K nexrad_rle
```

Figure 4.2: Dataset created

4.2.2 Tools

As already mentioned **AFL** and *afl-unicorn* were used for testing. However, during the project a dedicated set of tools and scripts were developed to partially automate and simplify data processing and fuzzing tasks. The tools are divided into two categories: **Data Tools** used to manage, modify and create the datasets and **Fuzzing Tools** used to facilitate and speed up the fuzzing process. The details and descriptions of the tools are as follows:

Data Tools

- **converter.sh** and **runner.sh** are scripts meant to interact with the data provided by the DO-358 supplement zip file. This archive contains many different files which are designed to be used with a dedicated test tool for real avionics hardware and software, for this reason there are 18 subfolders (called groups) and, each one of them, contains: *TestGroupXX Procedures.doc*, *TestGroupXX Stimulus.csv* and a *bin* folder with the actual data files. Every *bin* folder has many different files, each representing a unique type of information which has already been demodulated and is stored in a binary format. Since the program being tested only accepts data in a uplink (or downlink) format **converter.sh** was written to convert one single file or an entire directory in the appropriate format. Every resulting file contains just one encoded type of data and, feeding a large amount of those files into a program will be a long and tedious task. For this reason the **runner.sh** script will feed each file contained inside a specified directory through the user defined program either interactively, namely stopping after each file and asking if the user wants to continue, or simply running all files at once.

- **message_generator.py** is a simple script that, after receiving the first part of a demodulated Mode-S message, can calculate a correct CRC. This can be done virtually for every random string with a correct length but the script has not been widely used.

  Its only application was for test purposes since it was considered interesting to see what could happen when the program being tested was fed with particular messages such as those composed by all 1 or all 0 and having a valid CRC. This script could be part of a future bigger program that will test the protocols generating messages that always have a valid CRC.

Fuzzing Tools:

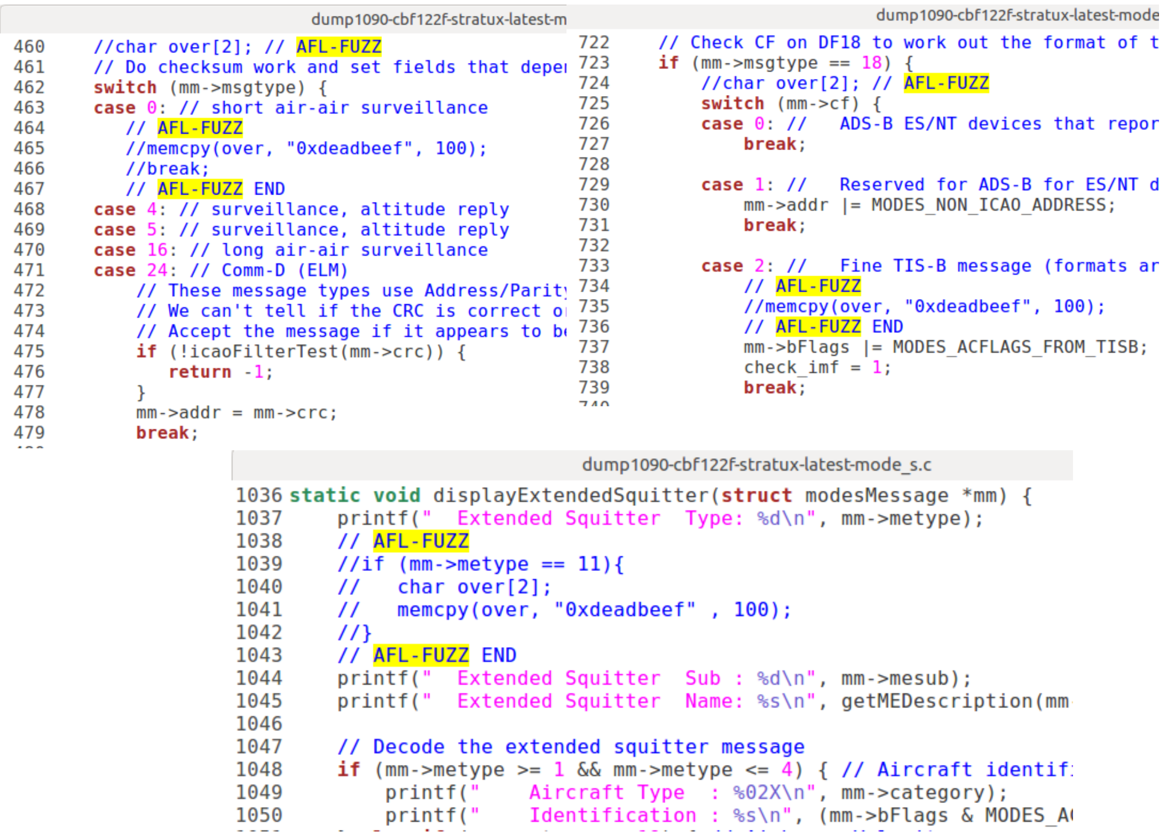
- **start.sh** is a script that will automatically spawn or resume a user defined number of fuzzers. In particular since **AFL** can be parallelized on many cores the script handles the creation of the input and output folders as well as a convenient naming of each fuzzer instance. It will take as input the number of cores to be used and all the other parameters required by the fuzzer, perform a check of the directories and then start the chosen number of fuzzers: 1 Master and $n - 1$ Slaves.
- **unicorn_template_generator.py** will create the template for the Unicorn Engine and populate it with all the required addresses such as the ones inside which the emulation should take place or the one of the functions to be skipped. The script is designed to be sourced inside a running session of GDB while the program is on a breakpoint inside a function; it will not work if called inside the main since it is specifically designed to search for the end of the function.
- **extract_from_memory.py** will dump the specified memory region from a running program in GDB to be used as input for `afl-unicorn`. This will generate a file that can be used with the previously generated template, this will perform the various mutations and then load it in the proper memory region.
- **afl utility scripts**:
  - **killem_afl.sh** → stop a running instance of afl.
  - **wazzup_afl.sh** → get information and statistics on the running instances of the specified fuzzer.
  - **afl_noroot.sh** → run afl on a system where the user does not have root privileges.

4.3 Proceedings

The analysis was first done on the **OldGen/1090ES** then on **UAT**. As already said the first step was to confirm that the fuzzer was working correctly and that it was able to crash the analyzed binaries. For this task some synthetic vulnerabilities were introduced in almost all the binaries starting from *dump1090*. The vulnerability had to be kept simple, powerful and yet easy to trigger, for this reason a simple buffer overflow was chosen. It is important to note that the synthetic vulnerability should not be triggered by the initial test case provided to **AFL** otherwise it will stop the fuzzing considering the binary not reliable.

The integration in only one binary of all the functions of *dump1090* had an impact on the performances of the fuzzer and made correctly test the demodulator and the decoder parts harder as the data had always to be demodulated first. This lead to some inconveniences: first of all it was harder to feed arbitrary data directly to the decoder as no modulator was available. Then, even just to test the decoder, raw data had to be supplied resulting in a useless demodulation step and in a waste of time as many mutations will have no effect and will be processed as random noise. Although it is possible to feed encoded messages to *dump1090* this is done thought a socket connection which requires a network fuzzer and is out of the scope of this research.

Three vulnerabilities were introduced in different parts of the latest *dump1090* release from the Stratux repository as it can be seen in Figure 4.3. The overflow was inside the decodeModesMessage function, the second one was inside the decodeExtendedSquitter function while the last one was in the printing function, the demodulator was not touched as, without specific knowledge, it is not easy to understand which part would be the best.



```
    //char over[2]; // AFL-FUZZ
    // Do checksum work and set fields that depe
    switch (mm->msgtype) {
    case 0: // short air-air surveillance
       // AFL-FUZZ
       //memcpy(over, "0xdeadbeef", 100);
       //break;
       // AFL-FUZZ END
    case 4: // surveillance, altitude reply
    case 5: // surveillance, altitude reply
    case 16: // long air-air surveillance
    case 24: // Comm-D (ELM)
        // These message types use Address/Parit
        // We can't tell if the CRC is correct o
        // Accept the message if it appears to b
        if (!icaoFilterTest(mm->crc)) {
           return -1;
        }
        mm->addr = mm->crc;
        break;
```

```
    // Check CF on DF18 to work out the format of t
    if (mm->msgtype == 18) {
        //char over[2]; // AFL-FUZZ
        switch (mm->cf) {
        case 0: //   ADS-B ES/NT devices that repor
            break;

        case 1: //   Reserved for ADS-B for ES/NT d
            mm->addr |= MODES_NON_ICAO_ADDRESS;
            break;

        case 2: //   Fine TIS-B message (formats ar
            // AFL-FUZZ
            //memcpy(over, "0xdeadbeef", 100);
            // AFL-FUZZ END
            mm->bFlags |= MODES_ACFLAGS_FROM_TISB;
            check_imf = 1;
            break;
```

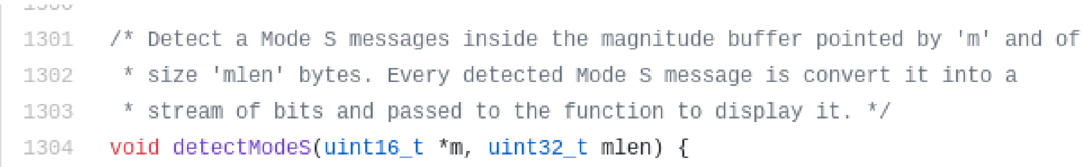
```
static void displayExtendedSquitter(struct modesMessage *mm) {
    printf("  Extended Squitter  Type: %d\n", mm->metype);
    // AFL-FUZZ
    //if (mm->metype == 11){
    //   char over[2];
    //   memcpy(over, "0xdeadbeef" , 100);
    //}
    // AFL-FUZZ END
    printf("  Extended Squitter  Sub : %d\n", mm->mesub);
    printf("  Extended Squitter  Name: %s\n", getMEDescription(mm

    // Decode the extended squitter message
    if (mm->metype >= 1 && mm->metype <= 4) { // Aircraft identif
        printf("    Aircraft Type  : %02X\n", mm->category);
        printf("    Identification : %s\n", (mm->bFlags & MODES_A
```

Figure 4.3: dump1090 vulnerabilities

1090_small.bin was then used to test the vulnerable binaries using 32 instances of **AFL** (1 Master 31 Slaves) for each of the 3 binaries, one per vulnerability. Interestingly the fuzzer was able to crash only the first binary, there could be multiple reasons for this as it will be discussed in Chapter 5.

Since even after a large amount of cumulative execution days the fuzzer was not able to find any vulnerability in the latest version of *dump1090* from the Stratux repository (short commit id *cbf122f*) using *1090_small.bin*, even with non instrumented binaries and the QEMU mode of **AFL**, the focus was shifted on another approach: *afl-unicorn*. Therefore it was necessary to understand how templates for this tool are generated, luckily a basic blank template comes inside the *afl-unicorn* repository along with a tool to dump the memory content and addresses. Using GDB with the GEF[3] plugin the template was at the beginning created by had and set to test the *detectModeS* function (in Figure 4.4) which is responsible of demodulating Mode S messages. Getting to a basic working version of the template is not an easy task considering that many of the libraries used by *dump1090* relays on System Calls which, since no underlying operating system is present inside the emulator, will cause unpredictable crashes. Therefore it was necessary to find the addresses of problematic functions, such as *printf, putchar, puts* and so on, and to configure hooks in order to skip such functions.



```
/* Detect a Mode S messages inside the magnitude buffer pointed by 'm' and of
 * size 'mlen' bytes. Every detected Mode S message is convert it into a
 * stream of bits and passed to the function to display it. */
void detectModeS(uint16_t *m, uint32_t mlen) {
```

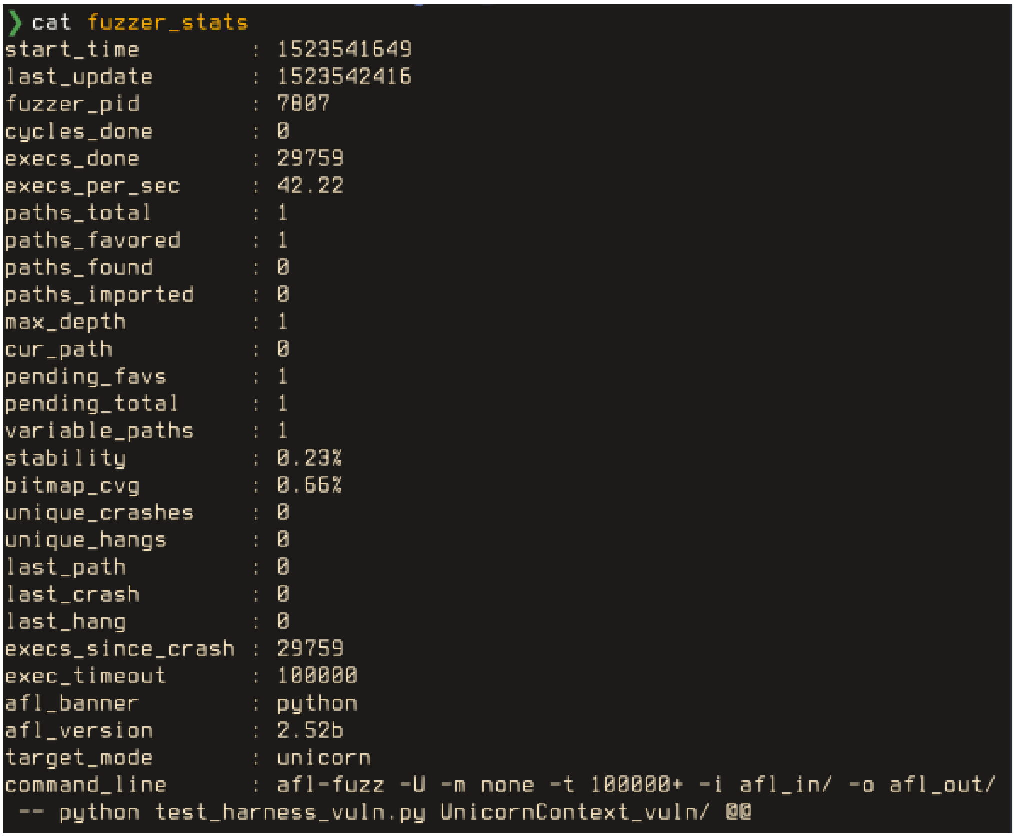
Figure 4.4: detectModeS function

This analysis was carried out on the ARM precompiled binary contained in the first release of the Stratux Linux image (stratux-v.0.1-08072015). To confirm that *afl-unicorn* was working as expected a vulnerable version of *dump1090*, with an overflow inside the *detectModeS* function, was compiled on the RaspberryPi, the context dumped with the proper scripts and the template created with *unicorn_template_generator.py*. The fuzzer was then run on the standard laptop using the generated template for an entire day, unluckily no crashes where produced both for the vulnerable and for the stock versions. This might be due to the low hardware resources of the regular laptop and to the low testing time since *afl-unicorn* has a much slower testing speed due to the many steps required by the Unicorn Engine to start the emulation and to the low execution speed of a program inside the emulator. For unknown reasons it was not possible to setup a working instance of the Unicorn Engine and therefore of *afl-unicorn* on the multi-core server since the execution of the emulation engine on an x86 and an x86_64 machine gives completely different results than the runs on the server. The results of the run on the regular laptop can be seen in Figure 4.5.

Moving on from *dump1090* the research focused on *dump978*, an experimental run of **AFL** was done using *978_random.bin* and the latest version of *dump978* (short commit id *9aea4f4*). However, the 978 demodulator was not the main target of the research which was more focused on *uat2text* and *extract_nexrad*, this last software is of particular interest as it is designed to handle packets containing images which can be encoded, among others, with the Huffman algorithm known through the years to be particularly difficult to implement in a safe way [46, 47]. To test those software the same approach as before was used, meaning that **AFL** was at first run on an artificially vulnerable version and then on the stock release. This resulted in three binaries, two from *extract_nexrad* and one from *uat2text*,

---

[3] GDB Enhanced Features - `https://github.com/hugsy/gef`



```
❯ cat fuzzer_stats
start_time        : 1523541649
last_update       : 1523542416
fuzzer_pid        : 7807
cycles_done       : 0
execs_done        : 29759
execs_per_sec     : 42.22
paths_total       : 1
paths_favored     : 1
paths_found       : 0
paths_imported    : 0
max_depth         : 1
cur_path          : 0
pending_favs      : 1
pending_total     : 1
variable_paths    : 1
stability         : 0.23%
bitmap_cvg        : 0.66%
unique_crashes    : 0
unique_hangs      : 0
last_path         : 0
last_crash        : 0
last_hang         : 0
execs_since_crash : 29759
exec_timeout      : 100000
afl_banner        : python
afl_version       : 2.52b
target_mode       : unicorn
command_line      : afl-fuzz -U -m none -t 100000+ -i afl_in/ -o afl_out/
 -- python test_harness_vuln.py UnicornContext_vuln/ @@
```

Figure 4.5: afl-unicorn statistics on the regular Laptop

each containing a vulnerability in a different part of the code, as it can be seen in Figure 4.6.

The first vulnerability in *extract_nexrad* is inserted to cause an overflow if the data is RLE encoded and the scale factor is greater than 0, this value can go from 0 to 2. The second vulnerability is triggered if the data is not RLE encoded and the scale factor is greater than 0. Although it is theoretically possible to send Huffman Encoded **NexRad** images no reference to Huffman Tables or Huffman Decoding was spotted in the source code.

In *uat2text* the vulnerability was put in the *uat_decode_uplink_mdb* function, this is where the decoding of **UAT** frames takes place, the vulnerability is triggered if the decode latitude is greater than 90 degrees.

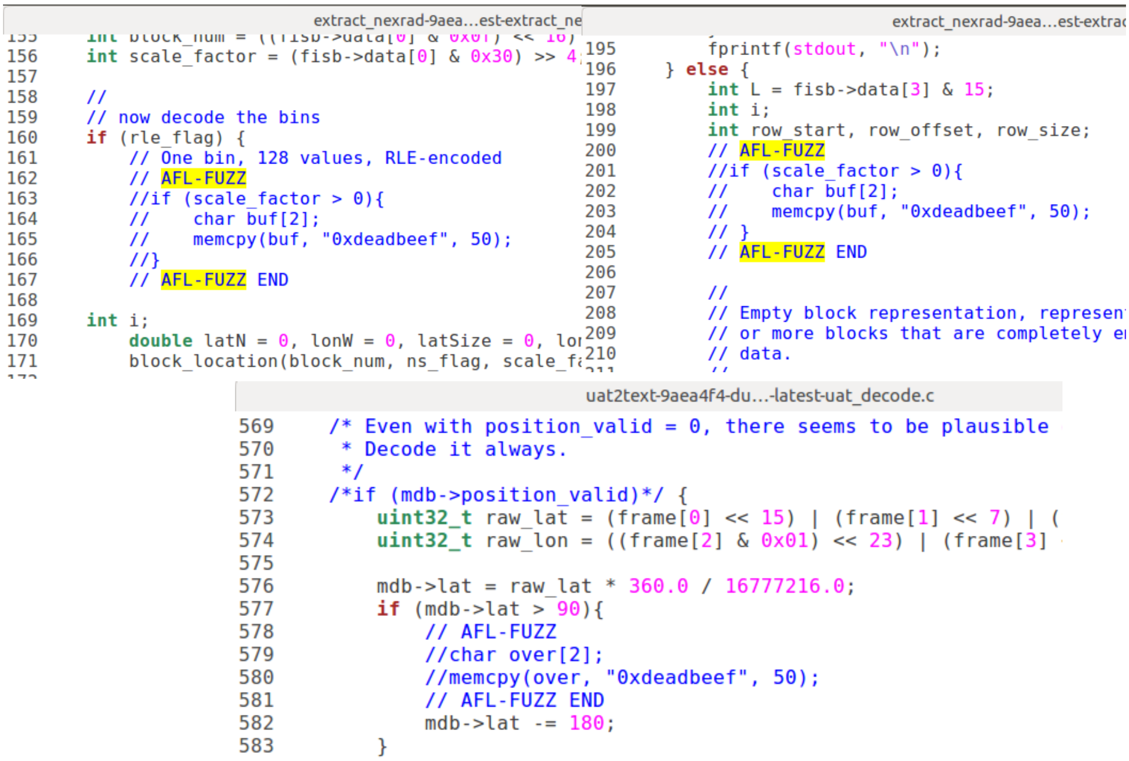
The fuzzer was successfully run on both vulnerable versions of *extract_nexrad* using *nexrad_rle* and *nexrad_norle* files, **AFL** was able to crash the two versions in a very short amount of time. After this the fuzzer was run on the stock version using the same two files however, even after many days of cumulative execution, no crash was triggered.

uat2text was then tested using *978_decoded* as input, even in this case **AFL** was able to crash the vulnerable binary almost instantly. However, when run on the stock binary using the same file as input, many days of cumulative execution produced no appreciable crash.

A last test, using a completely different binary found in *afl-demo*[4] repository,

---

[4] `https://gitlab.com/wolframroesler/afl-demo`



extract_nexrad-9aea…est-extract_ne

```
    int block_num = ((fisb->data[0] & 0x0f) << 16)
    int scale_factor = (fisb->data[0] & 0x30) >> 4

    //
    // now decode the bins
    if (rle_flag) {
        // One bin, 128 values, RLE-encoded
        // AFL-FUZZ
        //if (scale_factor > 0){
        //    char buf[2];
        //    memcpy(buf, "0xdeadbeef", 50);
        //}
        // AFL-FUZZ END

    int i;
        double latN = 0, lonW = 0, latSize = 0, lo
        block_location(block_num, ns_flag, scale_f
```

extract_nexrad-9aea…est-extrac

```
            fprintf(stdout, "\n");
    } else {
        int L = fisb->data[3] & 15;
        int i;
        int row_start, row_offset, row_size;
        // AFL-FUZZ
        //if (scale_factor > 0){
        //    char buf[2];
        //    memcpy(buf, "0xdeadbeef", 50);
        // }
        // AFL-FUZZ END

        //
        // Empty block representation, represen
        // or more blocks that are completely e
        // data.
```

uat2text-9aea4f4-du…-latest-uat_decode.c

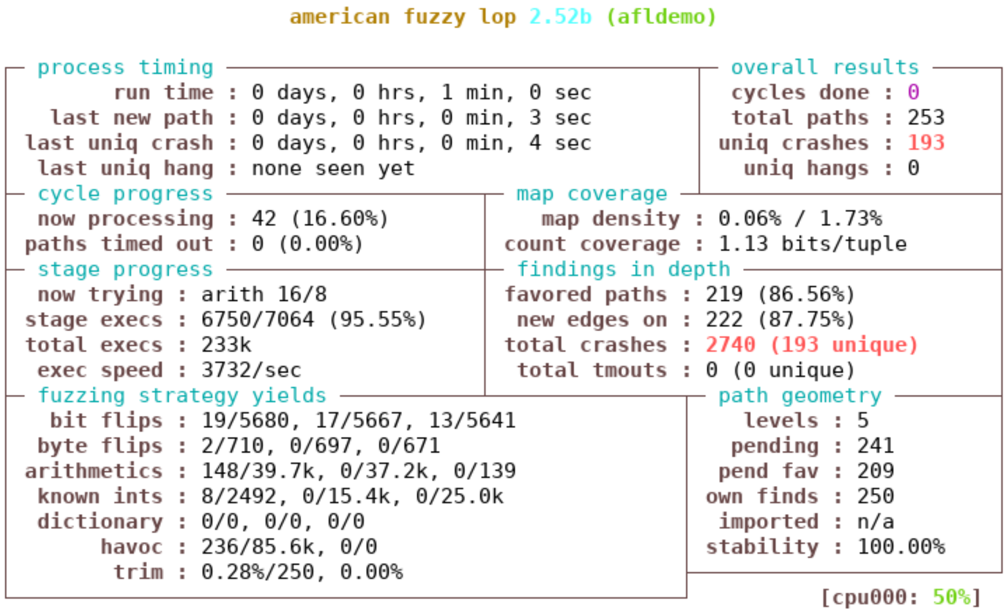
```
    /* Even with position_valid = 0, there seems to be plausible
     * Decode it always.
     */
    /*if (mdb->position_valid)*/ {
        uint32_t raw_lat = (frame[0] << 15) | (frame[1] << 7) | (
        uint32_t raw_lon = ((frame[2] & 0x01) << 23) | (frame[3]

        mdb->lat = raw_lat * 360.0 / 16777216.0;
        if (mdb->lat > 90){
            // AFL-FUZZ
            //char over[2];
            //memcpy(over, "0xdeadbeef", 50);
            // AFL-FUZZ END
            mdb->lat -= 180;
        }
```

Figure 4.6: extract_nexrad and uat2text vulnerabilities

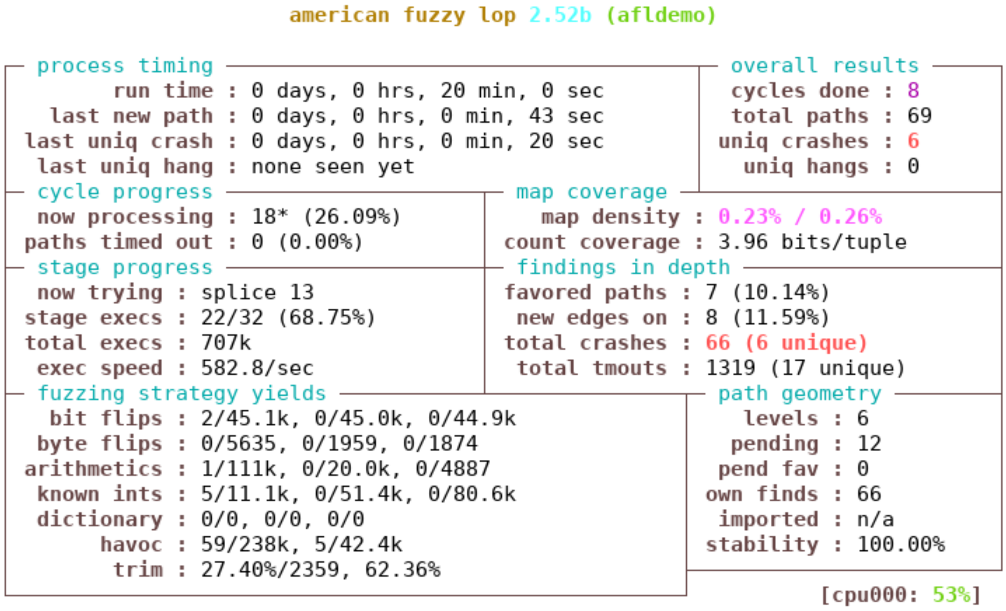
was carried out on all the flavors of **AFL** used during the research. This binary is built specifically to test the abilities of the fuzzer and is a simple and buggy URI decoder. It was run on just a single instance of **AFL**, **AFL** QEMU mode and *afl-unicorn*. The results can be seen in Figure 4.7.

The top image shows the results after a 1 minute run of **AFL**, which in such a short amount of time is able to discover 193 unique crashing inputs. The second image shows the results of a 20 minutes run of **AFL** on a BlackBox binary using QEMU mode. As it can be clearly seen this approach is way slower than the previous one as a matter of fact it was able to find only 6 unique crashes. The last three images are about *afl-unicorn*, as it can be seen from the last one it is not able to find any crashes and, most importantly, no new paths are triggered. This might be due to many different factors that still need further and deeper investigation. Although the two previous images show that the real registers, after the execution of the tested function (*uridecode*), from inside GDB and the ones resulting from the execution of Unicorn template have the same value except for some of them that are related to the address of where the input is loaded.



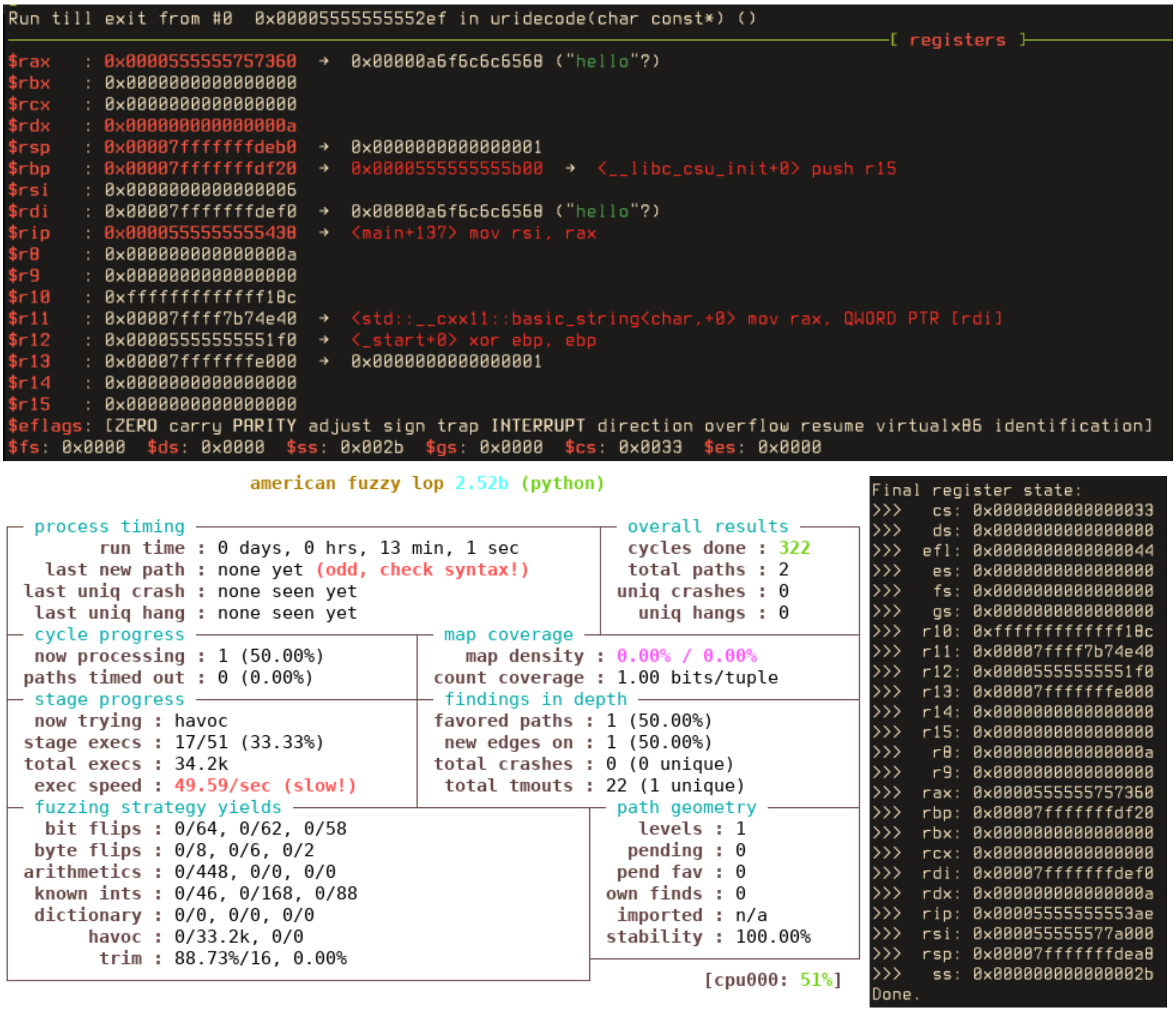
```
                     american fuzzy lop 2.52b (afldemo)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 1 min, 0 sec        │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 3 sec        │  total paths : 253   │
│ last uniq crash : 0 days, 0 hrs, 0 min, 4 sec        │ uniq crashes : 193   │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 42 (16.60%)       │    map density : 0.06% / 1.73%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.13 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : arith 16/8            │ favored paths : 219 (86.56%)          │
│ stage execs : 6750/7064 (95.55%)    │  new edges on : 222 (87.75%)          │
│ total execs : 233k                  │ total crashes : 2740 (193 unique)     │
│  exec speed : 3732/sec              │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ───────┤
│   bit flips : 19/5680, 17/5667, 13/5641             │    levels : 5         │
│  byte flips : 2/710, 0/697, 0/671                   │   pending : 241       │
│ arithmetics : 148/39.7k, 0/37.2k, 0/139             │  pend fav : 209       │
│  known ints : 8/2492, 0/15.4k, 0/25.0k              │ own finds : 250       │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a       │
│       havoc : 236/85.6k, 0/0                        │ stability : 100.00%   │
│        trim : 0.28%/250, 0.00%                      ├───────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 50%]
```



```
                     american fuzzy lop 2.52b (afldemo)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 20 min, 0 sec       │  cycles done : 8     │
│   last new path : 0 days, 0 hrs, 0 min, 43 sec       │  total paths : 69    │
│ last uniq crash : 0 days, 0 hrs, 0 min, 20 sec       │ uniq crashes : 6     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 18* (26.09%)      │    map density : 0.23% / 0.26%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 3.96 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 13             │ favored paths : 7 (10.14%)            │
│ stage execs : 22/32 (68.75%)        │  new edges on : 8 (11.59%)            │
│ total execs : 707k                  │ total crashes : 66 (6 unique)         │
│  exec speed : 582.8/sec             │  total tmouts : 1319 (17 unique)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ───────┤
│   bit flips : 2/45.1k, 0/45.0k, 0/44.9k             │    levels : 6         │
│  byte flips : 0/5635, 0/1959, 0/1874                │   pending : 12        │
│ arithmetics : 1/111k, 0/20.0k, 0/4887               │  pend fav : 0         │
│  known ints : 5/11.1k, 0/51.4k, 0/80.6k             │ own finds : 66        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a       │
│       havoc : 59/238k, 5/42.4k                      │ stability : 100.00%   │
│        trim : 27.40%/2359, 62.36%                   ├───────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 53%]
```



```
Run till exit from #0  0x00005555555552ef in uridecode(char const*) ()
──────────────────────────────────────────────────────────────────[ registers ]────
$rax   : 0x0000555555757360  →  0x00000a6f6c6c6568 ("hello"?)
$rbx   : 0x0000000000000000
$rcx   : 0x0000000000000000
$rdx   : 0x000000000000000a
$rsp   : 0x00007fffffffdeb0  →  0x0000000000000001
$rbp   : 0x00007fffffffdf20  →  0x0000555555555b00  →  <__libc_csu_init+0> push r15
$rsi   : 0x0000000000000006
$rdi   : 0x00007fffffffdef0  →  0x00000a6f6c6c6568 ("hello"?)
$rip   : 0x0000555555555430  →  <main+137> mov rsi, rax
$r8    : 0x000000000000000a
$r9    : 0x0000000000000000
$r10   : 0xfffffffffffff10c
$r11   : 0x00007ffff7b74e40  →  <std::__cxx11::basic_string<char,+0> mov rax, QWORD PTR [rdi]
$r12   : 0x00005555555551f0  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe000  →  0x0000000000000001
$r14   : 0x0000000000000000
$r15   : 0x0000000000000000
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$fs: 0x0000  $ds: 0x0000  $ss: 0x002b  $gs: 0x0000  $cs: 0x0033  $es: 0x0000
```

```
                american fuzzy lop 2.52b (python)

 process timing                                       overall results
        run time : 0 days, 0 hrs, 13 min, 1 sec       cycles done : 322
   last new path : none yet (odd, check syntax!)      total paths : 2
 last uniq crash : none seen yet                     uniq crashes : 0
  last uniq hang : none seen yet                       uniq hangs : 0
 cycle progress                       map coverage
  now processing : 1 (50.00%)           map density : 0.00% / 0.00%
 paths timed out : 0 (0.00%)         count coverage : 1.00 bits/tuple
 stage progress                       findings in depth
  now trying : havoc                 favored paths : 1 (50.00%)
 stage execs : 17/51 (33.33%)         new edges on : 1 (50.00%)
 total execs : 34.2k                total crashes : 0 (0 unique)
  exec speed : 49.59/sec (slow!)     total tmouts : 22 (1 unique)
 fuzzing strategy yields                               path geometry
   bit flips : 0/64, 0/62, 0/58                          levels : 1
  byte flips : 0/8, 0/6, 0/2                            pending : 0
 arithmetics : 0/448, 0/0, 0/0                         pend fav : 0
  known ints : 0/46, 0/168, 0/88                      own finds : 0
  dictionary : 0/0, 0/0, 0/0                           imported : n/a
       havoc : 0/33.2k, 0/0                           stability : 100.00%
        trim : 88.73%/16, 0.00%
                                                          [cpu000: 51%]
```

```
Final register state:
>>>   cs: 0x0000000000000033
>>>   ds: 0x0000000000000000
>>>  efl: 0x0000000000000044
>>>   es: 0x0000000000000000
>>>   fs: 0x0000000000000000
>>>   gs: 0x0000000000000000
>>>  r10: 0xfffffffffffff10c
>>>  r11: 0x00007ffff7b74e40
>>>  r12: 0x00005555555551f0
>>>  r13: 0x00007fffffffe000
>>>  r14: 0x0000000000000000
>>>  r15: 0x0000000000000000
>>>   r8: 0x000000000000000a
>>>   r9: 0x0000000000000000
>>>  rax: 0x0000555555757360
>>>  rbp: 0x00007fffffffdf20
>>>  rbx: 0x0000000000000000
>>>  rcx: 0x0000000000000000
>>>  rdi: 0x00007fffffffdef0
>>>  rdx: 0x000000000000000a
>>>  rip: 0x00005555555553ae
>>>  rsi: 0x000055555577a000
>>>  rsp: 0x00007fffffffdea8
>>>   ss: 0x000000000000002b
Done.
```

Figure 4.7: comparative test with afl-demo

Chapter 5

Results

5.1 Results and Caveats

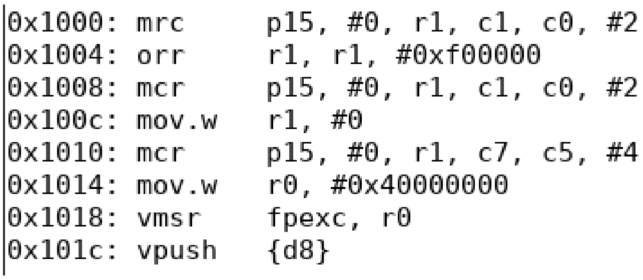
Many difficulties connected with this research result from the fact that almost all the documents produced by the **RTCA**, the **EUROCAE** and the **ICAO** are expensive commercial products and each of them ranges between 300 and 500 dollars. Therefore every piece of necessary information had to be acquired in an alternative way: either by conferences slides, papers or articles.

Regarding *dump1090* the extracted binaries behave differently from what was expected: the one from FlightRadar24 does not visually output any information on the screen but it is modified to upload data directly inside their network. On the other hand the binary extracted from the Stratux image have a strange behavior since, if fed with the *1090_small.bin* data file and given the *--no-crc-check* option it correctly outputs messages with CRC errors while the version from the *Antirez* repository, which is supposed to be the same as the extracted one, using the same file and parameters does not show such messages.

Working with the Unicorn Engine was neither easy nor straightforward. This emulator is really powerful and well developed; however, it comes with no documentation at all but just a couple of C or Python basic examples. Therefore, every time that more details were needed it was necessary to take a look at the source code and understand how to use a specific function or how a feature was implemented. Moreover, even if Unicorn is capable of emulating the ARM architecture in almost all his functionalities, it does not have default support for Vector Floating Point (VFP) instructions, which are widely used in the tested binary. Although this behavior, according to the ARM processor documentation, is the correct one, it resulted in the emulator not being able to recognize VFP instructions. As a consequence the template crashes while in execution. Following the ARM documentation and the issues in the Unicorn Engine repository it was possible to create a piece of Assembly code to enable VFP as in Figure 5.1 and 5.2.

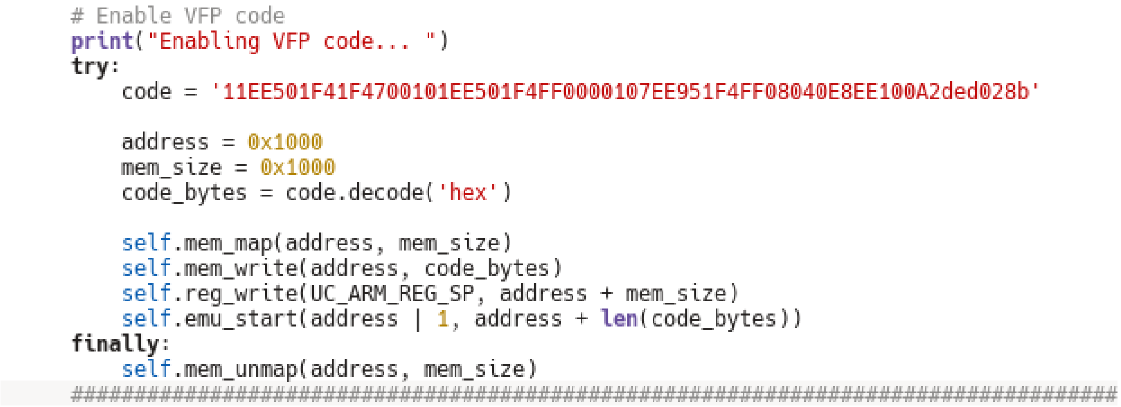
Even after the emulator was able to run VFP instructions a lot of debugging was needed to make a single template work, as it is quite common to have illegal memory access also in relation to library functions such as *puts, printf, etc* that need to be properly handled.

The Unicorn Engine has also a strange behavior when run on the multicore server. A quick investigation was carried out to understand more and the same



```
0x1000: mrc     p15, #0, r1, c1, c0, #2
0x1004: orr     r1, r1, #0xf00000
0x1008: mcr     p15, #0, r1, c1, c0, #2
0x100c: mov.w   r1, #0
0x1010: mcr     p15, #0, r1, c7, c5, #4
0x1014: mov.w   r0, #0x40000000
0x1018: vmsr    fpexc, r0
0x101c: vpush   {d8}
```

Figure 5.1: Assembly code to enable VFP instructions



```
# Enable VFP code
print("Enabling VFP code... ")
try:
    code = '11EE501F41F4700101EE501F4FF0000107EE951F4FF08040E8EE100A2ded028b'

    address = 0x1000
    mem_size = 0x1000
    code_bytes = code.decode('hex')

    self.mem_map(address, mem_size)
    self.mem_write(address, code_bytes)
    self.reg_write(UC_ARM_REG_SP, address + mem_size)
    self.emu_start(address | 1, address + len(code_bytes))
finally:
    self.mem_unmap(address, mem_size)
###################################################################################
```

Figure 5.2: Portion of python code to enable VFP instructions

template was run on different systems and different Operating Systems. In particular on two laptops, of which one was running Arch Linux 64bit with Linux Kernel 4.15.2 and the other one running Ubuntu Linux 16.04 LTS 32bit, the Unicorn Engine has exactly the same behavior on both Laptops.

The same behavior was also tested on a virtual machine running Fedora 27 64-bit while the same template run on the multicore server gives completely different results in terms of final register state. With the insertion of some debug lines inside the template it was possible to understand that on the server the emulation reaches immediately the end of the function, while the normal behavior observed in the other cases, establishes that the emulation should cycle inside the function. It is still unclear why the emulator has this behavior only on the server. It might depend on some problems regarding the size of the RAM or on the fact that it is a shared machine and the operating system runs inside a container or virtualized environment.

In addition to this, *afl-unicorn* was developed as an internal research project in a cybersecurity company. Although it is a great tool there is little to no documentation on his functioning and more importantly there is no real walkthrough on how to properly create a template for a binary. This is not a real problem since everything is clearly explained in the blog post about *afl-unicorn* and in the basic template provided. However, having to "work your way" through the tool is a great way to understand how it works but it requires quite a lot of time.

AFL had problems in finding crashes on the analyzed software and, in some cases, even in the intentionally vulnerable binaries. This can be caused by many factors:

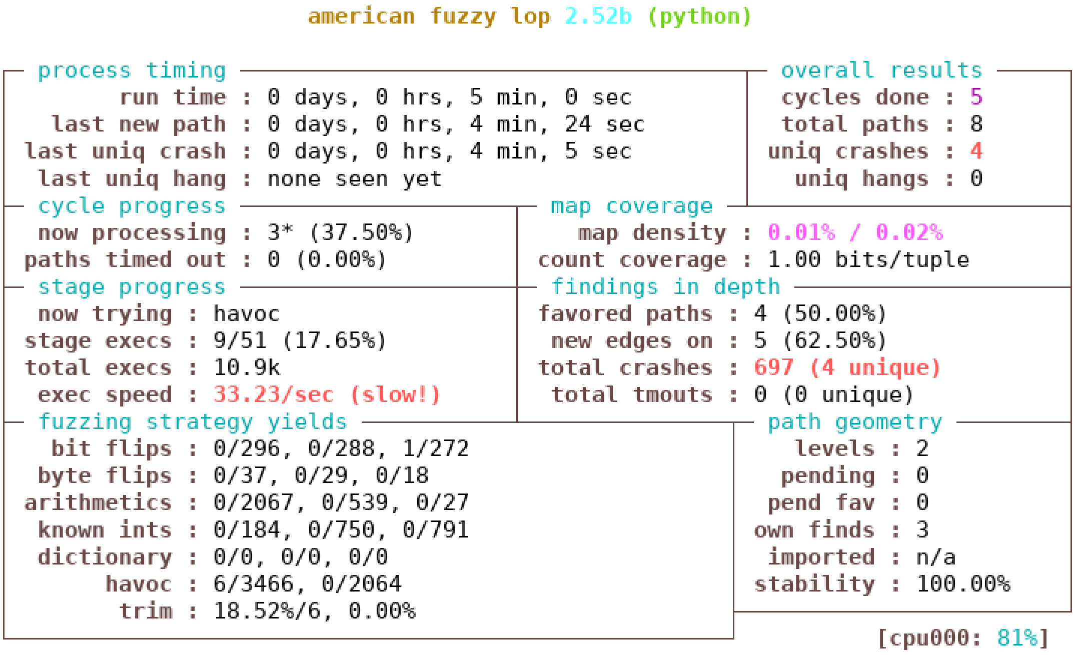
1. The used test input was not good enough, therefore, it will require a great deal of time for **AFL** to generate an input capable of triggering the vulnerability. This is also supported by the extremely low and slow growing code coverage obtained during the tests.
2. The vulnerability was actually triggered but the overflow did not cause any crashes, therefore resulting in a corrupted zombie program that might never crash. Moreover, **AFL** cannot detect memory leaks and other types of corruption in the target even though those are very likely security bugs. This type of problem is quite common in embedded devices as it has been widely detailed and analyzed in [48].
3. It is possible that the code being tested is written with security in mind and therefore no bugs are present in the code. The original author of *dump1090*, Salvatore Sanfilippo[1], has a strong security background and this might have influenced the security of code. However, the programs that have been tested are not the ones used in the real embedded systems in airplanes, which can have even worse implementations of the protocols.
4. Even if **AFL** represents the current state of the art in terms of binary fuzzing it might not be the proper tool to use in this situation.

As already mentioned acquiring data for *dump978* was a trivial process since it is not used in Europe. Moreover, *extract_nexrad* does not implement any functions to use Huffman encoding which was the most promising part since it is likely to be vulnerable. The sample **FIS-B** (text and image) data acquired from the GDL90 document [44] are not correctly decoded both by *extract_nexrad* and by *uat2text*. Further investigation discovered that there have been at least 2 revisions of the **RTCA** documents, such as RTCA DO-267A, that introduced changes in APDUs and packet formats. Some of those modifications, that are adopted in Europe under particular documents: European Technical Standard Order (ETSO), are detailed in the ETSO-C157a from European Aviation Safety Agency (**EASA**) [49]. This ETSO introduced incisive changes that, while the messages previously produced are still decoded as valid **UAT**, they are no longer correctly decoded as **FIS-B**, giving therefore random and useless results. It has been traced that ETSO-C157a has been introduced on 05/07/2012 and replaced on 16/12/2016 by ETSO-C157b which introduced further minor modifications.

As it can be seen in Figure 4.7 the different variations of **AFL** require different time to complete their jobs. In particular *afl-unicorn* behavior is interesting as it is the same in the *afl-test* binary and in *dump1090*. The most relevant fact is that the fuzzer is not able to trigger new internal states of the binary, meaning that no new paths are being discovered. This is completely different compared to the expected behavior that *alf-unicorn* has on the example binary and template that comes inside the repository and whose results can be seen in Figure 5.3

---

[1] `http://invece.org/`



```
                     american fuzzy lop 2.52b (python)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 5 min, 0 sec        │  cycles done : 5      │
│   last new path : 0 days, 0 hrs, 4 min, 24 sec       │  total paths : 8      │
│ last uniq crash : 0 days, 0 hrs, 4 min, 5 sec        │ uniq crashes : 4      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 3* (37.50%)       │    map density : 0.01% / 0.02%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 4 (50.00%)             │
│ stage execs : 9/51 (17.65%)         │  new edges on : 5 (62.50%)             │
│ total execs : 10.9k                 │ total crashes : 697 (4 unique)         │
│  exec speed : 33.23/sec (slow!)     │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/296, 0/288, 1/272                   │    levels : 2          │
│  byte flips : 0/37, 0/29, 0/18                      │   pending : 0          │
│ arithmetics : 0/2067, 0/539, 0/27                   │  pend fav : 0          │
│  known ints : 0/184, 0/750, 0/791                   │ own finds : 3          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 6/3466, 0/2064                        │ stability : 100.00%    │
│        trim : 18.52%/6, 0.00%                       ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 81%]
```

Figure 5.3: Run of afl-unicorn on the example template

There are some other tools and fuzzers that can be used to test for more vulnerabilities, for example some of the forks of **AFL** might produce better results and efficiently test these binaries. Other tools such as LLVM, which includes a fuzzer, a memory and an address sanitizer and Valgrind might be used to explore areas that **AFL** is not able to test.

Chapter 6

Conclusions and Future Work

This is the first public available research aiming towards security fuzzing of **ADS-B**, **Next-Gen** protocols and avionics devices. Some state of the art and effective methods to start fuzzing such devices and software have been used in the research and presented in this dissertation. This type of security analysis is particularly untrivial as it presents many caveats and difficulties associated with the availability of real hardware and software and to characteristic aspects of the RF world.

As previously anticipated the tools used in the research might not be the best to test this kind of software and protocols, therefore it is necessary to deeper explore the avionics field using the new and now available RF fuzzers previously mentioned in Chapter 4. Moreover, the same author of *librtlsdr* recently released another library (*fl2k*[50]) which leverages cheap usb3-to-VGA adapters turning them into devices capable of transmitting on Radio Frequencies. This, combined with the cheap real avionics devices found on eBay by Hugo Teso [9], might allow cheaper researches in this field.

Another option might be to develop a specific fuzzer or fork of **AFL** that can correctly interact with RF facing software. In particular regarding *dump1090*, although it is also interesting to explore what happens when the CRC of a message is incorrect, it might be useful to have a tool to generate random inputs that have a valid CRC.

Still linked with *dump1090* one of the main problems was to interact with modulated data, as the structure of this binary does not allow to test all its specific parts individually. Moreover, in all the samples some random and unwanted noise is always present since all the raw output from the RTL-SDR dongle is saved. This can be disadvantageous especially in the fuzzing task as many bits will be mutated without any direct effect on the target. In addition to this, large files slow down the fuzzing and are more difficult to be handled by **AFL**. For these reasons it might be very useful to create a tool which lets the user open a captured file and visually see its content. The user will then select the relevant part and create a new file. In this way it can also be easier to insert some controlled intentional noise inside an input file that will then be mutated since it is still interesting to see how the demodulator behaves in presence of fuzzed noise. Such tool would be beneficial both for 1090MHz and for 978MHz and will be, basically, an advanced modulator.

The revisions of the **RTCA** documents, which introduced major modifications, bring some questions on how real avionics components would behave in presence of an old software and a new packet and vice versa. Moreover it would be interesting to understand how the update is released and what are the different steps of the update process as it can introduce further problems such as orphan lines of code or phantom functions which might be exploitable. This certainly needs further investigation.

Also, all the tools, utilities and datasets as well as intentionally vulnerable samples used during this research are released in a open source repository. Hopefully this will help motivate further research in this field.

Bibliography

[1] Ms. Smith. *Homeland Security team remotely hacked a Boeing 757*. URL: https://www.csoonline.com/article/3236721/security/homeland-security-team-remotely-hacked-a-boeing-757.html.

[2] Amazon. *Prime Air Service - drone delivery system*. URL: https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011.

[3] Google. *Project Loon*. URL: https://x.company/loon/.

[4] PrecisionHawk. *Drones Sensors*. URL: https://www.precisionhawk.com/.

[5] Vigili del Fuoco. *Italian Firefighters adopting drones for SAR operations*. 2017. URL: http://www.vigilfuoco.it/aspx/notizia.aspx?codnews=40594.

[6] Facebook. *Aquila drones for global internet connection*. URL: https://info.internet.org/en/story/connectivity-lab/.

[7] Andrei Costin and Aurélien Francillon. "Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices". In: *BlackHat USA (BHUS)* (2012), pp. 1–12.

[8] Brad Haines. *Hackers + Airplanes*. Defcon 20 Slides. 2012. URL: https://renderlab.net/projects/ADS-B/Hackers-Airplanes-Defcon20-RenderMan.pdf.

[9] Hugo Teso. *Aircraft Hacking*. HTBSECCONF. 2013. URL: https://conference.hitb.org/hitbsecconf2013ams/materials/D1T1%20-%20Hugo%20Teso%20-%20Aircraft%20Hacking%20-%20Practical%20Aero%20Series.pdf.

[10] International Civil Aviation Organization (ICAO). *Convention on International Civil Aviation (doc. 7300/9)*. 1944. URL: https://www.icao.int/publications/Documents/7300%5C_cons.pdf.

[11] Google. *Project Loon ballons Mode C and ADS-B out*. URL: https://x.company/loon/faq/#flight-section.

[12] DJI. *Equipping drones with ADS-B*. URL: https://www.dji.com/newsroom/news/dji-and-uavionix-to-release-ads-b-collision-avoidance-developer-kit.

[13] flightradar24.com. *How it works*. https://www.flightradar24.com/how-it-works.

[14] flightaware.com. *FlightAware and ADS-B*. `https://flightaware.com/adsb/`.

[15] Matthias Schäfer et al. "OpenSky report 2017: Mode S and ADS-B usage of military and other state aircraft". In: *IEEE/AIAA Digital Avionics Systems Conference (DASC)*. IEEE. 2017, pp. 1–10.

[16] Woodrow Bellamy III. *Equipping 100,000 Aircraft*. `http://interactive.aviationtoday.com/avionicsmagazine/june-july-2016/equipping-100-000-aircraft/`.

[17] Junzi Sun. *The 1090MHz Riddle*. URL: `http://mode-s.org/decode/book-the_1090mhz_riddle-junzi_sun.pdf`.

[18] Ente Nazionale Assistenza Volo (ENAV). *Aeronautical Information Package (AIP) Italia*. URL: `https://www.aopa.it/pdf/AIRAC11-14.pdf`.

[19] Air France. *Information on ACARS system*. URL: `http://corporate.airfrance.com/en/acars-system`.

[20] E. Cook. "ADS-B, Friend or Foe: ADS-B Message Authentication for NextGen Aircraft". In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. Aug. 2015, pp. 1256–1261. DOI: `10.1109/HPCC-CSS-ICESS.2015.201`.

[21] International Civil Aviation Organization (ICAO). *ADS-B Implementation and Operations Guidance Document*. 2014. URL: `https://www.icao.int/APAC/Documents/edocs/cns/ADSB_AIGD7.pdf`.

[22] Martin Strohmeier et al. "Realities and challenges of nextgen air traffic management: The case of ADS-B". In: 52 (May 2014), pp. 111–118.

[23] European Union. "Commission Implementing Regulation (EU) 2017/386". In: *Official Journal of the European Union* (2017).

[24] federal government of the United States. *Electronic Code of Federal Regulations - Title 14 Aeronautics and Space*. 2017.

[25] David Salomon. *A Concise Introduction to Data Compression*. Springer, 2008. ISBN: 978-1-84800-071-1.

[26] Bran Haines. *They are doing WHAT! With Air Traffic Control*. Hackfest. 2014. URL: `https://renderlab.net/projects/ADS-B/Hackfest.pdf`.

[27] Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. "Security of ADS-B: State of the Art and Beyond". In: *IEE Communications Surveys & Tutorials* (July 13, 2013). arXiv: `http://arxiv.org/abs/1307.3664v1 [cs.CR]`.

[28] International Civil Aviation Organization (ICAO). *Global Operational Data Link (GOLD) Manual*. 2016.

[29] Pedram Amini Michael Sutton Adam Greene. *Fuzzing: Brute force vulnerability discovery*. Addison Wesley, 2007.

[30] Bryan So Barton P. Miller Lars Fredriksen. "An Empirical Study of the Reliability of UNIX Utilities". In: (1990). URL: `ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf`.

[31] Michal Zalewski (lcamtuf). *blogpost about afl input generation*. URL: `https://lcamtuf.blogspot.fi/2017/04/afl-experiments-or-please-eat-your.html`.

[32] Michal Zalewski (lcamtuf). *Technical Details of AFL*. URL: `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[33] mboehme. *AFLFast (extends AFL with Power Schedules)*. URL: `https://github.com/mboehme/aflfast`.

[34] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *ACM SIGSAC Conference on Computer and Communications Security* (2016).

[35] mboehme. *Pythia (extends AFL with Predictions)*. URL: `https://github.com/mboehme/pythia`.

[36] Nathan Voss. *afl-unicorn article*. URL: `https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf`.

[37] Nguyen Anh Quynh and Dang Hoang Vu. "Unicorn: Next Generation CPU Emulator Framework". In: *BlackHat USA (BHUS)* (2015). URL: `http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf`.

[38] Peach Fuzzer. *Platform Whitepaper*. URL: `https://www.peach.tech/wp-content/uploads/Peach-Fuzzer-Platform-Whitepaper.pdf`.

[39] Peach Fuzzer. *Software License*. URL: `http://community.peachfuzzer.com/License.html`.

[40] Defense Advanced Research Projects Agency (DARPA). *Cyber Grand Challenge (CGC)*. `https://www.darpa.mil/program/cyber-grand-challenge`.

[41] Jason Williams. *Cyber Grand Challenge (CGC) – FSK_Messaging_Service*. `https://github.com/trailofbits/cb-multios/tree/master/challenges/FSK_Messaging_Service`.

[42] Beyond Security. *Dynamic, Black Box Testing on the Radio Frequency Communications Protocols*. URL: `https://www.beyondsecurity.com/dynamic_fuzzing_testing_radio_frequency_communications_protocol_rfcomm.html`.

[43] Matt Knight and Ryan Speers. "TumbleRF: RF Fuzzing Made Easy". In: *BlackHat USA (BHUS)* (2018). URL: `https://www.blackhat.com/us-18/arsenal/schedule/index.html#tumblerf-rf-fuzzing-made-easy-12024`.

[44] Garmin. *GDL 90 Data Interface Specification*. 2007. URL: `https://www.faa.gov/nextgen/programs/adsb/Archival/media/GDL90_Public_ICD_RevA.PDF`.

[45] Radio Technical Commission For Aereonautics (RTCA). *DO-358 Supplement - Minimum Operational Performance Standards (MOPS) for Flight Information Services Broadcast (FIS-B) with Universal Access Transceiver (UAT).* `https://my.rtca.org/NC__Product?id=a1B36000001IcegEAC`.

[46] National Institute of Standards and Technology (NIST). *CVE-2017-6890.* 2017. URL: `https://nvd.nist.gov/vuln/detail/CVE-2017-6890`.

[47] CVE Details. *CVE-2007-4537.* 2007. URL: `https://www.cvedetails.com/cve/CVE-2007-4537/`.

[48] Marius Muench et al. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices". In: *Network and Distributed System Security (NDSS) Symposium*. NDSS 18. San Diego (USA), Feb. 2018.

[49] European Aviation Safety Agency (EASA). *List of all (current & historic) ETSOs.* URL: `https://www.easa.europa.eu/easa-and-you/aircraft-products/etso-authorisations/list-of-all-etso`.

[50] Open Source Mobile Communications. *Osmo-fl2k Wiki.* 2018. URL: `https://osmocom.org/projects/osmo-fl2k/wiki/Osmo-fl2k`.