MSSI CAPSTONE

Securing ADS-B Based Airborne Collision Avoidance Systems

TEAM

Purushottam A. Kulkarni, Ritvik Sachdev, Praveen Malhan

Advisors

Dr. Seth Nielson (JHUISI), Dr. Jonathan Petit (OnBoard Security Inc.)
ABSTRACT

With the increasing automation of Unmanned Aerial Vehicles (UAVs) it has become necessary to secure the protocols used for the communication and automated control of the same. This project aims at implementing an automated Airborne Collision Avoidance System (ACAS) to avoid collisions in real-time using Automatic Dependent Surveillance - Broadcast (ADS-B) messages. In addition to this, we demonstrate the insecurity of traditional ADS-B against common attacks such as packet forging, replay, message modification and Man-in-the-Middle (MITM) attacks. To protect against this, we make use of the Aerolink™ Library to provide the functionality of message authentication and integrity checking through short-term cryptographic signing of the messages to protect against all of the aforementioned attacks. Finally we present our results in the form of a video demo as well as the documented results of targeted fuzzing at the protocol level for ADS-B Based ACAS-Secure.
# Index

<table>
<thead>
<tr>
<th>Number</th>
<th>Title</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Automatic Dependent Surveillance - Broadcast (ADS-B)</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>ADS-B Message Types</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Actual ADS-B Implementation</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>The Experimental Setup</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>Weaknesses and Vulnerabilities in ADS-B</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Equipment Overview and the Collision Avoidance System</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Securing the Communication</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Testing and Analysis of Results</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>References</td>
<td>33</td>
</tr>
</tbody>
</table>
1. Introduction

With the increasing number of Unmanned Aerial Vehicles (UAV) in the air space, the biggest concern now becomes the safety of UAVs. A possible solution to this can be development of an on-board system which guides the drones into avoiding collisions. Since most radio controlled drones are based on communication protocols which make use of radio transmissions, we intend to write a protocol for these drones to avoid collision while maintaining authenticity and integrity of the messages by using Cryptography. We will implement our own Collision Control Protocol for these drones using the SDRM.

An aircraft surveillance system, Automatic Dependent Surveillance-Broadcast (ADS-B) is introduced by the FAA (Federal Aviation Administration) for the safety of aircrafts and avoid collisions. In the near future, ADS-B can replace radar as the primary surveillance method for controlling aircrafts worldwide[1]. ADS-B uses Global Positioning System (GPS) data to track the whereabouts of the airborne vehicle in real time and broadcasts this information in the airspace to improve situational awareness in the crowded airspace. Once this information is shared with all the other airborne vehicles, a Collision avoidance protocol instructs the UAVs to move in a certain direction, whenever another UAV is in close proximity. All the messages are sent in cleartext and hence it is easier for an attacker to intercept them and modify them. If an adversary can modify these messages, they can actually control the UAV and make it move as per their own needs.
This creates a need for the implementation of a cryptographically secure protocol which will prevent such attacks from happening. Added to this is the fact that drones have grown massively popular in the last few years and can have multiple mounted devices on them besides just the drone hardware. This problem can be solved by implementing a cryptographically secure Collision Control Protocol and could further existing research done in the Vehicle-2-Vehicle (V2V) security standard to provide safety by ensuring the security of the transmissions between vehicles and their controllers.

For the purpose of this project, we have partnered with OnBoard Security Inc. We are working with Dr. Johnathan Petit and Mr. Drew Van Duren as our external mentors for this project. The aim of this project is to adapt and use ADS-B for the purpose of implementing a simplified version of the Airborne Collision Avoidance System (ACAS) which is in turn secured by the Aerolink™ API graciously provided to us by the team at OnBoard Security, Inc. under a Non-Disclosure Agreement.

There are several data links for ADS-B, the most common ones operate at 1090 MHz[^2]. In our implementation, we are using 802.11 as the primary data link for communication of all the messages. ADS-B over 802.11 is implemented using a powerful packet manipulation library built in python known as Scapy[^3]. Scapy provides with the capability of forging or decoding packets. We used scapy to forge 802.11 beacon frames to transmit and receive ADS-B messages since our communication happens over WiFi and not a radio channel.

In real time applications, aircrafts are mounted with the ADS-B modules broadcasting over 1090 MHz and sending out the GPS data, velocity and identification information. In our
implementation we have a Parrot Bebop 2 drone controlled by a ground station (not mounted over the drone). Scapy is used to send out messages over WiFi instead of radio channel and all the messages from the drone are decoded by using a python library known as PyModeS\(^{[4]}\). The project is a simplified version of the real scenario and the code can be ported over to a Raspberry Pi and mounted on the drone without having to control it from a ground station.
2. Automatic Dependent Surveillance - Broadcast (ADS-B)

ADS-B is a satellite based surveillance system mounted over an aircraft to capture and send out position, velocity and the identification transmitted through a Mode S transponder usually over 1090 MHz \(^5\).

An ADS-B message is 112 bits long and consists of 5 message parts. The message structure can be seen as:

<table>
<thead>
<tr>
<th>DF (5)</th>
<th>CA (3)</th>
<th>ICAO (24)</th>
<th>DATA (56)</th>
<th>PI (24)</th>
</tr>
</thead>
</table>

The table below lists details of the message structure:

<table>
<thead>
<tr>
<th>nBits</th>
<th>Bits</th>
<th>Abbr.</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 - 5</td>
<td>DF</td>
<td>Downlink Format (17 or 18)</td>
</tr>
<tr>
<td>3</td>
<td>6 - 8</td>
<td>CA</td>
<td>Capability (additional identifier)</td>
</tr>
<tr>
<td>24</td>
<td>9- 32</td>
<td>ICAO</td>
<td>ICAO aircraft address</td>
</tr>
<tr>
<td>56</td>
<td>33 - 88</td>
<td>DATA</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>[33 - 37]</td>
<td>[TC]</td>
<td>Type code</td>
</tr>
<tr>
<td>24</td>
<td>89 - 112</td>
<td>PI</td>
<td>Parity/Interrogator ID</td>
</tr>
</tbody>
</table>

Any ADS-B message must start with the Downlink Format of 17 or 18 (10001 or 10010) for the first 5 bits. ADS-B uses cyclic redundancy check (CRC) to validate if the messages
received are correct or have been tampered with. The Parity bits (last 24 bits) are used for the CRC validation.

Example:

```
Raw message in hexadecimal:
8D48400D6202CC371C32CE0576098
```

<table>
<thead>
<tr>
<th>HEX</th>
<th>8D</th>
<th>48400D</th>
<th>202CC371C32CE</th>
<th>0576098</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN</td>
<td>100101</td>
<td>1001000000100</td>
<td>100011001110</td>
<td>11110010110</td>
</tr>
<tr>
<td>DEC</td>
<td>17</td>
<td>5</td>
<td>[4]</td>
<td>[DATA]</td>
</tr>
</tbody>
</table>

**2.1 ADS-B Message types:**

There are 5 message types sent to or received from an aircraft. Type Code is specified in the first 5 bits (33-37) of the Data segment. The type code related to the message definition is as below:

<table>
<thead>
<tr>
<th>TC</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 4</td>
<td>Aircraft identification</td>
</tr>
<tr>
<td>5 - 8</td>
<td>Surface position</td>
</tr>
<tr>
<td>9 - 18</td>
<td>Airborne position (w/ Baro Altitude)</td>
</tr>
<tr>
<td>19</td>
<td>Airborne velocities</td>
</tr>
<tr>
<td>20 - 22</td>
<td>Airborne position (w/ GNSS Height)</td>
</tr>
<tr>
<td>23 - 31</td>
<td>Reserved for other uses</td>
</tr>
</tbody>
</table>
Aircraft Identification: Bits 9-24 are used to identify the aircraft address. As per the above table, the Type Code will be 1-4 in the data segment. Let’s consider the HEX message received is ‘8D4840D6202CC371C32CE0576098’. The structure of the message is as below:

![HEX message structure](image)

The Downlink Format is 17 and type code value is 4 (which indicates the message is for aircraft identification). The remaining part of the data is ‘2CC371C32CE0’. Once this message is changed to decimal, the decimal to character conversion is as follows:

A-Z: 1-26
0-9: 48-57
Special characters: 27-47
_: 32 (the only special character in this message)

The decoded aircraft identification is KLM1023_.
Airborne Positions: An aircraft broadcasting its position in the airspace primarily will have Type Code as 9-18. The message is composed of the following:

<table>
<thead>
<tr>
<th>MSG bits</th>
<th># bits</th>
<th>Abbr</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>5</td>
<td>DF</td>
<td>Downlink format</td>
</tr>
<tr>
<td>33-37</td>
<td>5</td>
<td>TC</td>
<td>Type code</td>
</tr>
<tr>
<td>38-39</td>
<td>2</td>
<td>SS</td>
<td>Surveillance status</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>NICsb</td>
<td>NIC supplement-B</td>
</tr>
<tr>
<td>41-52</td>
<td>12</td>
<td>ALT</td>
<td>Altitude</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
<td>T</td>
<td>Time</td>
</tr>
<tr>
<td>54</td>
<td>1</td>
<td>F</td>
<td>CPR odd/even frame flag</td>
</tr>
<tr>
<td>55-71</td>
<td>17</td>
<td>LAT-CPR</td>
<td>Latitude in CPR format</td>
</tr>
<tr>
<td>72-88</td>
<td>17</td>
<td>LON-CPR</td>
<td>Longitude in CPR format</td>
</tr>
</tbody>
</table>

There are two types of position messages (odd or even) broadcasted alternatively. Both the messages are used to calculate the Latitude and Longitude of the aircraft, however either one can be used to calculate the altitude of the aircraft. There are two scenarios:

1. **Unknown Position** (Globally unambiguous position): In this case, both types of messages are used to decode the position.
2. **Previous Known Position** (Locally unambiguous position): Here, only one of the messages can be used to decode the current position.
2.1.1 Encoding Airborne Position in ADS-B odd/even messages:

The encoding of ADS-B type code 17 (Airborne Position with Barometric Altitude) messages takes place using Compact Position Reporting (CPR)\[9\]. In broad terms, Compact Position Reporting allows the encoding of ADS-B messages in such a way that fewer bits may be used to encode more decimal coordinates. As mentioned above, it is achieved by trading global position ambiguity and time for local position accuracy. The code snippet below shows our implementation of ADS-B encoding.

\[
\begin{align*}
ca &= 5 \\
tc &= 11 \\
ss &= 0 \\
nicsb &= 0 \\
time &= 0 \\
surface &= \text{False}
\end{align*}
\]

\[(\text{even, odd}) = \text{df17_pos_rep_encode}(ca, \text{icao}, tc, ss, nicsb, alt, time, lat, lon, surface)\]

This code was taken from the ADS-B Out project by Github user lyusupov\[9\]. We repurposed the code in this project to emit the even and odd messages and convert them to an acceptable format to us to transmit over WiFi.

In the snippet given above we can see that the variables ica, lat, lon and alt have not been set. The reason being that we take them in as arguments to the function which is called to trigger the encoding.
2.1.2 Decoding Airborne Position from received ADS-B Messages:

In order to decode ADS-B messages from their CPR format, we make use of the pyModeS library. This library provides us with an excellent API to decode captured messages and retrieve latitude, longitude and altitude of an aircraft. We then timestamp this data for further processing. This can be seen from the following snippet of code.

```python
import pyModeS as pms
altitude = pms.adsb.altitude(msg["even"])
latitude, longitude = pms.adsb.airborne_position(msg["even"], msg["odd"], 0, 0)
```

The code snippet above shows the use of the pyModeS library to calculate the position of a drone from even and odd ADS-B messages received over WiFi. Here we see that the 'even' message is used to calculate the altitude. This is not necessary as an 'odd' message may also be used in this case. However, in order to calculate the position of the aircraft in terms of latitude and longitude, we make use of both, the even and odd messages.

2.2 Actual ADS-B Implementation

In the real world, ADS-B messages are broadcasted to ground stations as well as other aircraft over Radio at a frequency of 1090 MHz as mentioned before. As seen from the figure below, we see that two approaching aircrafts, Plane A and Plane B, broadcast ADS-B position messages over this frequency to be received by any aircraft or ground station.
These messages are in turn used to calculate the airborne position with Barometric Altitude. In the next section we will explain the differences and similarities between our experimental setup and the current implementation of ADS-B.

![Diagram of ADS-B Implementation](image)

**Fig. 2.1.1 The actual Implementation of ADS-B**

From the above figure we can see that ADS-B position messages from Plane A are broadcast to every other entity in the figure i.e. Plane B, Radio Station A and Radio Station B. Here Radio Stations A and B may be ground control towers for the planes as well as any tower with the capability to listen in on the 1090 MHz frequency of ADS-B. This introduces an obvious security issue in ADS-B in terms of confidentiality of these messages.

Although we have slightly varied this implementation in our experimental set-up, this does not affect any issues that ADS-B inherently has in its actual implementation. This was done
keeping in mind our resource constraints and any legal issues that may come up when we perform actual security testing on ADS-B.

### 2.3 The Experimental Setup

Our experimental setup for the purpose of this projects modifies this existing implementation of ADS-B slightly due to limitations on the carrying capacities of the drones as well as limitations to the accuracy of GPS with respect to very small distances. In order to accommodate for these limitations, we eliminate the broadcast of messages by the Aircraft itself and instead perform this broadcast from ground stations (Laptop PCs) which control these drones remotely. We obtain the drone’s GPS coordinates through the Parrot SDK3. This experimental setup can be better explained with the help of figure 3.2 given below.
As seen from the figure above, we make use of 2 wireless interfaces per ground station. One is used to connect to and control the drone while the other is used as an ADS-B/802.11 transceiver in lieu of an actual Software Defined Radio. The transceiver sends and receives ADS-B messages which are used to calculate the positions of the drones, detect collisions using ACAS and then respectively transmit evasive maneuvers to the drones through the other wireless interface. Although this is not ideal considering the original implementation of ADS-B, we consider this an acceptable workaround considering the limitations mentioned above. The experimental setup is built on Virtual Machines such that the setup can be readily ported on a Raspberry Pi and be mounted on UAVs that don't have the weight restrictions.
3. Weaknesses and Vulnerabilities in ADS-B

The ADS-B solution seems like a pretty good solution, however, there are a few security concerns and weaknesses in the system that need to be addressed before it can completely take over the radar surveillance system. The primary concern is obviously the use of plaintext and unencrypted messages. The only protection the communication offers is error-code protection while using CRC to validate the messages. However, it is not difficult for an adversary to modify data and parity bits if the communication takes place in plaintext.

That being said, it is also important to note that these messages are also in no way authenticated to come from the right source. Therefore, an attacker can also fake an ADS-B message using relatively unsophisticated equipment like an SDR. Our goal in this project is to secure ADS-B using public-key cryptography and ensure confidentiality, authenticity and integrity of these messages.

ADS-B lacks the following security mechanisms to be addressed:

- lack of entity authentication to protect against message injection from unauthorized entities.
- lack of message signatures or authentication codes to protect against tampering of messages or impersonating aircrafts.
- lack of message encryption to protect against eavesdropping.
- lack of challenge-response mechanisms to protect against replay attacks.
- lack of ephemeral identifiers to protect against privacy tracking attacks.
An adversary can have two different positions while trying to attack an ADS-B system. Let's consider the attacker is ground-based. This type of an attacker is commonly presented or envisioned. Consider a malicious ground station capable of broadcasting a fake ADS-B signal. This can be used to confuse the pilot of a plane receiving these messages or another ground station by spoofing the position of a plane and cause them to change their trajectory or even land. This exploit is sufficiently proven by the ADS-B Out [10] tool developed by Github user lyusupov. An ADS-B message can easily be spoofed by the use of an actual Software Defined Radio capable of transmitting arbitrary radio signals. This can be explained with the help of the following figure.

**Fig 3.1 Aircraft Spoofing Attack on Traditional ADS-B**
The figure above depicts an attack which can be performed on an aircraft (Aircraft A) listening to ADS-B messages to detect other aircrafts in its vicinity. As seen above, an attacker can create a malicious broadcasting station with the help of an SDR and a connected PC. She can then forge and emit fake ADS-B position messages to Aircraft A which will cause it to assume that another aircraft is present at the position which is being broadcast by the attacker. As an evasive maneuver, Aircraft A would then have to change its path to go around the non-existent obstacle. At the same time, the attacker can also listen in on any ADS-B messages being relayed between the Aircraft and Ground Control Station A, thereby leaking it’s airborne position as well as other information such as velocity, callsign etc. without being detected.

The other scenario is when the attacker is airborne. This type of an attacker is not very well understood or modelled. However, leveraging technological advancements like drones and SDRs, it is easier to attack the ADS-B systems. The drones can easily carry a Raspberry Pi with exploits which can be used by a sophisticated attacker to take ‘control’ of the drones by causing them to arbitrarily change their path.
4. Equipment Overview and the Collision Avoidance System

For the purpose of our project, we used a Parrot BeBop 2 drone which is a First Person View (FPV) micro quad drone which has an in-built video camera and can transmit live feed to the device controlling it. For our testing, we did not use an Android or iOS device, in fact we used a Linux machine. We used Scapy, a python library to craft ADS-B packets and then sent them to the drone over WiFi to control it. The broadcast messages from the drone were decoded using PyModeS. We switched the wireless interfaces to monitor mode for it to work as a transceiver.

The BeBop can be programmed using SDK3 to connect, pilot, receive stream, save and download medias (photo and video), send and play autopilot flight plans and update the drone. This SDK is mainly written in C, it provides libraries for Unix system, Android (Java) and iOS. Due to the popularity of these drones, many reverse engineered libraries exist in Python written by the developer community. We utilise one such library called ByBop to control the drone via a Linux/Unix system.

The drone has a default internal IP address 192.168.42.1. To negotiate a connection, we connect a socket to the TCP Discovery Connection port which is 44444, and send a JSON string containing the following informations:
Here is an example of a valid connection JSON string:

```
{ "d2c_port":43210, "controller type":"Phone",
  "controller name":"com.example.arsdkapp" }
```

The “device_id” field is useful when re-connecting to a product: If a product receives a connection request with the “device_id” field, the connection will only be accepted if it matches the product serial number.

The device will answer a connection request with another JSON string sent on the same TCP socket. This response JSON string can contain the following information:

<table>
<thead>
<tr>
<th>Key</th>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d2c_port</td>
<td>Yes</td>
<td>The UDP port you will use to read data</td>
</tr>
<tr>
<td>controller_type</td>
<td>Yes</td>
<td>The type of the controller (e.g., “Phone”, “Tablet”...)</td>
</tr>
<tr>
<td>controller_name</td>
<td>Yes</td>
<td>The name of the controller (e.g., Application name)</td>
</tr>
<tr>
<td>device_id</td>
<td>No</td>
<td>The product serial number</td>
</tr>
</tbody>
</table>
Here is an example of a valid connection answer JSON string:

```json
{  "status":0,  "c2d port":54321,
  "arstream fragment size":65000,
  "arstream fragment maximum number":4,
  "arstream max ack interval":-1,  "c2d update port":51,
  "c2d user port":61 }
```

Rest of the communication takes place through **UDP** over the specified **c2d** and **d2c** ports.

Values for throttle, elevation, rotation, and aile are in the range [-100, 100]. The video stream can be accessed over the Real-time Transport Protocol (RTP) over port 55004. A sample streaming configuration is shown below:

---

<table>
<thead>
<tr>
<th>Key</th>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>Yes</td>
<td>If different from 0, it means that the connection is refused (see below)</td>
</tr>
<tr>
<td>c2d_port</td>
<td>Yes</td>
<td>The UDP port you will send data to (If the connection is refused, this value will be 0)</td>
</tr>
<tr>
<td>arstream_fragment_size</td>
<td>No</td>
<td>The size of ARStream fragments</td>
</tr>
<tr>
<td>arstream_fragment_max</td>
<td>No</td>
<td>The maximum number of ARStream fragments per video frame</td>
</tr>
<tr>
<td>arstream_max_ack_interval</td>
<td>No</td>
<td>The maximum time between ARStream ACKs</td>
</tr>
<tr>
<td>c2d_update_port</td>
<td>Yes</td>
<td>The FTP port for updating the product</td>
</tr>
<tr>
<td>c2d_user_port</td>
<td>No</td>
<td>Another FTP port for other uses</td>
</tr>
<tr>
<td>skycontroller_version</td>
<td>SC only</td>
<td>The SkyController version</td>
</tr>
</tbody>
</table>
Bebop.sdp

The Bebop.sdp file allows us to extract the video stream from the Drone to view its flight in real-time

4.1 Collision Avoidance Protocol

Once we were able to forge packets for the drone and control it from our machines, we had to come up with a protocol to define how the drone acts when it is proximity to another drone. This is basically named as the Airborne Collision Avoidance Protocol. Due to the lack of time and resources, we are implementing a very basic protocol which is:

“The Collision Avoidance protocol states that the drone will go left until collision is not detected anymore and then move back to its original path.”

Implementing the protocol:

Collision detected: keep giving values that make drone go left (aile +), count the number of cycles that this value has been sent

No More Collision in scope: Get the number of cycles drone got GO-LEFT command send GO-RIGHT command that many times.
5. Securing the Communication

As discussed in the earlier section, ADS-B communication is completely unencrypted and insecure. To secure our communication with the drone, we have made use of the Aerolink™ cryptographic library developed in-house by OnBoard Security. Due to the presence of a Non-Disclosure Agreement, we are bound to not disclose any cryptographic library specific details.

There is no encryption involved due to the time delay it might add to the communication which might slow down responses of the drones. All the broadcast messages sent out by the drone are hashed and then signed. This maintains the authenticity and integrity of the messages. There are two modes of communication, one without any sorts of protection and the other by using the signed messages. If we are communicating in the secure mode, the drones do not accept any unsigned or empty messages. In the secure mode, an adversary can no longer forge or change the message in transit due to the absence of a legitimate signing key.

The drones do not accept any messages that are more than 600 seconds older. This means that due to the presence of timestamps, replay attack also does not seem to be a possibility.
6. Tests and Analysis of Results

In order to prove that we were successfully able to implement a secure version of ADS-B, we developed 7 test cases, including a base case with cryptographic signing disabled, which covered the following areas:

1. Cryptographic signing disabled on both the sender and the receiver
2. Cryptographic signing enabled on the receiver but not the sender
3. Sending a malformed message from the sender and verifying it on the receiver
4. Sending a stored message from the past (more than 600 seconds old)
5. Sending a fake message from future by changing system time
6. Sending a message with a modified payload
7. Cryptographic signing enabled on the sender and the receiver

This comprehensive testing allowed us to prove that the protocol which we developed was actually secure against replay, message modification, message forging, and MITM attacks. This section describes each of these test cases in detail and also provides an insight into the different error messages that are received for each failed test case.

6.1 Test Case 1 - Base Case

In the base case, we disabled cryptographic signing on both the sender and the receiver. We set up the sender and the receiver on two separate Virtual Machines (VMs) with separate wireless interfaces. As shown in figure 6.1, we got the expected result. The protocol was able to obtain and decode all of the messages correctly. However, these messages would be indistinguishable in the case that an attacker encoded them himself and therefore this protocol is inherently insecure when the secure mode is disabled.
A TCPdump on the interface running the receiver would be able to trivially prove this and therefore satisfies the requirement for this test. The protocol transmits and receives 10 messages per second.

![TCPdump output]

**Fig 6.1 Test Case 1**

As seen from the figure above, we are able to recover the location of the receiver along with a timestamp in the form of a real-time stream. Our protocol therefore passes the base test case.

**6.2 Test Case 2 - Signing Disabled/Verification Enabled**

In the second test case, we disabled message signing on the sender, but did enable message verification on the receiver. Similar to test case 1, we set these up on two separate VMs with separate wireless interfaces. The messages transmitted from the sender were outright rejected by the receiver and therefore proves that such a mismatched mode of operation on the sender and receiver respectively does not persist due to a decompression
error. We simply drop the packets which give this type of error and continue to accept only properly signed packets for location calculation.

As seen from the figure above, in Test Case 2, the receiver throws a decompression error and does not continue with location calculation. Our protocol therefore passes this test as well with flying colors.

6.3 Test Case 3 - Garbage Message which is compressed but not signed.

For our third test case, we generated a message of 460 ‘A’s. We did this as 460 bytes is the size of the signed message which the Aerolink™ library generates. We then compressed this message so that the decompression error could be avoided. We transmitted this message to the receiver using our ADS-B API to observe how the protocol reacted to this case. As seen in Fig. 6.3, our protocol throws a PARSE_FAIL error in this case and outright rejects the message. This proves that a message that is malformed by an attacker in transit
cannot be verified by the cryptographic library, thereby giving protection against arbitrary message attacks.

**Fig 6.3 Test Case 3**

The figure above shows how the protocol throws a PARSE_FAIL error in the verification binary when a properly compressed but unsigned message is received. This proves that our protocol passes this test as well with great accuracy and as a result is secure against such attacks.

**6.4 Test Case 4 - Replay Attack**

The fourth test case comprised of a simulated replay attack. We generated fake messages which were properly signed by the correct keys. We stored these messages for over 600 seconds and then retransmitted them to a receiver. The setup for this test was the same as the previous test cases with the sender and the receiver on two separate VMs. As expected the protocol threw a MESSAGE_TOO_OLD error on receiving each of these messages as
shown in Fig 6.4. This just proves that our protocol is not vulnerable to the traditional replay attack and therefore achieves its goal of being secure against the same.

The figure above shows the MESSAGE_TOO_OLD error clearly proving that our protocol stands tough in the face of a replay attack. This shows that we achieved our goal of securing ADS-B against such attacks.

6.5 Test Case 5 - Future Message Attack

In this test case we generated messages which appeared to come from the future by setting our system time to the future and signing messages with the correct keys. This test case was written to show that even the condition that a malicious message in the future is correctly handled by the Aerolink™ API. Similarly, it also proves that both the sender and the receiver must be time-synched in order to correctly sign and verify messages. This in turn uncovers a drawback to such a real-time signing and verification methodology. As seen in
Fig. 6.5, we obtain the MESSAGE_IN_FUTURE error when such a message is sent by an attacker. This message is again rejected outright by our protocol and therefore not used for localization calculation.

Fig 6.5 Test Case 5

The above figure demonstrates the MESSAGE_IN_FUTURE error message thereby proving that our protocol is secure against Future Message Attacks. This proves that such an attack is also unviable against our system and also that both the sender and receiver must be time-synched in order to perform signing and verification correctly.

6.6 Test Case 6 - Message Modification Attack/MITM

The sixth test case contained a Man-In-The-Middle which intercepted and modified the payload of our ADS-B Secure messages and relayed them to the receiver. This test case produced a PARSE_FAIL error on the receivers end thereby proving that messages which
are modified in transit are also rejected by our protocol. This proves the security of our protocol against such MITM or Message Modification Attacks. This provides the crucial property of Integrity to the system which we developed.

Fig 6.6 Test Case 6

The above figure demonstrates the protocol passing the MITM or Message Modification test case in a resounding fashion. This proves that our solution provides integrity checks to ADS-B and therefore succeeds in achieving one of its primary goals.

6.7 Test Case 7 - Correct Operation Test

The final test case in our testing cycle aimed to show that the protocol was able to successfully verify the signatures that were correctly generated by a sender who had access to the right keys. This was done in order to prove that our protocol provided the property of authentication to the sender and the ability to verify the identity of the
sender to the receiver. This test demonstrated the base operation of ADS-B Secure and proved that our receiver could verify the authenticity of the messages. All such correctly signed messages were immediately accepted by the sender and used for position calculation.

Fig 6.7 Test Case 7

The above figure shows that when the messages are correctly signed using the proper keys and are not malformed in any way, they can be used to:

1. Verify the authenticity of the sender
2. Calculate, with high accuracy, the GPS position of the sender.

6.8 Testing Analysis

From the tests that we conducted, we were able to successfully prove that our protocol was secure against MITM, Message Modification, and Replay attacks. It was able to provide the properties of Integrity and Authenticity to traditional ADS-B. This result shows that we were able to successfully achieve all the goals that we set out to achieve when we took on this project. Thus ADS-B Secure under Aerolink™ is a viable solution to the inherent insecurity of ADS-B.
7. Conclusion

We surveyed a number of academic publications which are openly available to determine how ADS - B is broken due to the absence of any existing cryptographic functionality in the protocol itself. Two of these publications [1] & [2] demonstrate this and also suggest possible security measures. As mentioned above, we have completed the security evaluation of ADS-B. As a proof of concept, we have developed tools to replay, forge and decode captured ADS-B messages which can be found at our github repository [11]. Additionally we have also found a number of open-source tools to implement ACAS as well. However, these seem to be at an initial stage of development and therefore, we might need to develop these on our own as well or build on the development done by the current author [12] & [13].

We were able to successfully implement a secure version of ADS-B using the Aerolink™ cryptographic API. Due to a standing Non-Disclosure Agreement with OnBoard Security, Inc., we are unable to disclose the full functionality and the source code of the binaries we wrote for message signing and signature verification as a part of this report. However, we were able to successfully prove the security of the newly developed ADS-B Secure protocol against MITM, Message Modification, Replay and Message Forging attacks using this API and our own implementation of ADS-B over 802.11 using targeted fuzzing against our implementation under a variety of test cases.

Using this implementation we also were able to develop our own version of an Airborne Collision Avoidance System (ACAS). We successfully tested out this implementation on Parrot Bebop 2 drones to prove that our solution is not only functional, but also secure under the aforementioned attacks. We therefore provided security in the form of Physical Security (ACAS) as well as Integrity and Authentication to an inherently insecure system.
8. References

[1] EXPLOITING THE AUTOMATIC DEPENDENT SURVEILLANCE-BROADCAST SYSTEM VIA FALSE TARGET INJECTION

https://media.blackhat.com/bh-us-12/Briefings/Costin/BH_US_12_Costin_Ghosts_In_Air_WP.pdf

[3] Scapy Documentation
https://scapy.readthedocs.io/en/latest/

[4] PyModeS Documentation
http://pymodes.readthedocs.io/en/latest/


[6] ADS-B Message Types
http://adsb-decode-guide.readthedocs.io/en/latest/content/introduction.html#ads-b-message-types

[7] ADS-B Aircraft Identification

[8] ADS-B Aircraft Positions

[9] Compact Position Reporting
https://adsb-decode-guide.readthedocs.io/en/latest/content/cpr.html

https://github.com/lyusupov/ADSB-Out/


https://github.com/ritviksach/capstone

[12] Airborne Collision Avoidance System (ACAS)

https://github.com/xueyizou/ACASX_3D

[13] ACAS 2

https://github.com/wulfebw/acas_xu