Triton: A Software-Reconfigurable Federated Avionics Testbed

Sam Crow* Brown Farinholt* Brian Johannesmeyer† Karl Koscher‡ Stephen Checkoway§
Stefan Savage* Aaron Schulman* Alex C. Snoeren* Kirill Levchenko¶
**UC San Diego* †*VU Amsterdam* ‡*University of Washington* §*Oberlin College* ¶*University of Illinois*

Abstract

This paper describes the Triton federated-avionics security testbed that supports testing real aircraft electronic systems for security vulnerabilities. Because modern aircraft are complex systems of systems, the Triton testbed allows multiple systems to be instantiated for analysis in order to observe the aggregate behavior of multiple aircraft systems and identify their potential impact on flight safety. We describe two attack scenarios that motivated the design of the Triton testbed: ACARS message spoofing and the software update process for aircraft systems. The testbed allows us to analyze both scenarios to determine whether adversarial interference in their expected operation could cause harm. This paper does *not* describe any vulnerabilities in real aircraft systems; instead, it describes the design of the Triton testbed and our experiences using it.

One of the key features of the Triton testbed is the ability to mix simulated, emulated, and physical electronic systems as necessary for a particular experiment or analysis task. A physical system may interact with a simulated component or a system whose software is running in an emulator. To facilitate rapid reconfigurability, Triton is also entirely software reconfigurable: all wiring between components is virtual and can be changed without physical access to components. A prototype of the Triton testbed is used at two universities to evaluate the security of aircraft systems.

1 Introduction

Factories, chemical plants, automobiles, and aircraft have come to be described today as cyber-physical systems of systems—distinct systems connected to form a larger and more complex system. For many such systems, correct operation is critical to safety, making their security of paramount importance. Unfortunately, the defining characteristics of these systems, namely their heterogeneity and special purpose, make them hard to analyze. Today's security analysis tools are tailored to the analysis of server, desktop, and mobile software; analyzing systems of systems requires tools and techniques that can handle multiple heterogeneous systems working together to form a larger whole.

This paper is concerned with enabling the security analysis of a particular class of cyber-physical systems of systems, namely those of commercial transport[1] aircraft exemplified by the Boeing 737. Aircraft of that design era consist of a large number of discrete electronic systems interconnected by digital communication links. Hence, a security analysis requires determining whether an attacker who gains control over some number of constituent systems would be able to adversely affect flight safety. It should be noted that aircraft such as the Boeing 737, at least until the recent MAX series, allowed the pilot to completely override all electronic control of the aircraft, so that even in the worst case of complete compromise of all electronic systems, a skilled pilot could continue to fly the plane. However, a *security* analysis is distinct from a *failure* analysis in that a sophisticated adversary can present the appearance that everything is working correctly, leading the pilot to leave control of the aircraft to electronic systems.

[1]We focus on commercial airliners used to transport people and cargo.

For the results of a security analysis to be believed, the analysis must necessarily be carried out on the genuine article—an aircraft. Unfortunately, a Boeing 737 aircraft costs several million dollars and requires a ground crew to keep operational. However, because our analysis is only really concerned with the electronic systems, having an actual airframe—fuselage, engines, and all—adds little to the fidelity of a security analysis. Indeed, at a minimum only the specific systems under analysis are required, provided that the rest of the aircraft electronic environment can be adequately simulated.

This paper describes Triton, an avionics testbed that allows one or more aircraft electronic systems to be studied in an electronic environment resembling a field deployment. Triton is a *cross-mode* testbed, meaning that it enables physical, simulated, and emulated components to interact to orchestrate a specific experiment or scenario. For a example, a physical Flight Management Computer can communicate with a Communication Management Unit running in an emulator, interacting with a simulated VHF data radio.

To motivate the design of our testbed, we focus on two security *analysis tasks*: determining whether an adversarially crafted spoofed ACARS message could interfere with correct operation of aircraft systems that could affect the safety of flight, and evaluating the security of the software update process for aircraft electronic systems. While we designed the Triton testbed to support these two tasks, the testbed can be equally well support other kinds of analysis tasks. Furthermore, we expect that both the design, and our experiences building the testbed, will be of interest to other security researchers working with complex systems of systems.

The rest of this paper is organized as follows. Section 2 presents technical background, including specifics of aircraft systems, necessary for the rest of the paper. Section 3 describes the two analysis tasks in more detail. Section 4 then described the design of the Triton testbed. Section 5 briefly describes several experiments enabled by the testbed. Section 6 discusses our experience with our testbed. Section 8 concludes the paper.

2 Background

Electronic systems used in aircraft—termed *avionics*—span a wide range of functionality and design assurance. Avionics used on transport aircraft have evolved from independent electronic systems on the earliest aircraft, to separate interconnected systems, to fully integrated systems today. Separate interconnected avionics systems are called *federated avionics*, while the fully integrated systems are termed *integrated modular avionics*.[2] The most common transport aircraft in the skies today, including the Boeing 737 and Airbus 320 series, are of the federated avionics type. This important class of avionics is the target of our study.

2.1 Federated Avionics Architecture

In a federated avionics architecture, system functionality is implemented in discrete Line-Replaceable Units (LRUs). Figure 1 shows part of a simple federated avionics configuration that is responsible for air-ground communication using the Aircraft Communications Addressing and Reporting System (ACARS) on a Boeing 737 aircraft. ACARS, which allows aircraft to communicate with airline ground-based systems using 220-byte messages, is described in more detail in Section 2.3. Each box in the figure is a physically separate LRU connected to other units via serial communication links known by their standard number, ARINC 429; Section 2.2 describes these in more detail.

We briefly describe the high-level functionality of each LRU in Figure 1.

VHF Data Radio (VDR). The physical and link layer of ACARS is handled by the VHF Data Radio (VDR), which includes both the VHF transceiver and modem [6].

Communication Management Unit (CMU). In a federated avionics aircraft, the higher layers of ACARS communication are handled by the Communication Management Unit (CMU), a general-purpose communication gateway between aircraft systems and the outside world [9]. The CMU receives an incoming message from the VDR and either processes the message itself or relays it to another aircraft system, depending on the type of message.

Flight Management Computer (FMC). The FMC provides a variety of flight planning and execution tools. In particular, the FMC includes a navigation database, and can control the autoflight (autopilot and autothrust) system directly to fly a pre-programmed route.

[2]For transport aircraft, the transition from federated to integrated avionics occurred with the Boeing 777, which entered commercial service in 1995. All new Boeing and Airbus designs after the Boeing 777—namely the Boeing 787, the Airbus A380, and the Airbus A350 currently in development—use integrated avionics.

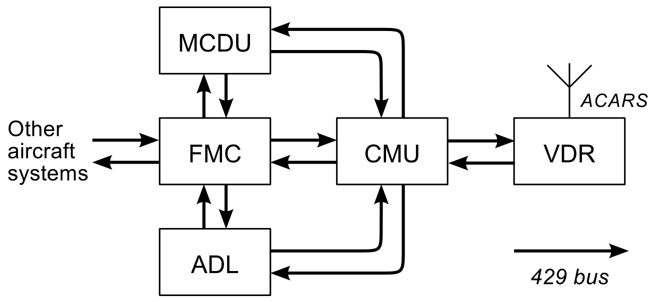

Figure 1: Fragment of Boeing 737 avionics interconnection centered at the Communication Management Unit (CMU).

Multi-Function Control and Display Unit (MCDU). The MCDU consists of a small display and keyboard located in the cockpit. It serves as the pilot interface to the CMU and FMC. In the case of former, for example, it displays ACARS messages to the pilot and allows the pilot to request certain types of information, such as destination airport weather reports, via ACARS. When interacting with the FMC, the MCDU displays the flight plan, and allows the pilot to automatically load a flight plan sent via ACARS from the airline's dispatcher.

Airborne Data Loader (ADL). The ADL is responsible for uploading software updates and navigation databases to various LRUs as well as downloading aircraft monitoring and flight data. The ADL consists of two parts: an in-cockpit control panel which selects which LRU to upload to or download from and a portable maintenance unit which performs the actual data transfer. Data transfer occurs via the ARINC 615 standard [4] which specifies the low-level network encoding and framing of data. Higher-level protocols like "upload firmware" or "download flight data" are not standardized. Instead, maintenance units are required to implement the data loading protocol for each LRU.

2.2 ARINC 429 Bus

Federated systems found on large transport aircraft are interconnected using the ARINC 429 bus, a unidirectional, multi-receiver bus [5]. Two-way communication between systems requires a pair of 429 buses, one in each direction, as shown in Figure 1.

ARINC 429 was originally designed to transmit simple status messages (as described below) and thus only supports sending a single 32-bit word at a time. Bits are sent as differential, bipolar, return-to-zero pulses at either 12.5 or 100 kbps. Eight of the 32 bits are reserved for the message label which identifies either the type of message or the destination LRU.

General word messages. General word messages are simple status messages broadcast by LRUs and identified by label. They may contain binary or binary-coded decimal values, as well as fields with discrete values. The most-significant bit is a parity bit, followed by two sign/status matrix bits, which indicate either the sign of the value encoded, or some other status (e.g., no computed data, failure warning, or a functional test result). A numeric value, padding, and discrete value fields may follow.

Character-Oriented Protocol (COP). ARINC 619 [8] defines a character-oriented protocol for sending character streams across a 429 bus. Originally intended for ACARS (see Section 2.3) applications (e.g., uploading a text message to the cockpit printer), it is also used to communicate with the MCDU. The COP can send up to three 7-bit characters at a time. A simple RTS/CTS, STX/ETX, and ACK/NAK scheme are used to ensure reliable delivery, with a single control character in the most-significant bits, followed by character-dependent control data. The destination is encoded in the message label.

Bit-Oriented Protocols (BOP). A limitation of the COP is that it only supports 7-bit characters. To transfer binary data, a byte stream had to be converted to a hexadecimal ASCII string first. The new bit-oriented protocols address this issue.

Version 1 of the BOP [8] is similar to the COP and transmits data words that can contain between 1 and 5 nibbles. A protocol word is used for flow control and protocol version negotiation. A solo word can transmit up to 16 bits without protocol overhead. The start-of-transmission word contains transmission metadata and the end-of-transmission word contains a 16-bit CRC.

2.3 ACARS

The Aircraft Communications Addressing and Reporting System (ACARS), defined in ARINC specification 618 [7], provides digital communication between an aircraft and ground systems using 220-byte messages. ACARS is used by airlines to track aircraft in flight and on the ground, by airline dispatcher to send flight plans to the cockpit, and by pilots to request and receiver weather reports, among other uses. Some ACARS downlink (aircraft-to-ground) messages are automatically generated by equipment on board, and some uplink (ground-to-aircraft) messages may be automatically acted on by systems on board. The original ACARS protocol used audio-frequency modulation sent using the AM-modulated VHF voice radio at a data rate of 2.4 kbps. While ACARS-over-AM continues to be used to this day, ACARS messages can also be carried over a newer 31.5-kbps data link called VHF Data Link Mode 2 (VDL2), HF Data Link, and satellite links. Neither the original ACARS protocol nor the data links commonly used to carry ACARS messages provides authentication, making it possible for an attacker to spoof both uplink and downlink messages. This raises some security concerns, as discussed below in Section 3.1.

3 Targeted Analysis Tasks

In designing the Triton testbed, our goal was to support a set of security analysis tasks, two of which we describe below.

3.1 ACARS Attack Vector

An important problem in an aircraft communication security is to ensure that untrusted input from the VHF radio cannot influence the operation of systems responsible for flight controls. Unfortunately, the ACARS protocol does not itself provide any authentication, and proposals for adding authentication at the application layer [21] have not been adopted in practice. Without a means to authenticate an ACARS message, and thus ensure that it comes from a trusted ground system, ACARS messages should be treated as untrusted by aircraft systems acting on these messages. Indeed, several recent presentations at computer security conferences have raised concerns about the potential for abusing any misplaced trust in ACARS [19, 25, 28]. One of the analysis tasks envisioned for the Triton testbed is to evaluate the practical possibility of such attacks.

In the federated avionics models, evaluating whether an aircraft is vulnerable to ACARS-based attacks means determining whether there is a control path from the over-the-air ACARS input to a critical system, such as the FMC. Because these systems are not physically isolated, we cannot automatically rule out the possibility that such a path exists: for example, a malicious ACARS message might cause the CMU to send a message to the FMC that would cause it to issue commands to the autopilot without the pilots' knowledge. If such an attack is possible, we should be able to reproduce it in our testbed. On the other hand, if such an attack is impossible, we would like to show that this is the case through an analysis of the software running on these components. Even if it is impossible to completely rule out such an attack, being able to rule out certain classes of attacks would give us more confidence in the security of the aircraft system as a whole. Figure 1 shows that there are several potential paths for such an attack:

- **Direct:** VDR $\rightarrow$ CMU $\rightarrow$ FMC,
- **Via MCDU:** VDR $\rightarrow$ CMU $\rightarrow$ MCDU $\rightarrow$ FMC, or
- **Via ADL:** VDR $\rightarrow$ CMU $\rightarrow$ ADL $\rightarrow$ FMC.

A real Boeing 737 aircraft has additional LRUs that could act as an intermediate hop between CMU and FMC. We chose the set of LRUs illustrated in Figure 1 because these systems are connected by bidirectional ARINC 429 links that exchange multi-word messages using some variant of the bit-oriented protocol designed for messages larger than would fit in a 32-bit ARINC 429 word.

It should be noted that each attack path may involve real-time message forwarding along the path, or may require compromising intermediate systems to enable them to send arbitrary messages. For example, the CMU is usually configured to relay certain ACARS messages to the FMC, such as flight plans generated by the airlines dispatcher. In normal use, such flight plans must be explicitly accepted by a pilot before the FMC acts on it. We would like to rule out the possibility that an adversarially crafted message causes the FMC to act on a flight plan without pilot input. We would also like to consider the possibility that the CMU is compromised by a adversarially crafted message, allowing the attacker to then send arbitrary ARINC 429 messages to the FMC (rather than ARINC 429 messages encapsulating ACARS messages). In this case, we would like to determine whether such a compromise of the CMU is possible, and, if so, whether the FMC could be co-opted to command the autopilot system without pilot approval by an attacker with the ability to send arbitrary messages from the CMU to the FMC.

3.2 Data Loader Attack Vector

Virtually all LRUs with significant computing capability are built to be field-upgradable, or in the vernacular of the aviation industry, to accommodate Loadable Software Parts (LSPs) [3]. LSPs can include both the core software installed on the LRU, databases used by the LRU (e.g., the navigation database used by the FMC) or configuration data (e.g., reflecting per carrier customization via well-defined interfaces). ARINC defines a series of standards governing this process, with the 665 series [3] defining the file formats and the 615 series [4] defining the transfer protocol (the most common 615 protocol is a point-to-point protocol based on ARINC 429, while the more recent 615A protocols are IP-based and use TFTP for data transfer). Note that these protocols only define how to transfer data to an LRU and thus a great deal of semantic detail (e.g., how an LRU is directed to start running new code) is vendor specific. Lastly, the path by which ADLs interface with an LRU can vary as well. While many LRUs provide a standard 53-pin ARINC 615 connector for updating, it is also common for such LRUs to be directly wired into a physical selector switch that allows a maintenance technician to use a single such port to multiplex update access across LRUs. One common approach is for technicians to preload the necessary LSPs on a portable data loader such as the PMAT-2000 [27] (which is a ruggedized PC with an appropriate interface and cable) and physically connect to each LRU they need to update. In some modern aircraft, a library of LSPs may instead be staged on an onboard mass storage ADL, which is periodically refreshed via USB or wireless protocols (e.g., such as Teledyne's GroundLink).

Because Airborne Data Loaders can directly update both LRU software and support data, their security is critical to the overall security of an aircraft. This situation is exacerbated because digital signatures for LSPs are a relatively modern innovation (the key standards, ARINC 835 [10] and 842 [11] were only published in 2011 and 2012) and thus many LRUs are unable to detect if an update has been tampered with. We have configured our testbed to explore several aspects of this threat vector, including reverse engineering which LRUs are vulnerable to rogue updates, vulnerabilities in popular ADLs and threats associated with LSP staging outside the aircraft (historically, LSPs were shipped via floppy disk, but today most are delivered via Internet-based services).

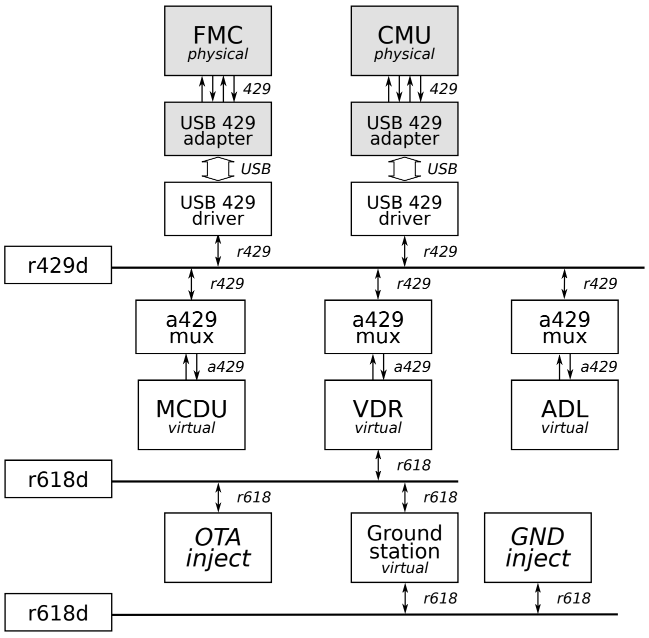

Figure 2: Portion of the Triton testbed.

4 Architecture

With the analysis tasks above in mind, we set out to design and build the Triton avionics testbed. Figure 2 shows the architecture of the testbed, described in more detail next.

4.1 Design Philosophy

Software-defined bus wiring. At the outset, we knew that we would be dealing with physical LRUs that we would want to connect to each other and to software-simulated components, and that we would want to reconfigure these arrangements for each experiment. To make this possible, we chose to *virtualize* the ARINC 429 bus, connecting the hardware LRUs to the virtual bus via adapters, rather than connecting simulated components to a physical ARINC 429 bus.

We modeled each physical bus as a broadcast medium for 32-bit ARINC 429 words. Both emulated LRUs and hardware LRUs (via an ARINC 429 adapter) could be connected to the same virtualized bus that, from the hardware LRUs point of view, would be logically indistinguishable from a physical interconnect. In Section 4.2, we describe our virtualized ARINC 429 bus and the underlying software components. When it came time to design the ACARS VHF communication medium to model the interception and injection of ACARS messages, the design philosophy of virtualizing physical media led to a similar design, where we modeled the ACARS data link as a broadcast medium to which software components could be connected as needed.

Unix philosophy. In designing the Triton testbed, we followed the Unix philosophy of creating simple programs that do one thing well and whose power lies in their composition. Applying this philosophy, we built each component of the testbed as a separate program that would communicate with others using TCP sockets and Unix pipes. Making Unix processes the basic unit of the system meant that components could be developed using any programming environment and language (e.g., C, Python) that supported TCP sockets. We discuss potential alternatives in Section 6.4.

The process-oriented model also made it easier to make the system hot-pluggable. While this was never an explicit requirement, the practical benefit was that it was easier to debug a specific component while the rest of the system continued to operate.

4.2 ARINC 429 Interconnect

Recall that the physical ARINC 429 bus connects a single transmitter to multiple receivers, allowing one device to transmit 32-bit ARINC 429 words to multiple listeners at once. A natural way to model this in software is using a publish-subscribe pattern, where each transmitter is a publisher and each receiver on the transmitter's bus a subscriber. The `r429d` daemon is a message broker that accepts TCP connections from other processes and routes ARINC 429 words between them. The `r429d` daemon multiplexes multiple virtual ARINC 429 busses over a single TCP connection; a connected process may be both a publisher and subscriber.

The R429 Protocol. We call the protocol spoken by the `r429d` daemon R429. Each R429 messages encapsulates a single 32-bit ARINC 429 word. Each virtual ARINC 429 bus is identified by an integer included in the R429 message, allowing the `r429d` daemon to route the ARINC 429 word to the correct set of receivers (subscribers). ARINC 429 adapters connect to `r429d` as publishers for their physical receiver channels and as subscribers for their physical transmit channels. Simulated devices (such as the VDR) connect to `r429d` as publishers for the ARINC 429 buses on which the LRU would normally be the transmitter, and as subscribers for the ARINC 429 buses on which the LRU would normally be the receiver.

The R429 protocol also includes provisions for timestamps on received ARINC 429 words and for scheduling ARINC 429 words for transmission at a precise time. This feature is intended for use with ARINC 429 adapters that support timestamps and fine-grained control over transmission time.

The A429 Protocol. In addition to the R429, which carries multiplexed ARINC 429 words, we also created A429, a simpler stream-oriented protocol that carries raw 32-bit ARINC 429 words without any encapsulation. The A429 protocol is intended to further simplify development of virtual ARINC 429 devices. In our current implementation, R429 is de-multiplexed into unidirectional A429 channels implemented as named Unix pipes. We can then connect a process to the virtual ARINC 429 bus by passing it the names of the named pipes on the command line. For example, the virtual MCDU command-line usage is

```
    $ a429_mcdu MAL out in
```

where *out* and *in* are the names of the input and output files (pipes), and *MAL* is the MCDU Address Label (an ARINC 429 protocol identifier used to distinguish multiple MCDUs).

Programs that use A429, such as `a429_mcdu` above, need an adapter to communicate using R429. This is implemented by `r429_piped` daemon (labeled `a429 mux` in Figure 2), which connects to the `r429d` daemon and demultiplexes the single R429 TCP connection into one or more unidirectional A429 named pipes. The `r429_piped` daemon handles processes repeatedly opening and closing the pipes as the client process is restarted.

Limitations. Triton replaces the physical ARINC 429 bus with message channels built on TCP and Unix pipes. Although this affords great flexibility, it does not reproduce the ARINC 429 medium perfectly. In particular, our virtualized ARINC 429 bus does not capture the electrical and timing characteristics of the signal, and thus Triton is not appropriate for experiments that require precise control of these properties. None of the experiments we consider require such fine control over timing and all of our devices are able to communicate correctly.

4.3 ACARS Medium

The ACARS communication medium is simulated by the R618 protocol and the `r618d` daemon. Similar to R429, the R618 is a TCP-based protocol that encapsulates and multiplexes ACARS communication channels, with `r618d` serving as the message broker. Devices that want to communicate using ACARS connect to the `r618d` daemon and monitor messages tagged with a specific frequency. There are two R618 domains, one simulating the air–ground link, and the other simulating the ground segment. The two are joined by an ACARS ground station, which takes care of downlinked message acknowledgement and uplinked message block number sequencing. ACARS messages injected into the air–ground R618 domain are received by the VDR as sent, while messages injected into the ground R618 domain are processed by the ground station and correctly numbered before being uplinked. Injecting messages into the air–ground domain models an attacker who can communicate directly with the aircraft without involving a real ground station. Injecting messages into the ground domain models an attacker who must deal with ground station contention.

4.4 Physical Components

Our research tasks (Section 3) involve analyzing the behavior of real avionics and so we connect several LRUs to our testbed, as shown at the top of Figure 2. Since our initial focus is on the Communication Management Unit (CMU) and the Flight Management Computer (FMC), these are the first physical LRUs we use. In particular, our testbed has several Rockwell–Collins CMU-900s, Honeywell Mark III CMUs, and Smiths FMCs from Boeing 737 aircraft. We purchased these LRUs on eBay or from aircraft parts dealers in "as removed" condition traceable to specific aircraft.

4.5 Simulated Components

In addition to the physical LRUs, our testbed includes several LRUs simulated in software. Each simulated LRU communicates using the A429 protocol described above.

Multi-Function Control and Display Unit (MCDU). The virtual MCDU communicates with a CMU using a protocol defined in ARINC characteristic 739 [2]. It simulates a 24 column by 14 row MCDU display and keyboard, allowing us to interact with a CMU using a computer terminal rather than a physical MCDU.

VHF Data Radio (VDR). The simulated VDR communicates with a CMU using the ARINC 429 Bit-Oriented Protocol. The RF medium is simulated using the R618 protocol and the `r618d` daemon (Section 4.3), which route ACARS messages to each device (virtually) tuned to a given frequency. Other devices, described below, can inject ACARS messages into this medium, modeling an attacker who can transmit ACARS messages in the appropriate VHF frequency. Our virtual VDR simulates messages received over both the original ACARS-over-AM and the newer ACARS over VHF Data Link Model 2.

Printer. The virtual printer communicates with a CMU using a protocol defined in ARINC characteristic 740 [1]. It supports printing ACARS messages received by the CMU. In particular, the ACARS protocol allows ACARS messages to be forwarded directly to the printer (without pilot interaction). This allows us to test whether the CMU provides any filtering of such ACARS messages or whether it forwards all such messages to the printer indiscriminately.

Flight Management Computer (FMC). In addition to a real FMC (Section 4.4), our testbed also has an implementation of a virtual FMC, which can communicate with a CMU as defined in ARINC characteristic 758 [9]. Our simulated FMC does not support all the features of a real FMC, such as a navigation database, but it can receive and acknowledge ACARS messages forwarded from a CMU. As with the printer, this allows us to test whether the CMU provides any filtering of such ACARS messages or whether it forwards all such messages to the FMC indiscriminately.

4.6 Emulated Components

Both the Honeywell and Rockwell–Collins CMU use an x86 main processor. We have extracted the firmware of both CMUs, allowing us to run the firmware in an x86 emulator such as QEMU. We have two modes of emulation, pure emulation and hybrid emulation. In the pure emulation mode, we emulate firmware using QEMU using custom QEMU machines and devices. In the hybrid emulation mode, the firmware is run in QEMU with I/O redirected to the real hardware using an approach similar to Surrogates [17].

The Rockwell–Collins CMU-900 uses an AMD AM486 processor as its main CPU. This feature-poor processor lacks the necessary debugging hardware for hybrid emulation. Instead, we reverse-engineered enough of its peripherals, including its ARINC 429 controller to implement our own versions as QEMU devices. The 429 connections are exposed as TCP sockets. Our current implementation is sufficient to support the firmware's data loading protocol which is handled by the main CPU. Other uses of 429 are handled by a dedicated I/O processor, an Intel 386EX; support for emulating the I/O processor is in progress.

We plan to support emulation of the Smiths FMC in future work.

5 Experiment Examples

The Triton testbed supports several kinds of experiments. Here, we describe two such experiments based on the analysis tasks outlined in Section 3.

5.1 ACARS Experiments

Our testbed allows us to inject arbitrary messages for processing by the CMU and FMC. In particular, sending an ACARS message over the virtual air-ground medium takes only a few lines of code. The message is then processed by the simulated VDR and delivered to the CMU, which may be the physical CMU or the CMU code running in QEMU. The latter allows us to snapshot and examine the state of memory as the message is processed, as well as to add more sophisticated analysis tasks such as taint-tracking the message data. Delivering the message to hardware provides the highest degree of execution fidelity and is useful to confirm whether behaviors observed in emulation are an artifact of emulation or not. During message processing, we can interact with the CMU using the virtual MCDU to observe how it responds to various messages and to test whether certain messages trigger a notification or not.

The CMU may also be connected to an FMC, which may be simulated or physical. With a simulated FMC, we can test which ACARS messages are forwarded by the CMU to the FMC. To observe how an FMC would react to these messages, we can connect the physical FMC to the virtual bus and forward ARINC 429 traffic from physical CMU to physical FMC over the virtual ARINC 429 links.

The most complex testbed configuration realized in the lab so far consisted of the Honeywell Mark III CMU code running in QEMU (as described in Section 4.6) using the physical ARINC 429 interfaces of the CMU to communicate with a simulated MCDU, VDR, and cockpit printer. The VDR is also connected to a simulated ACARS ground station, allowing us to send ACARS messages to the CMU and observe its behavior in QEMU.

5.2 Data Loader Experiments

We implemented the low-level, generic ARINC 615 [4] data loading protocol as a Python module which we can use to upload data to LRUs. Using this capability as a building block, it is easy to implement the higher-level, LRU-specific protocols. We reverse engineered and implemented the high-level data loading protocol for one of our CMUs, the Rockwell–Collins

CMU-900. In the lab, we can configure our testbed to connect our simulated data loader to either an emulated or physical CMU and upload data.

We are expanding this capability to other LRUs and investigating both the ACARS attack vector which leverages the ADL (Section 3.1) and the data loader attack vector (Section 3.2).

6 Discussion

We believe that the Triton testbed meets the explicit and implicit requirements for which it was designed. In this section, we start by describing some of the challenges associated with working with avionics in the lab. Next we discuss two lessons learned from our early testbed design. We end with a discussion of alternative designs.

6.1 Challenges Working with Avionics

Integrating physical avionics components with simulated and emulated components as described in Section 4 is complicated by several factors, including boutique power requirements, ARINC 429 networking, and unusual system design choices made by the avionics manufacturers.

Many avionics, including our CMUs and FMCs, are powered by a 115 V, 400 Hz alternating-current power supply. While some avionics also support the standard US residential 120 V, 60 Hz AC for ground-based testing, others do not.[3] This choice of frequency complicates running the LRUs on a lab bench by requiring either an expensive power supply or hardware modifications to allow 60 Hz power. We chose the former approach.

In contrast to other common vehicular networks like the automotive CAN bus, which connects multiple electronic control units (the automotive equivalent of an LRU) together using a single shared bus which is frequently exposed via the standard OBD-II port, ARINC 429 requires bi-directional links between communicating LRUs. For example, a CMU that follows ARINC 758 has forty-eight 429 inputs and twelve 429 outputs [9, Attachment 1–6]. Wiring just the desired 429 connections to the appropriate pins in the 300-pin connector is a delicate task.

The design of our Rockwell-Collins CMU-900 complicates its analysis. The CMU-900 is itself a system-of-systems with three heterogeneous processors: The main processor is an AMD Am486 and its I/O processor is an Intel 386ex. The firmware for both make extensive use of x86's protected mode segments which are not well-supported by most debugging tools. Other, less problematic, but similarly custom design choices include using RS-232 serial ports in nonstandard configurations.

6.2 Lessons Learned: Two Early Lessons

Following the Unix philosophy (Section 4.1), we designed the Triton testbed around the concept of a process as the basic component of the system. ARINC 429 adapter drivers, simulated components, and tools are all processes that communicate via a shared medium simulated by the `r429d` and `r618d` daemons. As noted, this process-oriented design keeps the system program-language agnostic, allowing components to be developed using any language or programming environment that supports TCP sockets.

One early failure to adhere to this philosophy was the decision to make the ARINC 429 adapter part of the `r429d` daemon. On startup, the `r429d` daemon read a configuration file that specified which adapter driver to use and how ARINC 429 channels identified by the driver would map to ARINC 429 channels presented by `r429d`. The `r429d` daemon would load the adapter driver, actually a child process that would communicate with `r429d` using pipes, at startup. At the time we made this decision, we imagined that most of the debugging and troubleshooting would be in the simulated components; this proved not to be the case. We built the first-generation ARINC 429 adapter ourselves, which involved considerable debugging of both the driver and the firmware. We never managed to make it completely reliable, necessitating frequent adapter power cycling and driver restarts. To restart the driver, we needed to restart the `r429d` process, which tore down the virtual 429 bus between physical and simulated components. Each time we restarted the `r429d`, we needed to re-attach the R429 monitoring tool, virtual MCDU, and virtual VDR to the bus.

We eventually changed the model to one where the adapter driver is a separate process that attaches to the `r429d` daemon independently of other components. This greatly simplifies the `r429d` daemon, which no longer needs a configuration or the ability to spawn a driver process. It also simplifies adapter-driver testing and debugging, because it does not require restarting the `r429d` daemon.

We also eventually abandoned our own ARINC 429 adapter hardware completely and opted for a commercial product, which proved to be much more reliable in practice. While the process of building our own ARINC 429 adapter was educational, that decision cost us many hours that could have been better spent on security analysis.

6.3 Two Virtual ARINC 429 Protocols

Recall that our testbed has two protocols used to virtualize ARINC 429 buses: R429 and A429. The R429 protocol carries multiplexed ARINC 429 words between devices attached to a virtual ARINC 429 bus and the message broker daemon, `r429d`, which implements the ARINC 429 one-transmitter, multiple-receivers topology as a publish-subscribe scheme. To simplify application interfaces, we also created the A429 protocol, which carries raw 32-bit ARINC 429 words. This requires running at least one (often more than one) instance of the `r429_piped` daemon to bridge between R429 and A429.

While the intent was to simplify application design, having two protocols communicating via the `r429_piped` daemon acting as a software adapter actually adds additional steps. At-

[3]The authors received a first-hand lesson in what happens when a too-low frequency AC current is applied to avionics of the second type, to wit a fried transformer.

taching a device such as an MCDU to the virtual ARINC 429 bus should have been a simple process. Instead, the user first needs to make sure the correct named pipes exist in the file system, and, if not, create them using the `mkfifo` command. Next, the user has to run an instance of the `r429_piped` daemon, specifying on the command line the named pipes and to which virtual R429 channels they map. Only then can the user run the virtual device, providing it the names of the pipes on the command line. Combined with the physical ARINC 429 adapter and driver problems (Section 6.2) that required restarting the `r429d` daemon, this design led to a lot of open terminal windows and process restarts, needlessly increasing workflow complexity.

In retrospect, A429 was not necessary and adds additional complexity to our workflow. The intended benefit of A429, namely simplifying the application interface to the virtualized ARINC 429 bus, could be provided by library functions that connect to `r429d` using the R429 protocol.

6.4 Alternate Designs

It is worth considering alternatives to the architecture of the Triton testbed. For example, the entire testbed could have been built following the GNU Radio model: a monolithic process where components such as simulated LRUs are modules connected together by intra-process queues. A configuration file, or even straight-line initialization code, could assemble an experiment using components defined as C++ classes with inputs and outputs wired together at initialization. Such close integration of components is arguably necessary for GNU Radio to achieve the signal processing throughput needed to implement a software-defined radio application. However, the bandwidth requirements for ARINC 429 are much more modest—high speed links signal at 100 kHz—and well within the bandwidth of a loopback TCP link.

The process model also allows components to be hot pluggable: simulated components, diagnostic tools, and injection tools can be attached and detached from the system as necessary without stopping the experiment. A monolithic design model would have required considerable engineering to support such a feature—one we probably would not have implemented. Finally, the monolithic design would not offer the same flexibility in the choice of programming language used to develop each component. We find this flexibility to be useful in practice: Although we wrote the majority of the Triton testbed in C, we implemented our simulated Airborne Data Loader (ADL) in Python.

7 Related Work

While little of the work related to empirical aviation cybersecurity is public, there is an emerging open literature both from the academic and independent security research communities. Most of this has focused on threats and vulnerabilities associated with particular aviation communications channels including ACARS [16,26], ADS-B [15], and Satellite [22–24] channels. In general, these efforts have focused on individual protocols or receivers in isolation—exploring design and implementation vulnerabilities and the potential for effects through that channel alone. An exception to this is Hugh Teso's 2013 Hack In The Box talk which discussed the possibility of lateral movement from one avionics component to another. While Teso's claims are controversial (and widely disputed) his presentation describes a software testbed purporting to run emulated code from different avionics components [28]. However, a criticism of Teso's testbed is that as it is entirely based on software and lacks real avionics components, it has limited fidelity for describing the behavior of real aircraft. By contrast, the Department of Homeland Security recently acquired an operational Boeing 757 to explore end-to-end security issues as part of their Aviation Cybersecurity Initiative (ACI) program [12]. This approach has the benefit of extremely high fidelity but is expensive to procure, operate, and maintain.[4] We are motivated by the same kinds of system-wide security questions, but the combination of our goals and means has led to a hybrid testbed that combines both LRUs from real aircraft (physical components), emulated avionics software, and simulated components.

Outside security, we are aware of multiple industry efforts to create avionics testbeds in support of development and test for commercial aircraft. Indeed, we believe that such testbeds have been created internally by a range of airframe and avionics manufacturers. Publicly described examples of such testbeds include Eurocontrol's Link 2000+ testbed [13] and the Air Force's Reconfigurable Cockpit and Avionics Testbed (RCAT) operated by MITRE [20, 29]. The purpose of these testbeds is to test the integration of new or modified components in a realistic environment. Unlike the DHS approach, these involve only a small subset of key components; however, they are similar in (typically) using only real avionics equipment and no software emulation or simulation.

Finally, there are a broad array of emulation testbeds that have supported security research. Among the best known are Utah's Emulab [30] which allow experiments over hundreds of machines in a contained environment. This architecture was expanded to include complex topologies and routing combinations of emulated and real network equipment in the DETER testbed [18] and expanded yet more in DARPA's National Cyber Range program [14]. Our work is both smaller scale and significantly more bespoke, but borrows from the hybrid nature of these later testbeds—combining real and emulated components in a single environment as needed.

8 Conclusion

While its design continues to evolve as we acquire additional physical components, the Triton testbed has allowed us to operate a number of critical avionics components, namely the CMU, FMU and MCDU, for several years in their as-installed configurations without the need for an actual Boeing 737

[4] Note that some of our authors participated in the DHS ACI program, but the work described in this paper is entirely separate from that effort.

airframe. Several of these components will not even boot without interrogating the ARINC 429 bus for the (apparent) presence of various additional pieces of equipment that the Triton design allows us to faithfully simulate—or at least stub out with sufficient fidelity such that the components under test proceed without error.

Our simulated ARINC 429 bus greatly simplifies inspecting and interposing on inter-component messaging, thereby facilitating security analyses that attempt to expose any potential stepping-stone attacks. One key challenge that remains to be addressed by our cross-mode design, however, is systematically identifying and addressing tight timing constraints that components occasionally place on inter-component messaging. Such constraints are especially challenging to deal with in emulated components which may not be as performant or deterministic as their physical counterparts.

Even in its current state, however, Triton enables us to conduct not only the two exemplar analyses discussed in this paper, but several additional, ongoing studies that will dramatically increase our understanding—and the security—of the complex interdependence between the federated avionics that underpin much of the world's commercial air transport fleet.

9 Acknowledgements

We are grateful to the anonymous reviewers for their many helpful comments and suggestions.

This work was supported by the National Science Foundation grants NSF-1901728 and NSF-1646493 and by generous research, operational, and/or in-kind support from the UCSD Center for Networked Systems (CNS).

References

[1] ARINC. *Multiple-Input Cockpit Printer*. Aeronautical Radio, Inc., June 1988. ARINC Characteristic 740-1.

[2] ARINC. *Multi-Purpose Control and Display Unit*. Aeronautical Radio, Inc., Dec. 1998. ARINC Characteristic 739A-1.

[3] ARINC. *Loadable Software Standards*. Aeronautical Radio, Inc., Jan. 2001. ARINC Report 665.

[4] ARINC. *Airborne Computer High Speed Data Loader*. Aeronautical Radio, Inc., May 2002. ARINC Report 615-4.

[5] ARINC. *Mark 33 Digital Information Transfer System (DITS) Part 1 Functional Description, Electrical Interface, Label Assignments and Word Formats*. Aeronautical Radio, Inc., May 2004. ARINC Specification 429 Part 1-17.

[6] ARINC. *VHF Data Radio*. Aeronautical Radio, Inc., Aug. 2004. ARINC Characteristic 750-4.

[7] ARINC. *Air/Ground Character-Oriented Protocol Specification*. Aeronautical Radio, Inc., June 2006. ARINC Report 618-6.

[8] ARINC. *ACARS Protocols for Avionc End Systems*. Aeronautical Radio, Inc., June 2009. ARINC Specification 619-3.

[9] ARINC. *Communications Management Unit (CMU) Mark 2*. Aeronautical Radio, Inc., Jan. 2011. ARINC Characteristic 758-3.

[10] ARINC. *Guidance for Security of Loadable Software Parts Using Digital Signatures*. Aeronautical Radio, Inc., Nov. 2011. ARINC Characteristic 835.

[11] ARINC. *Guidance for Usage of Digital Certificates*. Aeronautical Radio, Inc., June 2012. ARINC Characteristic 842.

[12] C. Biesecker. Boeing 757 testing shows airplanes vulnerable to hacking, DHS says. *Avionics International*, Nov. 2017.

[13] Eurocontrol LINK 2000+ Programme. The LINK2000+ test facility presentation. Oct. 2004.

[14] B. Ferguson, A. Tall, and D. Olsen. National Cyber Range overview. In *Proceedings of the IEEE Military Communications Conference*, Oct. 2014.

[15] B. Haines. Hackers + airplanes: No good can come of this. Presented at DEFCON 20, 2012.

[16] D. Hoffman and S. Rezchikov. Busting the BARR: Tracking "untrackable" private aircraft for fun & profit. Presented at DEFCON 20, 2012.

[17] K. Koscher, T. Kohno, and D. Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2015.

[18] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. T. Wroclawski, and S. Schwab. The DETER Project: Advancing the science of cyber security experimentation and test. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security 2010*, Nov. 2010.

[19] P. Polstra and C. Polly. Cyberhijacking Airplanes: Truth or Fiction? Presented at DEFCON 22, Aug. 2014.

[20] C. Risley, J. McMath, and B. Payne. Experimental encryption of aircraft communications addressing and reporting system (ACARS) aeronautical operational control (AOC) messages. In *Proceedings of Digital Avionics Systems Conference*, Oct. 2001.

[21] A. Roy. Secure Aircraft Communications Addressing and Reporting System (ACARS). In *Proceedings of the Digital Avionics Systems Conference*, 2001.

[22] R. Santamarta. SATCOM terminals hacking: By air, sea and land. Presented at Blackhat USA, 2014.

[23] R. Santamarta. A wake-up call for SATCOM security. 2014.

[24] R. Santamarta. Last call for SATCOM security. Presented at Blackhat USA, 2018.

[25] M. Smith, D. Moser, M. Strohmeier, V. Lenders, and I. Martinovic. Undermining Privacy in the Aircraft Communications Addressing and Reporting System (ACARS). *Proceedings of the Privacy Enhancing Technologies Symposium*, 2018(3):105–122, 2018.

[26] M. Smith, D. Moser, M. Strohmeier, V. Lenders, and I. Martinovic. Undermining privacy in the aircraftcommunications addressing and reportingsystem (ACARS). *Proceedings on Privacy Enhancing Technologies*, (3):105–122, 2018.

[27] Teledyne Controls. PMAT 2000 System: Portable Maintenance Access Terminal 2000, Nov. 2017.

[28] H. Teso. Aircraft hacking: Practical aero series. Presented at Hack In The Box Security Conference, Apr. 2013.

[29] D. Van Cleave. RCAT: Tool to achieve GATM. *Avionics Magazine*, pages 30–32, Sept. 2003.

[30] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 255–270. USENIX Association, Dec. 2002.