UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Security Testing Tools for Complex Cyber-Physical Systems

Permalink
https://escholarship.org/uc/item/51r2593g

Author
Crow, Sam

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Security Testing Tools for Complex Cyber-Physical Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Sam Crow

Committee in charge:

Professor Aaron Schulman, Chair
Professor Dinesh Bharadia
Professor Sujit Dey
Professor Rajesh Kumar Gupta
Professor Geoffrey M. Voelker

2022

Copyright
Sam Crow, 2022
All rights reserved.

The dissertation of Sam Crow is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

TABLE OF CONTENTS

Dissertation Approval Page iii

Table of Contents iv

List of Figures vi

Acknowledgements vii

Vita ix

Abstract of the Dissertation x

Chapter 1 Introduction 1
1.1 Testing challenges 2
1.2 Thesis statement 3
1.3 Dissertation organization 3

Chapter 2 Background 5
2.1 System example 5
2.2 Unit and system testing 6
2.3 Wireless monitoring 6

Chapter 3 Flexible Avionics Testing 8
3.1 Introduction 8
3.2 Background 10
3.2.1 Federated avionics architecture 10
3.2.2 ARINC 429 bus 12
3.2.3 ACARS 13
3.3 Targeted analysis tasks 14
3.3.1 ACARS attack vector 14
3.3.2 Data loader attack vector 16
3.4 Architecture 17
3.4.1 Design philosophy 17
3.4.2 ARINC 429 interconnect 19
3.4.3 ACARS medium 21
3.4.4 Physical components 22
3.4.5 Simulated components 22
3.4.6 Emulated components 23
3.5 Experiment examples 24
3.5.1 ACARS experiments 24
3.5.2 Data loader experiments 25
3.6 Discussion 25

3.6.1 Challenges working with avionics 25
3.6.2 Lessons learned: two early lessons 26
3.6.3 Two virtual ARINC 429 protocols 28
3.6.4 Alternate designs 29
3.7 Related work 29
3.8 Conclusion 31

Chapter 4 Bus Driver: No-cut Message Modification on Aviation Data Buses 33
4.1 Introduction 33
4.2 Threat model 36
4.2.1 Constraints and opportunity 36
4.2.2 Bus Driver: A no-cut interception attack 38
4.2.3 Challenges 40
4.3 Feasibility of Bus Driver on aviation buses 41
4.3.1 ARINC 429 42
4.3.2 Proof-of-concept ARINC 429 implant 44
4.3.3 MIL-STD-1553 48
4.3.4 Proof-of-concept MIL-STD-1553 implant 50
4.4 Case study 55
4.4.1 Boeing 737 systems 55
4.4.2 Test setup 56
4.4.3 Implant capabilities 58
4.5 Defenses 63
4.6 Related work 65
4.7 Ethics 67
4.8 Conclusion 68

Chapter 5 Implementing High-Speed Scanning for Wireless Devices 70
5.1 Introduction 70
5.2 Bluetooth scanning 71
5.2.1 Multi-channel Bluetooth scanning 71
5.3 Hardware limitations 72
5.4 Receiving using SparSDR 73
5.5 Scanner implementation 74
5.5.1 Timing Challenges 74
5.5.2 Performance Challenges 76
5.6 Evaluation 78
5.7 Conclusion 79

Chapter 6 Conclusion and Future Directions 81

Bibliography 83

LIST OF FIGURES

Figure 3.1: Fragment of Boeing 737 avionics interconnection centered at the Communication Management Unit (CMU). 12
Figure 3.2: Portion of the Triton testbed. 18

Figure 4.1: Wire loom from a Boeing 737. It is difficult to find and cut specific wires in an unlabeled bundle of cables. 37
Figure 4.2: Unused data bus ports on the Boeing 737 present an opportunity for an attacker to hide an implant inside of the port. 38
Figure 4.3: A conventional attacker-in-the-middle attack splits a bus into two parts to intercept and modify messages. A Bus Driver implant just needs to connect to one point to intercept data and change it. 39
Figure 4.4: (a) An ARINC 429 bus with an accessible connector, (b) An implant changing the voltage on the bus 43
Figure 4.5: A simplified block diagram of the 429 implant 45
Figure 4.6: The ARINC 429 bus waveform during a legitimate transmission (blue) is nearly indistinguishable from the Bus Driver attack (red). 48
Figure 4.7: A MIL-STD-1553 bus's transformer-coupled stubs offers protection against a Bus Driver implant. 49
Figure 4.8: A block diagram of a MIL-STD-1553 Bus Driver attack which requires two connections to the bus. 51
Figure 4.9: A legitimate MIL-STD-1553 waveform compared to the Bus Driver's attempt to cancel the waveform. 54
Figure 4.10: FMC, MCDU, and implant communication modes 57
Figure 4.11: Our test setup with the main components labeled 59
Figure 4.12: The MCDU screen showing a normal page from the FMC (top) and with some text modified by an implant (bottom) 61

Figure 5.1: The messages sent and received by a conventional scanner and a multichannel scanner over the same duration 72
Figure 5.2: A timeline of the transmit and receive periods within a 10-millisecond scanning cycle 75
Figure 5.3: A timeline of what can go wrong if the sample interrupts are not correctly aligned 76
Figure 5.4: The total CPU and memory usage of all the scanning software with different numbers of discoverable devices 79

ACKNOWLEDGEMENTS

First and foremost I am grateful to my advisor, Aaron Schulman, for his encouragement, support, and creative advice for solving technical problems. I appreciate all the work he did to keep my hardware projects going through the pandemic even when I couldn't go to the lab.

It has been a privilege to work with Kirill Levchenko, Stephen Checkoway, Stefan Savage, and Alex Snoeren in the Aerosec group. I knew I had come to the right place when Kirill showed me some actual airplane parts I could work with, and with their help this turned into an exciting set of projects.

I 'd like to thank Dinesh Bharadia for always encouraging me to improve my work; Geoff Voelker for his support and organizing some spectacular hikes; and my other committee members, Sujit Dey and Rajesh Gupta, for their guidance.

Jennifer Folkestad has been invaluable for her efficient and helpful administrative support, negotiating with airplane parts dealers and placing plenty of urgent orders for electronic components.

I am greatful for the opportunity to collaborate with Moein Khazraee, who was always generous with his time and advice on FPGAs and signal processing.

It has been great to work with Nishant Bhaskar and Raghav Subbaraman, who I could always depend on to keep track of what works and solve problems systematically, even when we faced bewildering technical challenges.

Reprinted chapters Chapter 3, in full, is a reprint of the material as it appears in Sam Crow, Brown Farinholt, Brian Johannesmeyer, Karl Koscher, Stephen Checkoway, Stefan Savage, Aaron Schulman, Alex C Snoeren, and Kirill Levchenko. Triton: A software-reconfigurable federated avionics testbed. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019. The dissertation author was the primary investigator and author of this paper.

This chapter 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).

Chapter 4, in full, is a reprint of material under submission to USENIX Security 2023: Sam Crow, Stephen Checkoway, Patrick Mercier, Pat Pannuto, Stefan Savage, and Aaron Schulman. Bus driver: No-cut attacker-in-the-middle capability on aviation data buses. The dissertation author was the primary investigator and author of this paper.

Chapter 5 was supported by a grant from Amateur Radio Digital Communications.

VITA

2017	B. Sc. in Computer Engineering *cum laude*, University of Washington, Seattle
2022	Ph. D. in Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Sam Crow, Brown Farinholt, Brian Johannesmeyer, Karl Koscher, Stephen Checkoway, Stefan Savage, Aaron Schulman, Alex C Snoeren, and Kirill Levchenko, "Triton: A Software-reconfigurable Federated Avionics Testbed", *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.

Moein Khazraee, Yeswanth Guddeti, Sam Crow, Alex C Snoeren, Kirill Levchenko, Dinesh Bharadia, and Aaron Schulman, "SparSDR: Sparsity-proportional Backhaul and Compute for SDRs", *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019

ABSTRACT OF THE DISSERTATION

Security Testing Tools for Complex Cyber-Physical Systems

by

Sam Crow

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2022

Professor Aaron Schulman, Chair

The modern world depends on the safe operation of infrastructure and vehicles for transport, communication, and power distribution. All these systems are increasingly computerized and interconnected. Connecting devices together into large systems creates new security vulnerabilities that conventional tests on single devices cannot find. To help find these vulnerabilities, we developed new tools to make testing complex systems easier. First, the Triton testbed provides a flexible way to test many interconnected devices that can include physical devices on real hardware, emulated devices running real software, and simulated devices that replicate the expected behavior of a physical device. Second, a Bus Driver device can modify messages on a wired communication bus to see how devices respond to modified messages. Finally, for part of

the HyperScanner project, we adapted a low-cost software-defined radio to scan for Bluetooth devices more quickly by sending and receiving on many channels at the same time. The same approach could also be used to monitor wireless communication on many different protocols to detect problems with wireless security. These tools will enable more effective security testing to make complex cyber-physical systems safer and more reliable.

Chapter 1

Introduction

The modern world depends on the safe operation of infrastructure and vehicles for transport, communication, and power distribution. All these systems are increasingly computerized and interconnected using many different forms of wired and wireless communication. This can make them more efficient and safer, but also exposes them to new kinds of attacks that can have disastrous real-world consequences.

The first widely known example of an electronic attack with physical impacts was the Stuxnet worm, which infected industrial control systems and damaged Uranium enrichment centrifuges [12]. A smaller-scale attack derailed trams by wirelessly controlling track switches [17]. One researcher claimed to have accessed an airplane's onboard computers through its in-flight entertainment system and changed its flight path [13]. Although there is no evidence that this attack actually succeeded, the design of modern airplane systems makes a similar attack possible. Multiple groups have found remotely exploitable vulnerabilities in cars that let attackers control all their onboard control units [18, 58].

To protect these cyber-physical systems from attacks, we need to design effective security features, implement the features correctly, and test to confirm that they work. The easiest form of security testing looks at a single device. This is partially effective and can find some security

problems. However, new problems can appear when combining devices into larger systems. Testing a complete system with many interconnected computers and many different forms of communication is challenging and sometimes impractical with conventional methods.

Simulation is a cost-effective way to test a complete system, but most simulations gloss over some behavior that can be important for security. Sometimes a simulated device has the same functionality as a real device but is not running exactly the same software. If the real device software has a buffer overflow or similar vulnerability, the simulated device may not have the same problem. The solution is to emulate every device using the same software that real devices use. This makes the simulation more accurate, but it still does not account for hardware details. The processors that the software runs on might have hardware bugs or side channels that make specific hardware vulnerable. Also, the communication interfaces on some devices could have electrical properties that make it possible to interfere with communication in unexpected ways. Overall, because attacks take advantage of behavior that the system designers could not foresee, most simulations make systems look more secure than they really are. We need to test on real hardware to find more vulnerabilities.

1.1 Testing challenges

This dissertation focuses on three specific testing challenges:

System testing An organization developing a system usually builds a test setup to check that the complete system works correctly. The details of how these test setups work are not public, and outside researchers rarely get to run their own tests. This makes it difficult to independently find system-level security problems.

Message modification testing An attacker with physical access to part of a system can interfere with wired communication between devices. For a security testing program to be

comprehensive, it should be able to check how devices respond to this kind of attack.

Wireless monitoring and scanning Some systems use wireless communication, which can involve several different protocols and a wide range of frequencies. A comprehensive security testing setup needs to monitor all relevant frequencies all the time to find unexpected or incorrect messages. This requires either a separate radio dedicated to each frequency, or a wideband software-defined radio and a powerful computer to decode the received signals. Both of these conventional options are expensive and inconvenient.

1.2 Thesis statement

In this dissertation, I demonstrate that new tools can effectively test the security of communication within cyber-physical systems, without needing a complete system of real hardware and without knowing exactly how every device should behave. With easier testing, anyone developing cyber-physical systems will be able to improve the security of those systems, making them safer and more reliable.

1.3 Dissertation organization

The main chapters of this dissertation cover three testing tools for different aspects of cyber-physical systems.

Chapter 3: Flexible Avionics Testing. This chapter describes a test setup for avionics that can connect real hardware to emulated or simulated parts. This is important for testing an entire system of devices to find security problems that single-device tests cannot find. A complete system of avionics usually includes several computers, screens, radios, sensors, and actuators that work together. If it is not practical to assemble a complete test setup with all real hardware, this test setup can still perform useful tests with some devices emulated or simulated.

Chapter 4: Bus Driver: No-cut Message Modification on Aviation Data Buses. This chapter describes how to connect to a wired communication bus and block or modify messages that other devices send. Although I demonstrate a device that performs an attack, the same approach can help test devices to see how they respond to modified messages. The conventional approach requires disconnecting the bus wires and installing test devices that receive messages on one part of the bus and send modified messages to another part. This new approach allows a single test device to modify messages from any other connected device without being moved around or rewired.

Chapter 5: Implementing High-Speed Scanning for Wireless Devices. This chapter describes a contribution to a larger project to make a low-cost, high-speed Bluetooth scanner. Because the project uses a software-defined radio, it is not limited to Bluetooth and could support many other wireless protocols. The scanner's ability to monitor all messages across many channels will help with testing devices under development to check that all their wireless communication works correctly. For security testing of existing systems, it can be used to detect malicious devices and devices that are legitimate but configured incorrectly.

Chapter 2

Background

2.1 System example

As an example of a cyber-physical system that uses many forms of wired and wireless communication, consider a Boeing 787 airliner. Most of the computing is concentrated in two Common Computing Resource (CCR) cabinets, which run several independent applications on the same hardware [50, 60]. The CCR cabinets communicate with the sensors and actuators mainly using Avionics Full-Duplex Switched Ethernet (ARINC 664). There are also 21 Remote Data Concentrators (RDCs) that bridge between the main Ethernet network and other devices that use ARINC 429 or CAN bus. Some important functionality uses wireless communcation. For example, the cabin emergency lights use a Bluetooth mesh network to send control signals [52, 51]. The core computer systems also connect to the in-flight entertainment system to send details about the plane's location and altitude. The plane communicates wirelessly with ground stations using Aircraft Communications Addressing and Reporting System (ACARS) and Gatelink (over Wi-Fi). That system covers just one vehicle, but from publicly available information it includes at least 23 boxes of avionics hardware using at least three different wired communication buses and three wireless protocols.

2.2 Unit and system testing

The simplest form of security testing looks at one device at a time. It is reasonably easy to test all the inputs and outputs of one device for a few different operating modes so we can convince ourselves that it is secure in isolation. This usually requires making some assumptions, such as that an attacker will not be able to send signals to the device because some other part of the system protects it.

In a complete system, however, some of those assumptions turn out to be wrong. The isolation boundaries between different parts of the system might not be perfectly reliable. With multiple applications running on the same hardware, side channels might leak important information. If legitimate messages get redirected to a different device that was not expecting them, they could cause unexpected behavior. Several proposed attacks on the Boeing 787 start by compromising an external communication interface to get access to the onboard network, and then move on to other devices that were supposed to be protected from outside attackers [50]. The functional testing of the 787 avionics used one stage of testing each function in isolation, and a second stage to check that all the functions work correctly when combined [60]. By the same logic, comprehensive security testing needs to involve more than one isolated device.

Some security vulnerabilities can be definitively demonstrated or disproven only by testing the complete system. Boeing can replicate the electronics of a Boeing 787 with all the sensors and actuators in a lab and do basic testing. For independent security researchers, who are likely to spend more time looking for vulnerabilities, the cost of buying all the parts is prohibitive.

2.3 Wireless monitoring

A separate problem is monitoring wireless communication on multiple protocols and multiple channels. The 787 uses both Wi-Fi and Bluetooth [50], which share 100 MHz of spectrum in the 2.4 GHz industrial, scientific, and medical band. Different Wi-Fi networks can

use any of about 11 different channels [34], and a conventional radio can monitor only one channel at a time. Bluetooth uses frequency hopping, so a pair of connected devices uses a different channel for each message [16]. A conventional Bluetooth radio can monitor only one channel at a time, so it can monitor messages between one pair of devices only.

To monitor everything that all nearby devices are sending, we need to receive on every channel all the time. One way to do this is to run a separate conventional radio for every channel and collect the messages that they decode. With 79 Bluetooth channels and about 11 Wi-Fi channels, this requires a lot of hardware.

A common alternative uses a wideband Software-Defined Radio (SDR) that can monitor all channels at the same time. The SDR captures radio signals and converts them into digital samples. To convert the samples into useful information, software decoders need to process all the samples, separate them into channels, and extract the content of each message. The decoding process becomes very processor-intensive with many channels because every decoder needs to run all the time, even when there are no useful signals. An SDR and a computer powerful enough to decode all the signals cost thousands of dollars each. Chapter 5 describes a way to monitor many channels using lower-cost hardware.

Chapter 3

Flexible Avionics Testing

3.1 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 chapter 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

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

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.

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 chapter 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 chapter is organized as follows. Section 3.2 presents technical background, including specifics of aircraft systems, necessary for the rest of the chapter. Section 3.3 describes the two analysis tasks in more detail. Section 3.4 then described the design of the Triton testbed. Section 3.5 briefly describes several experiments enabled by the testbed. Section 3.6 discusses our experience with our testbed. Section 3.8 concludes the chapter.

3.2 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.

3.2.1 Federated avionics architecture

In a federated avionics architecture, system functionality is implemented in discrete Line-Replaceable Units (LRUs). Figure 3.1 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 3.2.3. 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 3.2.2

[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.

describes these in more detail.

We briefly describe the high-level functionality of each LRU in Figure 3.1.

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.

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

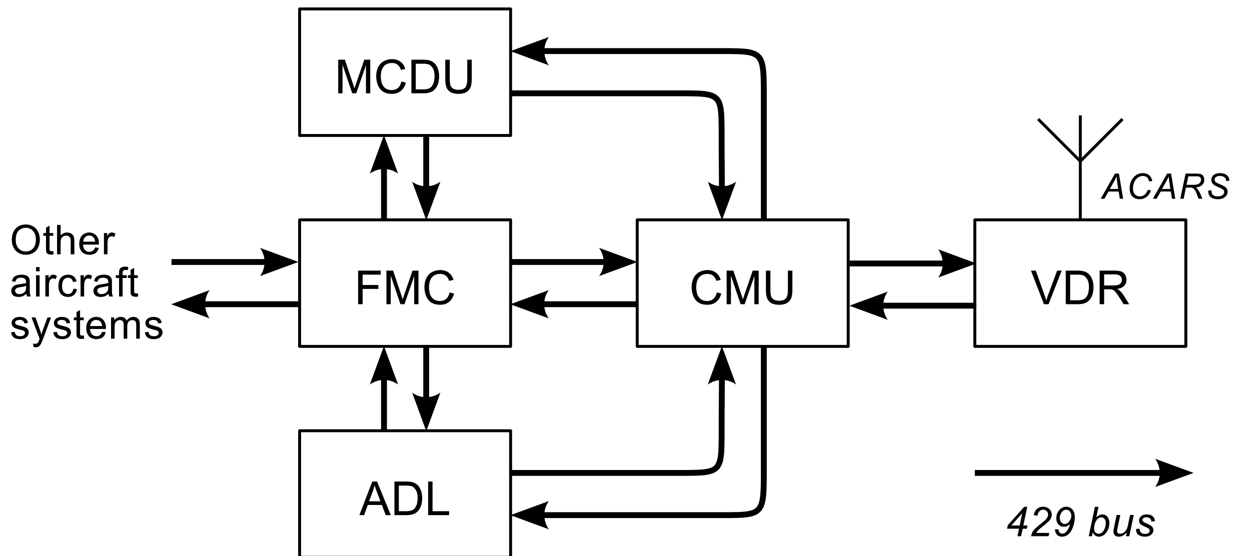

Figure 3.1: Fragment of Boeing 737 avionics interconnection centered at the Communication Management Unit (CMU).

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.

3.2.2 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 3.1.

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 3.2.3) 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.

3.2.3 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 3.3.1.

3.3 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.

3.3.1 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 [46] 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 [56, 44, 53]. 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 3.1 shows that there are several potential paths for such an attack:

- **Direct:** VDR → CMU → FMC,
- **Via MCDU:** VDR → CMU → MCDU → FMC, or
- **Via ADL:** VDR → CMU → ADL → 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 3.1 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.

3.3.2 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 [55] (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).

3.4 Architecture

With the analysis tasks above in mind, we set out to design and build the Triton avionics testbed. Figure 3.2 shows the architecture of the testbed, described in more detail next.

3.4.1 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

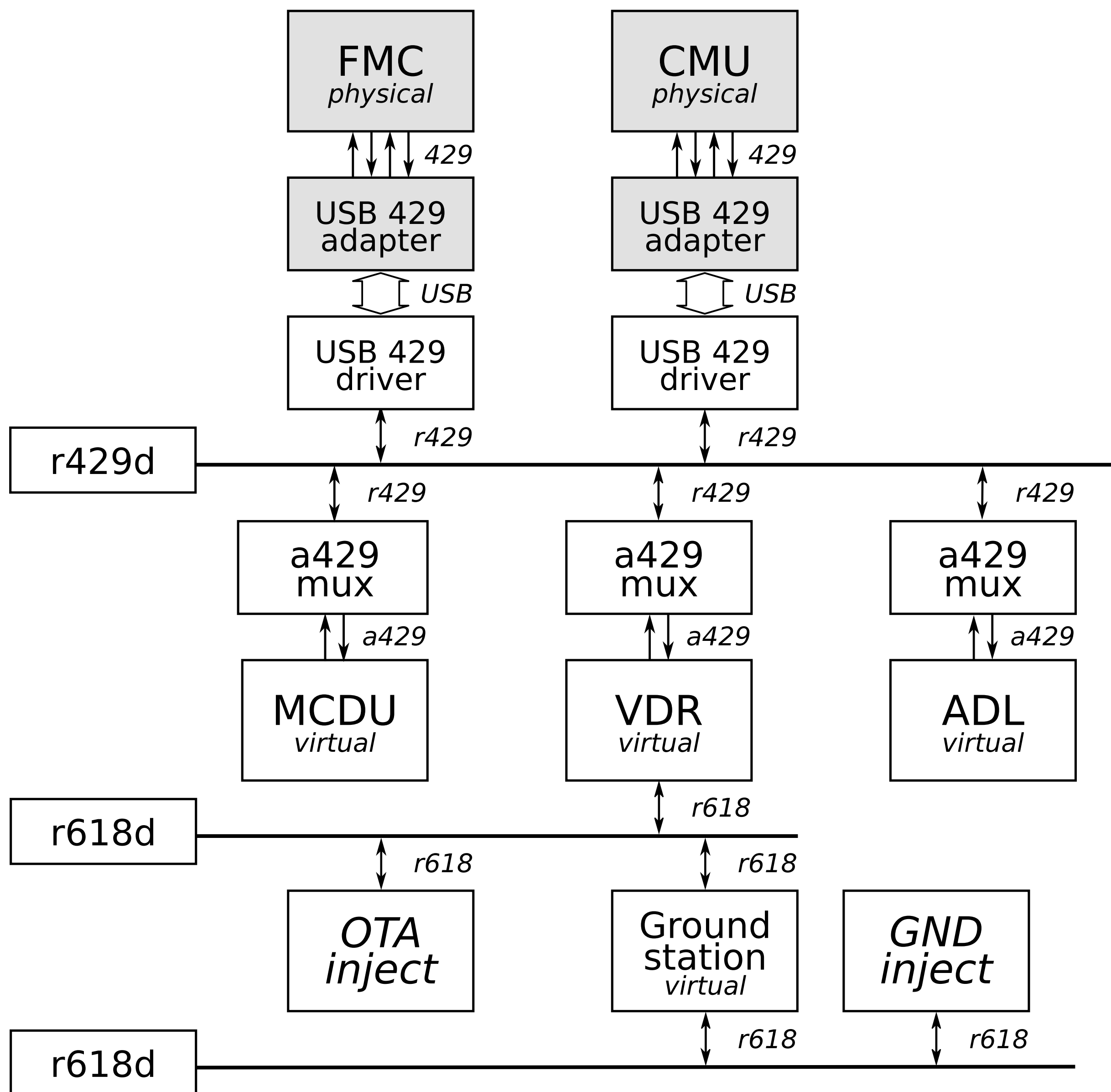

Figure 3.2: Portion of the Triton testbed.

virtualized bus that, from the hardware LRUs point of view, would be logically indistinguishable from a physical interconnect. In Section 3.4.2, 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 3.6.4.

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.

3.4.2 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 3.2), 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.

3.4.3 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.

3.4.4 Physical components

Our research tasks (Section 3.3) involve analyzing the behavior of real avionics and so we connect several LRUs to our testbed, as shown at the top of Figure 3.2. 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.

3.4.5 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 3.4.3), 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 3.4.4), 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.

3.4.6 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 [38].

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.

3.5 Experiment examples

The Triton testbed supports several kinds of experiments. Here, we describe two such experiments based on the analysis tasks outlined in Section 3.3.

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 3.4.6) 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.

3.5.2 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 3.3.1) and the data loader attack vector (Section 3.3.2).

3.6 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.

3.6.1 Challenges working with avionics

Integrating physical avionics components with simulated and emulated components as described in Section 3.4 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.

3.6.2 Lessons learned: two early lessons

Following the Unix philosophy (Section 3.4.1), 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

[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.

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 `R`429 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.

3.6.3 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. Attaching 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 3.6.2) 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.

3.6.4 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.

3.7 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 [53, 32], ADS-B [30], and Satellite [48, 47, 49] 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 [56]. 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 [15]. 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 [23] and the Air Force's Reconfigurable Cockpit and Avionics Testbed (RCAT) operated by MITRE [45, 59]. 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 [61] 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

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

testbed [41] and expanded yet more in DARPA's National Cyber Range program [24]. 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.

3.8 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 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 chapter, 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.

Chapter 3, in full, is a reprint of the material as it appears in Sam Crow, Brown Farinholt, Brian Johannesmeyer, Karl Koscher, Stephen Checkoway, Stefan Savage, Aaron Schulman,

Alex C Snoeren, and Kirill Levchenko. Triton: A software-reconfigurable federated avionics testbed. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Bus Driver: No-cut Message Modification on Aviation Data Buses

After building a test setup, the next major challenge is how to use it most efficiently to find security problems. Although this chapter describes a device that performs an attack, the same approach can help test devices to see how they respond to modified messages. Compared to the normal method of disconnecting the bus wires and installing something in the middle, this approach lets a single test device on a bus can modify messages from any other connected device without being moved around or rewired.

4.1 Introduction

This chapter investigates a seemingly simple, but practically quite complex, question about industrial data buses: the extent to which attacker-in-the-middle capabilities require the attacker to physically interpose on all communications (i.e., cutting the wires and partitioning the bus), or if such capabilities can be obtained with a no-cut attack (i.e., through signal manipulation) by attaching a device to an unused port on the bus.

This question is motivated by the observation that such buses are ubiquitous in a wide array of safety critical applications (notably in transportation and defense) and typically lack meaningful cryptographic security that would guarantee message integrity, freshness or authenticity. This design has been deemed sufficient because such systems (e.g., airplanes, ships, automobiles, weapons systems, etc.) are typically self-contained and thus, physical access is required to interact with their internal communications. While physical access is commonly deemed "out of scope" in many non-consumer threat models, there are practical reasons for reconsidering this issue. Indeed, the underlying premise has traditionally been that significantly tampering with a system's internal communication networks is a complex, time-consuming process that is too operationally challenging to be attractive for most attackers. To be concrete: we imagine it would be difficult to obtain hours of unmonitored time to rewire and tamper with systems in a commercial aircraft.[1]

However, there are reasons to reconsider this risk given both changes in technology and attacker capabilities. We increasingly see evidence that transitory physical access (i.e., physical access for a few minutes or less) is sufficient to obtain bus access. For example, recent attacks on gas station pump card readers (i.e., skimmers) have shown that attackers can reliably install vampire taps on internal data buses in less than 10 seconds [14]. Similarly, a range of platforms provide "debug" or "maintenance" ports that provide signal access to key buses even more quickly, without needing any access to underlying wires (e.g., OBD-II for automotive CAN buses [37, 25]). Finally, modern commodity board fabrication enables even amateur attackers to construct highly miniaturized "implants" that incorporate both bus communication drivers and general-purpose microprocessors, but also wireless communications to allow remote control or configuration. Thus, in a range of contexts, transient physical access may be sufficient to allow signal eavesdropping or signal injection. However, such access is not, by itself, sufficient to edit or replace existing legitimate messages on the bus—an attacker-in-the-middle capability—because

[1]By contrast, it is understood that, for consumer electronics, it is entirely reasonable to anticipate that a device—such as a smart phone or game console—may be put on a lab bench for hours or days to gain access to protected information.

simply obtaining bus access does not foreclose other parties from transmitting.

In this chapter, we explore this question in some depth in the context of two critical industrial buses: ARINC 429, the workhorse bus for commercial and transport aviation, and MIL-STD-1553, an equivalently common bus used in military aircraft and weapons systems. We explore the extent to which a foreign implant with physical access to a port on such a bus is able to effectively override or cancel signals from the connected bus devices and substitute the signals of their choosing, without being detected. We show that direct-coupled buses, such as ARINC 429, are at risk from such manipulations, ironically because their underlying robustness provides the freedom to effectively override and replace existing signals. However, we show that transformer-coupled buses, such as MIL-STD-1553, present significant obstacles to such attacks, requiring an implant to obtain multiple independent bus connections.

In particular, this chapter makes the following contributions:

- **Bus Vulnerability Analysis**. We analyze the logical and physical characteristics of ARINC 429 and MIL-STD-1553 to identify the necessary conditions to cancel and replace existing data signals.

- **Empirical assessment**. We build implants that carry out such attacks in practice in a bench-top setting.

- **End-to-end case study**. We design, implement and demonstrate a prototype implant that can be easily installed into a maintenance accessible port in the Electronic and Equipment bay of the Boeing 737. We show that, using a version of the techniques we describe, it is able to completely mediate communications between the Flight Management Computer (FMC), which is responsible for flight plan navigation, and the Multi-Purpose Control Display Unit (MCDU), which provides the user interface for such functions.

Finally, we discuss potential countermeasures or mitigations that would prevent or greatly complicate such attacks in practice.

4.2 Threat model

In this section we introduce Bus Driver attacks, a new no-cut interception attack on industrial data buses. Attackers with brief physical access (∼1 min) can implant a device into an unused bus port and perform attacker-in-the-middle attacks without intercepting the wiring. We first provide background on how this threat is different from prior attacker-in-the-middle scenarios. We then provide background on the electronics of data buses to explain why a no-cut interception attack is feasible, followed by a description of the technical challenges attackers face in achieving a transparent no-cut attack.

4.2.1 Constraints and opportunity

In an aviation environment, attackers can only have brief physical access to the electronics components in a target system. Industrial systems are often in use, so physical access is limited to brief time periods when the system is idle or being maintained (e.g., a plane at the gate). During those idle times, an attacker has an opportunity to access the electronic components of the system. For instance, aviation ground crew can access the Electrical / Equipment compartment (E&E) of the Boeing 737—where its data bus connectors are located—when it is parked at the gate. This is routinely done to inspect for equipment errors. Empirical measurements show that a person on the ground without any special equipment can open (or close) the 737 E&E compartment in approximately 15 seconds.

Constraints: time and avoiding detection. An attacker needs to quickly find the data bus wires that they want to intercept (i.e., the one with their target transmitters and receivers), and intercept them without being detected. Given these constraints, it is not feasible for an attacker to perform a typical attacker-in-the-middle attack. The first challenge is that industrial systems often are constructed with many bundles of wires (e.g., a wire loom from a Boeing 737 shown in Fig. 4.1), each containing dozens of unlabeled wires. This makes it difficult for an attacker

Figure 4.1: Wire loom from a Boeing 737. It is difficult to find and cut specific wires in an unlabeled bundle of cables.

to identify and cut a specific set of wires for a specific bus that connects to a target transmitter and receiver. The second challenge is, they may not be able to cut them because equipment on industrial data buses tends to perform frequent probing to ensure ensure its components are operating properly. If it is not, an equipment failure will be recorded (e.g., a BITE error on the Boeing 737). Some buses have built-in mechanisms that check for errors like invalid message checksums, unexpected bit patterns in messages, and messages that are too far, or even too close together in time. To avoid detection, the attacker needs to avoid creating any errors–even transient ones–that the target system can detect. Finally, if the attacker requires persistence, their device should be difficult to physically detect and splices into the middle of a wiring loom may be more challenging to conceal.

Opportunity: Unused data bus ports. However, many industrial systems expose their data buses through unused ports. In particular, aircraft and military equipment often have unused ports directly connected to data buses for troubleshooting and training purposes. For instance, the Boeing 737 has multiple maintenance and test ports providing access to data buses in the E&E compartment. An example of one such port is shown on the top of Fig. 4.2, which exposes an

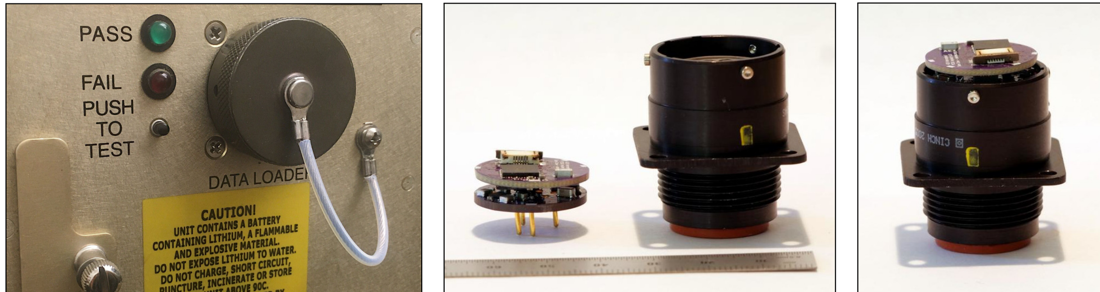

Figure 4.2: Unused data bus ports on the Boeing 737 present an opportunity for an attacker to hide an implant inside of the port.

ARINC 429 data bus to load firmware updates into a particular "line replaceable unit" (LRU) on the Boeing 737.

Data bus ports also present an opportunity to avoid physical detection while simultaneously providing persistence to the attacker: a malicious device can be hidden inside of the unused data port. Figure 4.2 shows an example of an entire physical implant that we constructed to fit inside of the dimensions of a test port on the Boeing 737 (details in Section 4.4). Indeed, this implant can even be remotely controlled because it includes a built-in WiFi and Bluetooth wireless interface. For instance, this could allow it to be operated over an aircraft's in-flight WiFi network, whose main control is also in the E&E bay. Based on our experience, we believe that an attacker with access to an open 737 E&E compartment would be able to remove the dust cap from an existing maintenance port and connect a miniaturized implant device (such as the one we have constructed) within 30 seconds.

4.2.2 Bus Driver: A no-cut interception attack

In this work we evaluate a new type of industrial data bus attack called a Bus Driver attack. A Bus Driver can achieve the same interception and overwriting capability as attacker-in-the-middle, however, instead of splicing a Bus Driver only needs to attach to the bus on an unused port. Figure 4.3 compares a Bus Driver attack with a traditional attacker-in-the-middle. Bus Driver attacks are inspired by a class of wireless communication attacks that also achieve

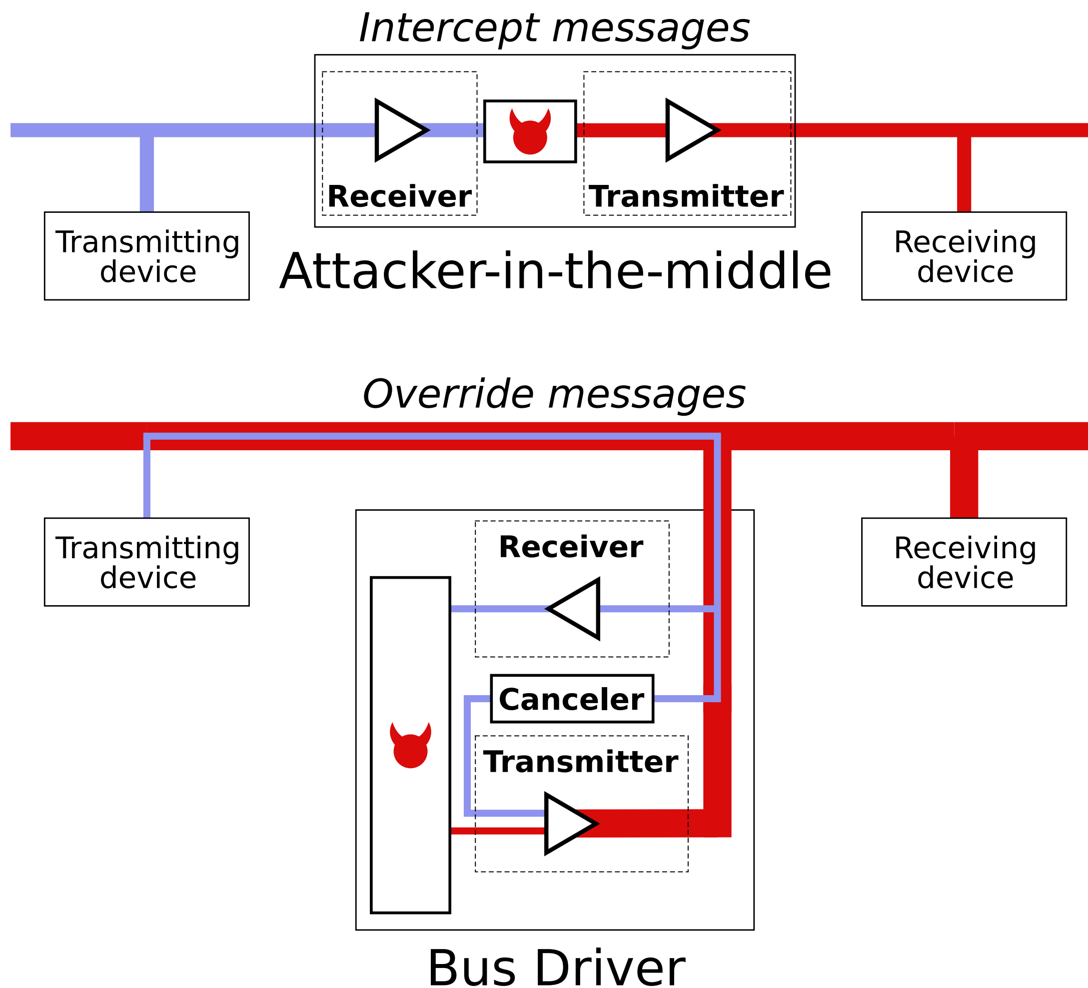

Figure 4.3: A conventional attacker-in-the-middle attack splits a bus into two parts to intercept and modify messages. A Bus Driver implant just needs to connect to one point to intercept data and change it.

attacker-in-the-middle capability, but using active cancellation, instead of physically separating the channels between the target device [28, 19].

Essentially, a Bus Driver intercepts messages by simultaneously receiving what the legitimate transmitter is sending while also cancelling the original signal so that it does not reach the legitimate receiving device (Fig. 4.3 bottom). Then, it modifies the data and injects a malicious version of the transmission to be received by the receiving device. Compared to cutting wires, a Bus Driver requires no physical modification of the bus; therefore, it evades most electronic forms of detection. It also has the additional benefit that it does not require specific placement on the bus between the target transmitter receiver, as long as the unused port is on the same bus, it can perform the attack regardless of where the targets are relative to the malicious device on the bus.

Electrical description. In a wired data bus, a device can send messages to the other device by changing the voltage between two wires. Nominally, a transmitting device drives a small amount of current through its connection to the bus to induce a voltage change that the receiving device can detect. To suppress a message, an implant needs to (effectively) sink this current such that no perceptible voltage change occurs. To completely override a message, an implant must also drive sufficient current to induce its desired voltage on the bus. To manipulate a message on the fly, an implant must be able to monitor what it is cancelling while it is overriding, so that the implant can know the original data.

4.2.3 Challenges

Although the attack is simple in principle, there are several bus implementation-specific challenges that may make it infeasible to execute Bus Driver attacks in practice.

C_{SAFE}**– Not destroying the legitimate transceivers.** Canceling a transmission on a bus requires driving current opposing the transmission on the bus. Electrically, this will cause the legitimate

transmitter and receiver to sink that current, likely significantly more than it is designed to handle. This could lead to the destruction of bus frontends on legitimate devices.

c_{CANCEL}– **Fully canceling the legitimate transmission.** The simplest way an attacker can cancel a legitimate transmission is to short-circuit the two bus wires. The transmitting device will then try to send a message by sending current through the wires. Because of the short circuit, there will be no voltage difference between the wires. However, this would result in not being able to inject a malicious data signal because the short circuit blocks all signals. Instead, the cancellation has to be strong enough to cancel the legitimate signal, but not so strong that it also cancels the injected signal. This becomes particularly difficult for data buses that are not directly coupled electrically between the transmitter and receiver. For instance, industrial data ports can be *transformer* coupled to the data bus, which means that an attack cannot directly drive current on the bus wires to cancel a transmission. Instead, it has to deliver that current through a transformer that was not explicitly designed with enough bandwidth or current delivery capacity to allow for canceling transmissions on the bus.

c_{SYNC}– **Synchronizing timing with a legitimate transmitter.** Canceling and overriding transmissions on industrial data buses will require tight time synchronization between the legitimate transmitter and the attacker. The reason is that these data buses have tight timing requirements for when messages should arrive, and if messages are missing, improperly canceled such that they introduce bit errors, or arriving at the wrong time, then the system will detect that the device transmitting or receiving has failed.

4.3 Feasibility of Bus Driver on aviation buses

In this section we evaluate the feasibility of Bus Driver attacks on two different buses commonly used in commercial (ARINC 429) and military (MIL-STD-1553) aircraft.

4.3.1 ARINC 429

The first bus we test the Bus Driver attack on is ARINC 429 [5]. ARINC 429 is a unidirectional serial bus in common use in commercial and transport aircraft (e.g., 737). Each bus consists of one transmitter connected to one or more receivers. Bidirectional communication between aircraft computers—called *avionics*—is achieved using two buses transmitting in opposite directions between the avionics.

Attackers need to find the specific bus for their target. Aircraft have many separate ARINC 429 buses, each with one transmitter and a small number of receivers. One of the benefits to this approach is that an attacker attempting to manipulate the traffic on any given bus has to physically connect to that specific bus. Also, to manipulate messages sent in both directions between devices, the attacker must connect to both buses. This stands in contrast to other common buses like CAN where a single bus supports every device in the system, including multiple transmitters and receivers.

ARINC 429 is a simple and reliable bus. ARINC 429 is an electrically simple, two-wire bus. The transmitter and each receiver directly connect to the bus and share a common ground [5, Section 2.2.1]. An ARINC 429 transmitter is directly connected to each bus wire with a $37.5\,\Omega$ resistor [5, Section 2.2.4.1 and Attachment 4]; see Figure 4.4. These two resistors are designed to provide an output impedance that matches the characteristic impedance of the bus wires. They also have the effect of limiting the current that can flow into or out of the transmitter. The two bus wires are normally at zero volts with respect to the shared ground. The transmitter sends a message by supplying $5\,\mathrm{V}$ on one wire and $-5\,\mathrm{V}$ on the other wire. The receivers measure the differential voltage between the bus wires. A positive voltage is interpreted as a 1 bit, and a negative voltage as a 0 bit.

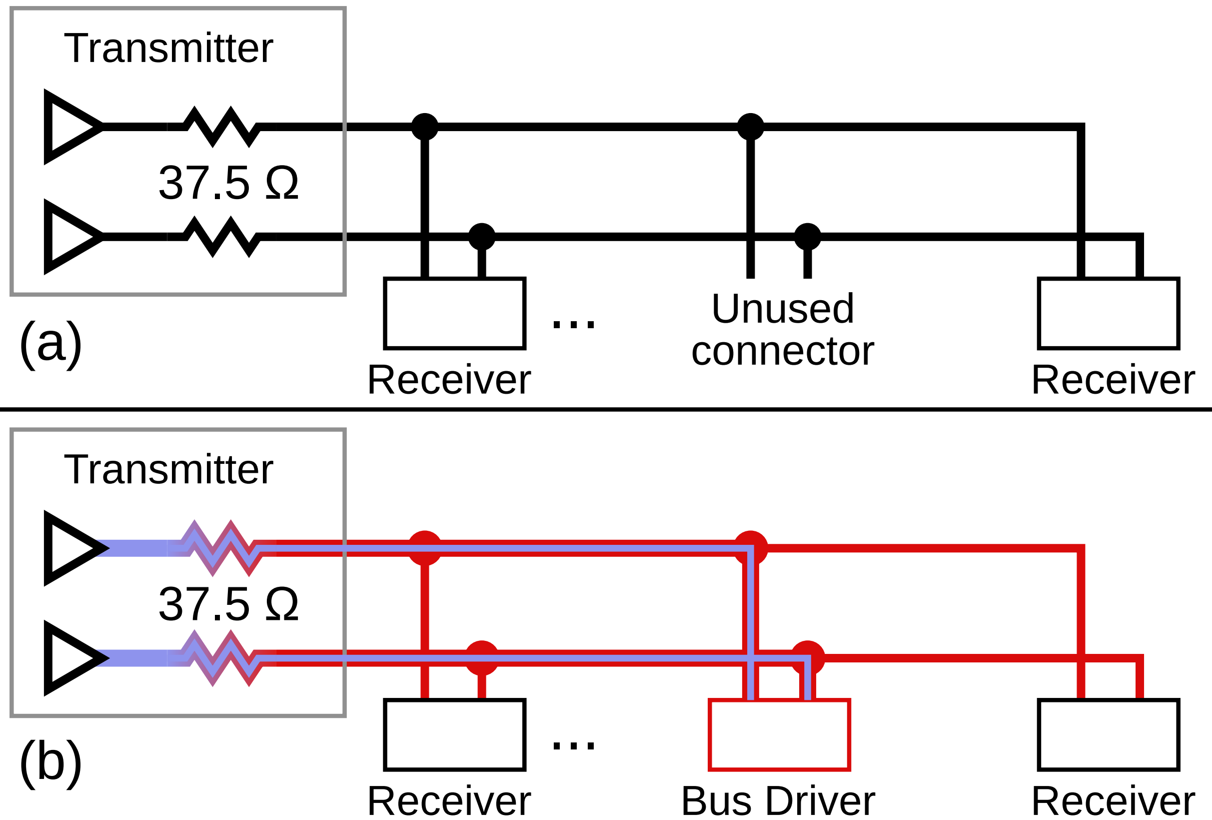

Figure 4.4: (a) An ARINC 429 bus with an accessible connector, (b) An implant changing the voltage on the bus. The thick lines represent the current that flows between the implant and transmitter. The resistors in the transmitter limit the current and protect the transmitter from overheating.

ARINC 429's resistors make it vulnerable. One unintended consequence of the inclusion of these resistors is that they limit the current from the legitimate transmitter. This makes it possible for a rogue device connected to an used port on the bus to override the legitimate transmitter and cancel, allow, or modify a transmission.

4.3.2 Proof-of-concept ARINC 429 implant

Despite the simplicity of the ARINC 429 bus, actually overriding specific messages is more complicated than connecting a transmitter (without resistors) to the bus and transmitting. In the next section, we describe our proof-of-concept ARINC 429 implant.

Theory. The resistors limit current from the legitimate transmitter, enabling a rogue device to fully cancel a legitimate transmission ($\mathbf{C}_{\text{CANCEL}}$). While the rogue device is canceling and modifying a transmission, the legitimate transmitter will be acting as a current sink. However, the resistors prevent the transmitter from sinking too much current, protecting it from damage ($\mathbf{C}_{\text{SAFE}}$), not destroying the legitimate transceivers. It's important to note that this is not a quirk of one particular implementation of the standard. Instead, the standard mandates the inclusion of the resistors [5] which enable the Bus Driver attack. We also have to synchronize timing with the legitimate transmitter ($\mathbf{C}_{\text{SYNC}}$).

Implant Design Figure 4.5 shows the major components of the ARINC 429 Bus Driver. standard ARINC 429 transceiver driven by a microcontroller (MCU) is connected to the bus with the addition of two high-current amplifiers between the transmitter and the output. These amplifiers drive enough current that the Bus Driver's ARINC 429 transmissions cancel out and replace the transmissions on the wire from the legitimate transmitter. In addition to the normal ARINC 429 receiver that allows the implant to receive the legitimate transmissions when it is not overriding, it includes a current sensor and comparator attached to the bus after the amplifiers that allows the

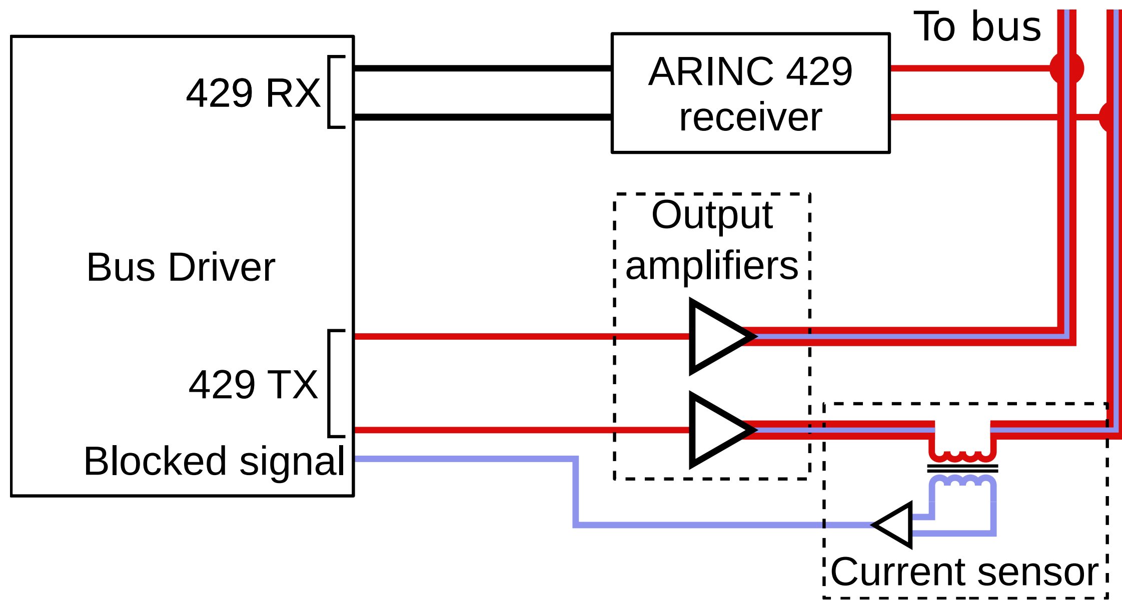

Figure 4.5: A simplified block diagram of the 429 implant. When the implant is active, its output amplifiers drive current through the bus. A current sensor measures that current. If the implant is blocking a message, the current reveals the content of the blocked message.

Bus Driver to read legitimate transmissions on the bus simultaneously while it is canceling them.

Implementation. We designed and built several versions of a proof-of-concept ARINC 429 Bus Driver which can allow, block, or modify legitimate ARINC 429 transmissions. In addition, our implant can read the messages that the legitimate transmitter sends and at the same time prevent any other receiver on the bus from receiving a message. See Section 4.4 for a case study where an attacker who has brief physical access to a Boeing 737's Electronic Equipment Bay can override information sent from the Flight Management Computer (FMC) to a display in the cockpit.

The first version of the implant used a separate Holt HI-3593 ARINC 429 physical layer transceiver chip to receive and send messages on the bus with the correct timing ($\mathrm{C_{SYNC}}$). This limited what the implant could do in a few important ways. First, the MCU did not know that another device was sending a message until after the entire message had arrived. Second, when the MCU instructed the HI-3593 to send a message, the HI-3593 added an unpredictable delay of

tens of microseconds before sending the first bit of the message. That made it impossible for the implant to detect a specific label in a message and modify other bits within the same message, or to reliably send a message in between two other messages sent by another device with the minimum time period (40 µs) between the messages. In essence, the first version of the implant was not able to overcome $\mathbf{c}_{\text{SYNC}}$.

In order to overcome this challenge, we removed the separate ARINC 429 chip and implemented its functionality in firmware on the microcontroller. This change has two clear benefits but comes with an additional challenge. First, it enables the implant to react to legitimate messages during transmission. Second, it saves space on the circuit board. Saving space is not our primary concern, but it did ease the layout of the final prototype which is space-constrained since we designed it to fit inside a standard receptacle (Figure 4.2)). The new challenge is that because the MCU can no longer leverage a separate chip for sending and receiving ARINC 429 words, we needed to carefully structure the firmware to send and receive at 100 kbps.

To receive messages, we use a Holt HI-8591 ARINC 429 physical layer receiver chipset to convert the bus voltage into two digital signals. We use the MCU's motor control pulse width modulation (MCPWM) peripheral to measure the duration of each bit in the message and trigger an interrupt when one of the signals becomes high. The interrupt handler checks that each bit has the correct timing and collects the 32 bits of each message into one integer value for processing. This works reliably as long as the microcontroller is running at its maximum frequency of 240 MHz. Otherwise, the interrupt handler takes too long to run and misses some messages.

An alternative approach to waiting for an entire 429 word to be received before acting is to trigger its overriding functionality after the initial bits of the message—for example the 8-bit label—has been received. That lets the implant modify the other bits in the message.

To transmit messages, we use bit banging. Because the MCU's software libraries do not provide a good way to wait with a resolution of more than 1 µs, and to avoid timing errors

caused by interrupts, we do not use either of the MCU's main cores. Instead, we use the ESP32's Ultra-Low-Power (ULP) coprocessor. The ULP coprocessor has a very restricted instruction set that makes it challenging to program, but it can wait for a precise number of clock cycles and never gets interrupted by any other software. The ULP software controls two digital outputs, which the output amplifiers convert into positive and negative voltage on the bus.

While blocking or overriding legitimate transmissions, current flows between the legitimate transmitter and the implant through the bus wires. The direction of current depends on the contents of the legitimate transmission. By connecting a current sensing transformer to the output of one output amplifier, the microcontroller can measure this current over time and decode the message, even though the other receivers on the bus will not receive the message.

The ability of the implant to read transmissions while simultaneously hiding them from other receivers enables attacker-in-the-middle capabilities without needing to physically disconnect the bus between the transmitter and receivers.

Evaluation A full case study showing an end-to-end attack where the implant can change characters on a critical cockpit display of a Boeing 737 using the ARINC 429 Bus Driver is described in Section 4.4.

Although the implant can succeed to cancel and override ARINC 429 messages, it may introduce a significantly different waveform onto the bus when it does this that could easily be detected by improving the bus fault detection logic in ARINC 429 receivers. Figure 4.6 shows the waveform of a legitimate transmission (blue) followed by a canceled and overridden transmission (red). The Bus Driver cancels entirely, and produces an identical waveform as the legitimate transmitter. However, the Bus Driver's waveform does have more noise (likely due to the high-gain amplifier) and the voltage on the bus is not exactly the same as the legitimate transmitter. Since they are not identical, it may be possible to create a defense against this attack by sampling the full analog waveform of the bus and finding incongruities using digital signal

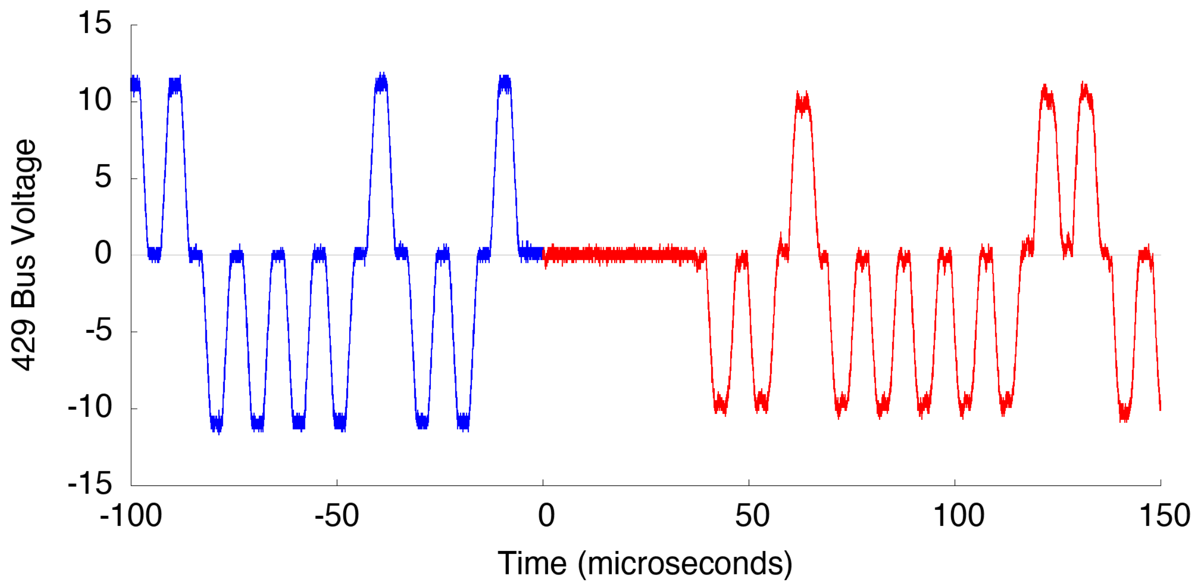

Figure 4.6: The ARINC 429 bus waveform during a legitimate transmission (blue) is nearly indistinguishable from the Bus Driver attack (red).

processing. However, as we describe in Section 4.5, this will significantly increase the complexity and cost of all ARINC 429 receivers in the system.

The simplicity of the ARINC 429 bus—specifically that devices directly connect to the bus—coupled with the transmitter's output resistors makes the attack easier than it would be on a more complex bus with isolation between devices. Nevertheless, this direct connection is not required.

4.3.3 MIL-STD-1553

We have demonstrated that ARINC 429's design as a resilient industrial bus made it particularly vulnerable to Bus Driver attacks, but there is another bus standard, MIL-STD-1553, which is used by the military for aircraft applications [57], and also in weapon systems (e.g., the Javelin Missile). Here, we evaluate if a Bus Driver attack is feasible on this bus.

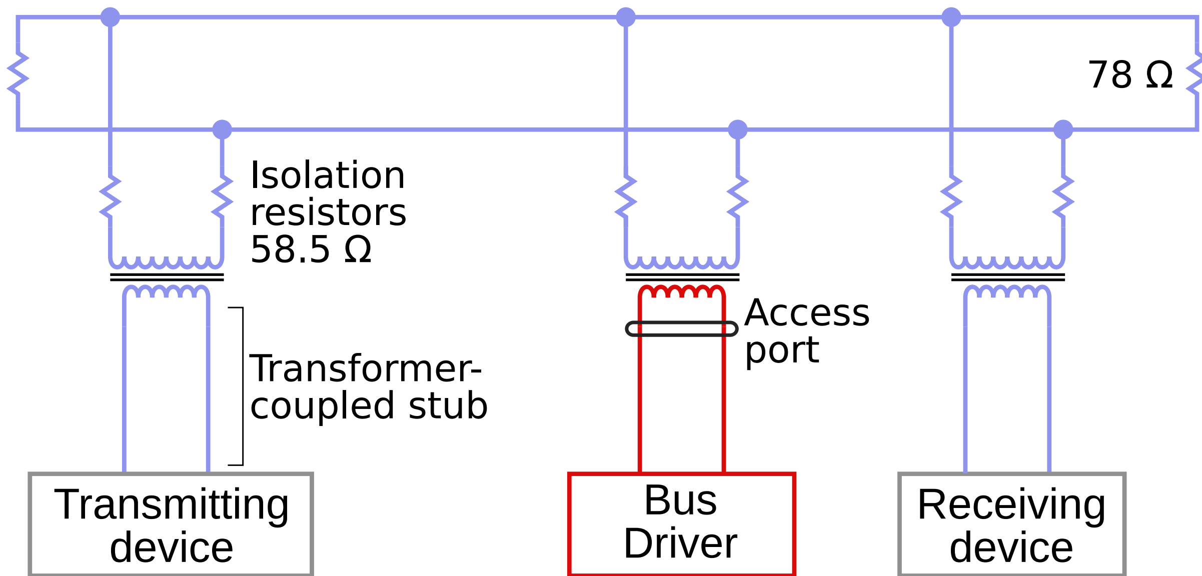

Figure 4.7: A MIL-STD-1553 bus's transformer-coupled stubs offers protection against a Bus Driver implant.

Differences between MIL-STD-1553 and ARINC 429. MIL-STD-1553 has an even more resilient bus design than ARINC 429. In particular, MIL-STD-1553 adds coupling transformers which isolate devices from the bus wiring. These make it difficult to fully cancel the legitimate signal on the bus, which potentially makes it infeasible to perform a Bus Driver attack ($\mathbf{C}_{\text{CANCEL}}$). The coupling transformer must be physically near the main bus wiring, while up to 20 ft of wire are permitted between the transformer at the bus connection and the port where a device ultimately attaches—a Bus Driver cannot opt to omit nor can it access to remove the coupling transformer. An additional challenge is that the MIL-STD-1553 standard requires receivers to tolerate significant attenuation: a Bus Driver attack has to cancel the legitimate signal from up to 13 V down to under 0.6 V [57].

Isolation provides some protection against Bus Drivers. The original intent of transformer isolation is to improve bus resilience to faults (e.g., a short cannot render the bus inoperable), but we also find it serves as a mitigation for Bus Driver attacks. These transformers provide galvanic isolation between the devices and the bus—an air gap that makes it infeasible for current to

directly flow between the bus and the attached device. With transformer-coupling, the amplifier-based attack circuit we described for ARINC 429 would not work. The problem is, the amplifier's cancellation would only cancel signals on the same side of the transformer that the attacker is on, the other side of the bus will still have the legitimate transmission on it (Fig. 4.7).

Due to this limitation, it is impossible to perform a Bus Driver attack through a single port to a transformer-coupled bus. However, we will next demonstrate that connecting to a MIL-STD-1553 bus through two independent ports may make it feasible to perform a Bus Driver attack. The presence of isolation still increases the difficulty of the Bus Driver attack as it now requires two open ports in close proximity on a target system. We note, however, that some MIL-STD-1553 bus coupling modules include multiple independent connection ports on a single device [21].

4.3.4 Proof-of-concept MIL-STD-1553 implant

We now describe an implant we built that demonstrates a Bus Driver attack may be feasible on MIL-STD-1553.

Theory. This attack requires two independent bus connections. Two ports are needed as this approach must separate sensing the legitimate transmission on the bus from injecting the cancellation signal onto the bus. This separation is needed because active cancellation is a feedback loop.

To cancel, the original signal is inverted and amplifyed to create a cancellation signal. If this cancellation signal is connected directly back to the sampling input (i.e., injected onto the same bus connection that was used to listen to the bus), the slight imperfections in cancellation[2] will also feed back into the cancellation circuit, which will then be further amplified over and over again [42]. One way to address this problem is to physically separate sensing from cancellation,

[2]There is necessarily a small phase offset, since it takes time for the input signal to be inverted and amplified.

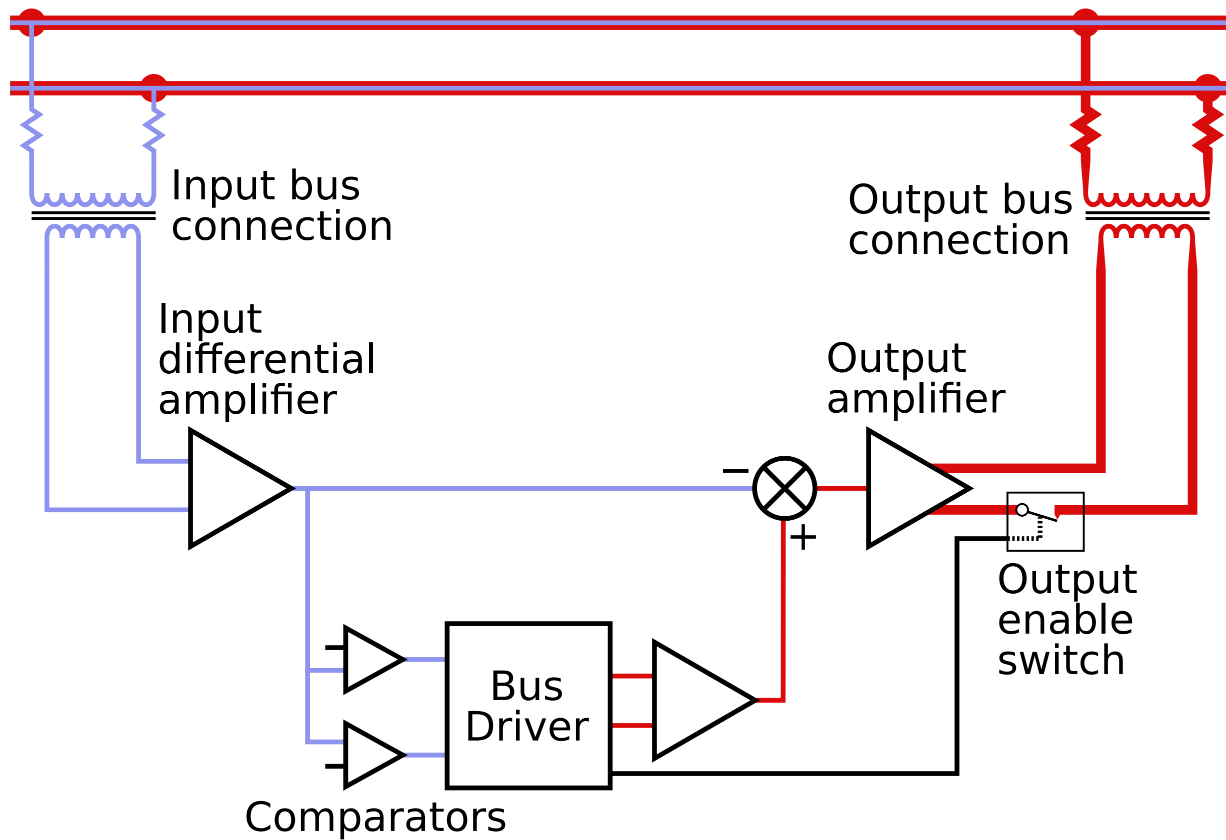

Figure 4.8: A block diagram of a MIL-STD-1553 Bus Driver attack which requires two connections to the bus.

which for our application will require attaching two different isolation transformers to the bus (i.e., connecting to two access ports). To help provide an intuitive understanding here, consider how noise canceling headphones are designed: the microphone sensing noise is separated from the speaker that outputs the cancellation signal into the ear such that the cancellation waveform is not picked up by the microphone.

Implant Design. Figure 4.8 shows the main components of the prototype Bus Driver implant. The input bus port leads to a differential amplifier, which senses the voltage on the bus. That voltage goes to two comparators that digitize the voltage as either positive, negative, or near-zero. A microcontroller (MCU) receives the signals from the comparators to monitor traffic on the bus. The MCU controls an analog switch that can enable or disable the output on the output bus port.

To inject messages while cancelling, the MCU has two digital outputs connected to a differential amplifier that can produce positive or negative voltages. Just before the output amplifier, another circuit adds the desired bus voltage from the MCU to the negative actual bus voltage. The sum voltage controls a high-current output amplifier that applies a voltage to the output port on the bus, which moves the actual bus voltage closer to the desired voltage.

Implementation. To evaluate the feasiblity of this attack, we created a prototype of a MIL-STD-1553 Bus Driver implant, and we evaluated how well it could overwrite a legitimate transmission on its transformer-coupled bus.

To control the implant and decide how message should be modified, we use an STMicroelectronics STM32F446 microcontroller. We hoped that this microcontroller would be able to receive MIL-STD-1553 messages in software. Although the STM32F446 is not as fast as an ESP32 (180 MHz compared to 240 MHz), it has less software overhead and gives us more control by default.

To receive messages, we first tried using an interrupt handler that runs every time the bus voltage changes. That approach worked for ARINC 429, but we quickly found that it was not

practical for MIL-STD-1553. MIL-STD-1553 uses a higher data rate (one megabit per second), so synchronizing with the transmitter is more challenging when performing selective overwriting. Because the bus voltage can change every 500 ns, the software has fewer than 90 clock cycles to check the signal timing and record one bit of the message.

To synchronize the Bus Driver with the legitimate signals, we start by detecting the sync waveform at the beginning of each data message. This waveform has a pulse of low voltage followed by a pulse of high voltage, both 1.5 µs long. Our software constantly polls the two inputs that represent the current bus voltage. It ignores any pulses that are too short or too long. When a message starts with a negative pulse and the beginning of a positive pulse, the software starts generating signals to modify the rest of the message.

Generating signals to send MIL-STD-1553 messages with the correct timing is easier, but still will ultimately require bitbanging. Using the quad SPI peripheral to control the two outputs works well and produces the perfect bit rate. However, there is an unpredictable delay of up to 500 ns between when the software asks to send a message and when the first bit actually appears on the outputs. A delay of 500 ns between the sync waveform and the rest of the message makes the entire message invalid.

That unpredictable delay is the same problem that we had using a separate chip to send ARINC 429 messages. Again here, bit banging provides better timing control. While inefficient, polling and bit banging lets the implant reliably detect every data message on the bus and send signals to modify the message content with the correct timing.

Evaluation. We evaluated the feasibility of this attack by observing how well the prototype Bus Driver can cancel and override transmissions from the transceivers of an industry-standard Holt HI-6138 MIL-STD-1553 chip.

We discovered that while the prototype can sense the bus voltage and supply the cancellation voltage to block messages, it could not achieve full cancellation. The nominal bus voltage

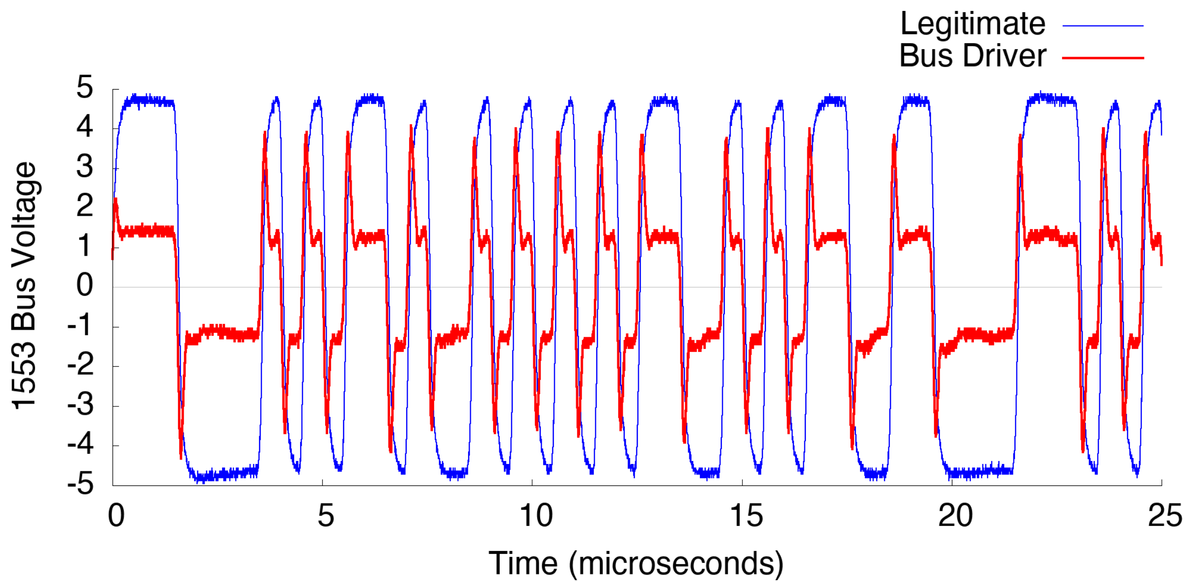

Figure 4.9: A legitimate MIL-STD-1553 waveform compared to the Bus Driver's attempt to cancel the waveform.

for a legitimate transmission is 9.7 V peak-to-peak. The implant reduces it to only 3.0 V, well above the standard's threshold of operation of 0.6 V peak-to-peak. This is not enough to fully cancel a legitimate transmission, but it shows that the concept of canceling using two ports may be feasible.

Our prototype setups so far could not achieve full cancellation due to several factors. Figure 4.9 shows a legitimate MIL-STD-1553 waveform compared to the waveform when the Bus Driver implant tries to cancel the transmission. Specifically, it does not cancel the beginning and end of each bit as there is a delay in the cancellation circuit's inverting amplifier that is proportional to its bandwidth. However, during the transmission of each bit it does succeed to decrease the amplitude of the bit substantially, however because the isolation resistors in the port between the implant's output and the bus, the amount of current that the output amplifier can supply is limited.

One possible solution is to use a different output amplifier with a higher maximum output voltage. Therefore, we tried several different high-gain amplifiers, including amplifiers designed

specifically for this type of application, namely long-distance telecommunication systems (e.g., VDSL). However, we found that the higher the output voltage, the more likely it was that the cancellation circuit would end up in an oscillation feedback loop. This is due to a classic problem with feedback loops where higher gain can result in unwanted oscillation.

Summary. Overall, galvanic isolation from transformers significantly mitigates Bus Driver attacks. It requires an attacker find two unused ports and limits the range of voltages which can be effectively cancelled.

4.4 Case study

To show that the Bus Driver attack is viable, we will explain in this section how an implant can attack a Boeing 737 Next Generation airliner. The 737 Next Generation series is one of the world's most popular airliners, with over six thousand delivered between 1997 and 2020 [31].

4.4.1 Boeing 737 systems

A Boeing 737 has many onboard computers that communicate using ARINC 429. Two of these computers, the Flight Management Computer and Multi-Purpose Control Display Unit, are vulnerable to an implant changing the messages they send to each other.

The Flight Management Computer (FMC) calculates flight plans between airports that pass through predefined waypoints and follow standard departure and arrival paths. In flight, it commands the autopilot and authothrottle to follow the planned route. The FMC also calculates important performance numbers, like the minimum runway length the plane needs to safely take off and the slowest speed it can fly at without stalling.

The Multi-Purpose Control Display Unit (MCDU), is the interface between the pilots and the FMC. It has a keypad and screen that the pilots use to enter and view information about the

flight plan. There are two MCDUs on the flight deck, one for the captain and one for the first officer.

Each MCDU has one ARINC 429 bus that it uses to send button press events to the FMC, and another bus where it receives text from the FMC to show on its screen. The two MCDUs operate independently and can show different information.

MCDU text protocol The FMC uses a simple protocol to split a page of text into 32-bit messages and send the text to the MCDU. After sending three messages to start the process, the FMC sends each character in a separate message. When a character repeats, the FMC uses run-length encoding to compress the text. Instead of sending a message for each consecutive instance of the character, it sends one message with the number of times the character repeats and one message with the character to repeat. To control where characters appear on the screen, the FMC simply sends space characters so the following text will show up in the correct place. The MCDU does not need to acknowledge any of these messages.

An accessible connector The buses between the FMC and the captain's MCDU also branch off to a connector that can be used during FMC maintenance. This connector is an ideal place for an implant. Anyone on the ground around the plane can reach the connector through a hatch. As figure 4.2 shows, one of our implant prototypes plugs into the connector and fits almost completely inside it. The connector supplies power to the implant and lets it access the buses. The connector is normally covered by a cap like in figure 4.2, which completely hides the implant. The implant can wait there until it activates.

4.4.2 Test setup

To demonstrate the efficacy of our implant on a real ARINC 429 bus, we assembled a test bed using real avionics, including a flight management computer (FMC) a multipurpose control

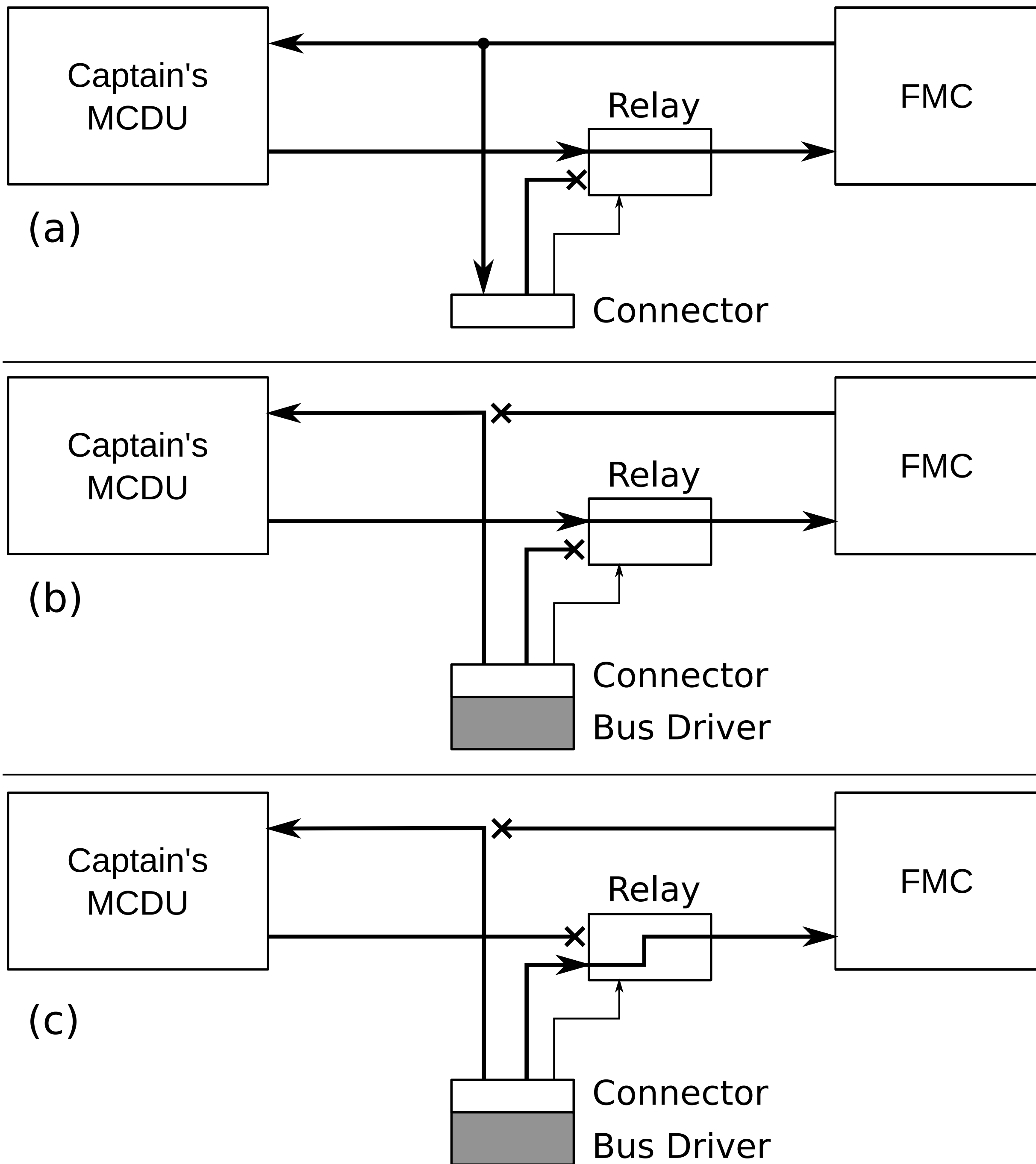

Figure 4.10: (a) The FMC and MCDU communicate normally. (b) An implant in the connector can modify or replace the text the FMC sends to show on the MCDU. (c) The implant can also switch the relay so it can send commands to the FMC.

display unit (MCDU), and an integrated flight system accessory unit (IFSAU).

The challenges associated with constructing avionics test beds are described in chapter 3. One challenge is that avionics are designed to use a 400 Hz AC power supply. A second challenge is that these systems are designed to operate in conjunction with many other avionics, some of which we do not own. We addressed the latter challenge by simulating the missing systems which are required for correct operation.

Our FMC is a GE 2907A4 that previously served on a Southwest Airlines 737. Our MCDU is a GE 2577F1 from a Batavia Air 737. Although these parts were designed for the older Classic-series 737s, some Next Generation 737s still fly with these parts today. They are interchangeable with and closely related to the newer FMC and MCDU models that are more common in Next Generation 737s.

The relay shown in Fig. 4.10 is part of the IFSAU, Boeing part number 65-52820-2. Figure 4.11 shows the complete test setup with an implant connected.

4.4.3 Implant capabilities

The implant can use a few different methods to change the text the MCDU shows or control the FMC.

Complete text replacement The implant can block all messages from the FMC to the MCDU and send its own messages to control what the MCDU shows to the pilots. This method is simple because the implant does not need to precisely time its messages, and it provides complete control over the screen. The main limitation is that the implant has no way to know when a pilot presses a button on the MCDU. A pilot would immediately suspect a problem if the MCDU screen did not respond to button presses.

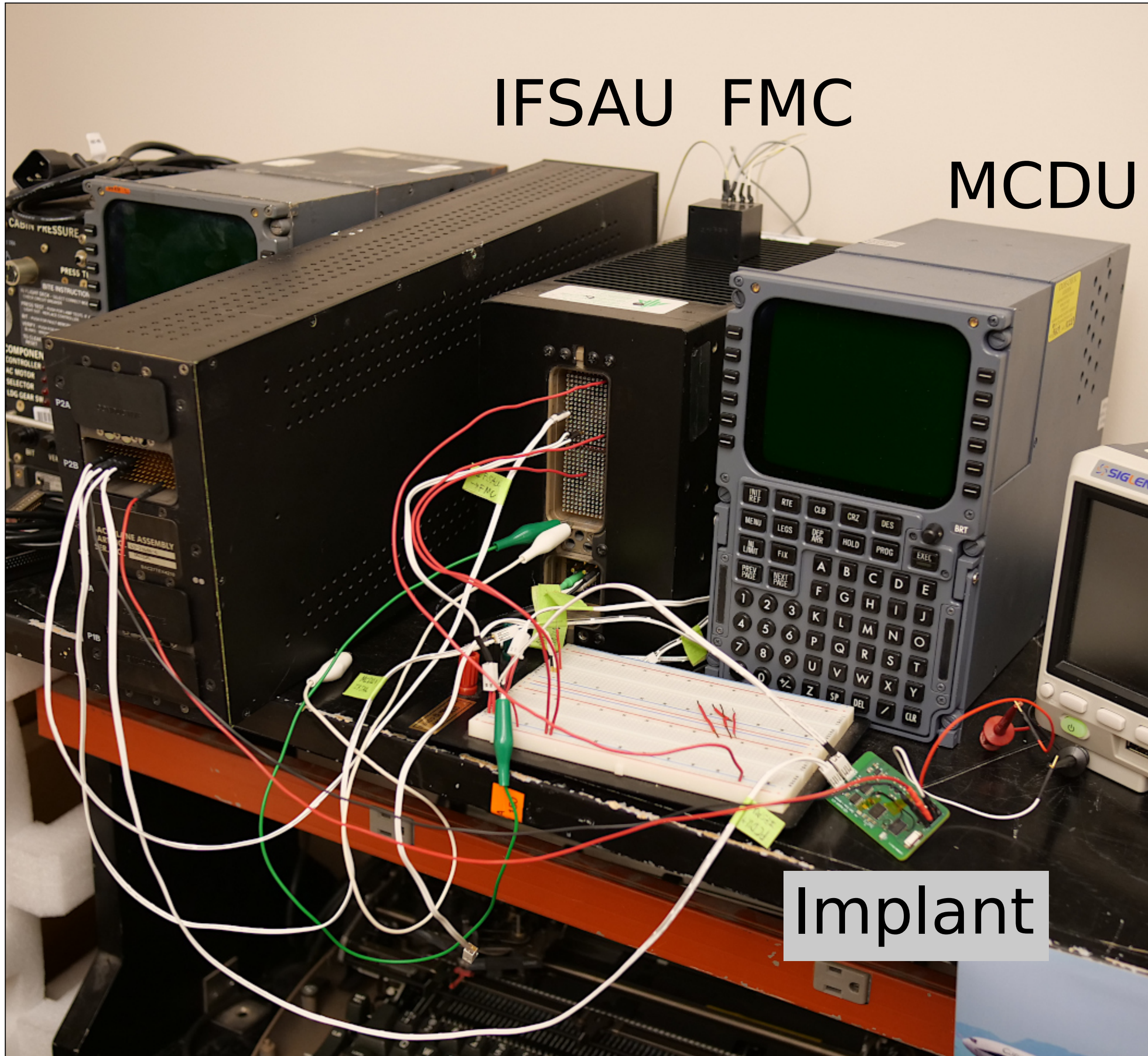

Figure 4.11: Our test setup with the main components labeled. The implant shown here is larger than the one in Fig. 4.2 because this version includes additional space between the components for easier debugging.

Partial text replacement To change the text on the screen in a way that is more difficult to detect, the implant can change only some of the messages from the FMC and let the other messages continue to the MCDU normally. The implant monitors what the FMC is sending and waits for a pattern of characters. When the pattern appears, the implant activates its outputs and sends a few messages to change the next few characters. The implant then deactivates its outputs and lets the FMC send the rest of the screen content normally. For example, the implant can detect the sequence I, D, E and replace the next character with any other character.

The replaced text needs to be encoded with the same number of ARINC 429 messages as the original text. Because the FMC uses run-length encoding to send text, some substitutions are not possible. For example, the FMC sends the number 10000 using three messages: character '1', repeat the next character four times, character '0'. The replacement text should also have five characters encoded in three messages. Any sequence with the same character repeated four times and some other character at the beginning or end would work, like 11114 or 3 (with four space characters before the digit). Other numbers like 10300 are not suitable because they need more than three messages.

The FMC sends all the messages that make up a screen update in rapid succession with a gap of 40 microseconds between messages. When the implant modifies one of those messages, it needs to send its message within about ten microseconds of the ideal time. If the message is too early or late, the gap before or after the message will be too short. That makes the MCDU exit to its menu page, which alerts the pilots that something is wrong.

Responsive complete text replacement The implant can make arbitrary changes to the text on the screen while still responding to button presses by following these steps.

1. Monitor messages that the FMC sends and store a complete screen of text in memory.

2. Change the text in memory.

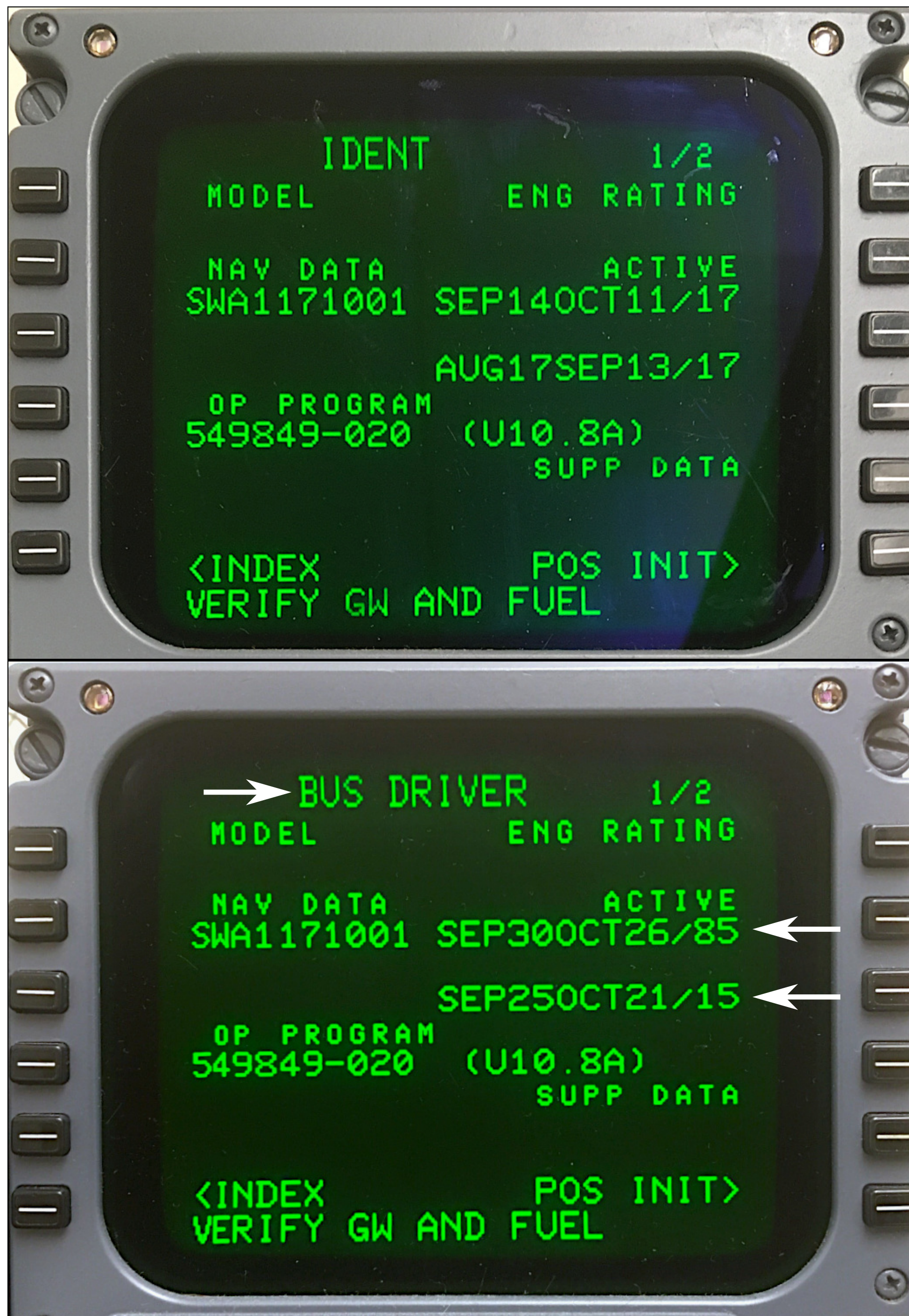

Figure 4.12: The MCDU screen showing a normal page from the FMC (top) and with some text modified by an implant (bottom). The title has been replaced by a longer title and the dates of the navigation database have changed. Arbitrary textual modifications are possible.

3. Encode the modified text into messages and send them to the MCDU.

The implant does not need to send messages with precise timing, and the modified text can be encoded with a different number of messages from the original text. Fig. 4.12 shows a simple modification that changes the title and the validity dates of the navigation database. Arbitrary textual modification—including adding, removing, or modifying letters, numbers, and symbols—is possible.

Current sensing The basic version of the implant cannot monitor messages from the FMC without also letting the MCDU receive those messages. The implant and MCDU receive text from the FMC at the same time, and the MCDU displays it immediately. Although the implant quickly sends its modified text to the MCDU, the original text briefly flashes on the screen, revealing that the implant is active.

To solve this problem, the implant needs to find out what the FMC is trying to send without letting the MCDU receive those messages. When the implant blocks a message by forcing the bus voltage to zero, current flows between the FMC and the implant's output amplifiers. The current is proportional to the voltage that the FMC is trying to apply to the bus and can be up to about 100 mA. The implant can use a current sensing transformer to measure the current and decode what the FMC is sending.

Commercially available current sensing transformers are larger than most other parts of the implant, so they may not be practical in implants that need to fit in small spaces. As an alternative to a current sensing transformer, an inline resistor with an amplifier might also work. The amplifier would need to tolerate common-mode voltages of $\pm$ 5 V that routinely appear on the bus.

Controlling the FMC The implant can control a relay that lets it send commands to the FMC, as shown in figure 4.10 (c). By simulating button presses, the implant can do anything that the

pilots can do. This is especially powerful when combined with the ability to change the content on the MCDU screen. The implant could secretly change the flight plan using these steps:

1. Record what the FMC is sending to the MCDU

2. Repeatedly send the same content to the MCDU, and block the FMC from sending anything. This effectively freezes the MCDU and prevents the pilots from seeing what the implant is doing.

3. Send button presses to the FMC that change a waypoint in the flight plan, or change the target altitude

4. Return the FMC to its previous mode so it displays the same content on the MCDU as before

If the change to the flight plan is subtle and the pilots are not paying close attention, the plane could end up deviating from its authorized route.

4.5 Defenses

Because there are many systems already in use that depend on ARINC 429, MIL-STD-1553 and other protocols, it is not practical to add new security features that require updating all existing hardware. There are still some possible defenses that can detect this attack.

Measuring voltage When an implant is actively driving the bus, the voltage on the bus is different from what the transmitting device wants. One way to detect this situation is to have an additional receiver connected to the bus near the transmitter. This receiver would measure the actual voltage on the bus. If the voltage the receiver senses does not match what the transmitter is trying to send, there is a problem. Some buses already use this feature. For example, CAN bus devices stop sending and report an error if the bus voltage is incorrect.

For ARINC 429, this would be fairly difficult to implement. The devices we have looked at have either a transmitter or receiver, but not both, connected to each bus. Devices would need hardware changes to add this capability.

Devices that use MIL-STD-1553 always have transceivers that can sense the bus voltage, so they may need only software additions or minor hardware changes.

Measuring current As described in section 4.4.3, the 429 implant drives about 100 mA of current through the bus when it is in conflict with another transmitter. That is significantly more current than during normal operation. An added device near each transmitter could measure the current flowing through the bus and detect unusually large amounts of current. This could be integrated into the transmitter or in a separate unit, as long as it is between the transmitter and the implant.

This is more practical than the previous defense because it can be added to existing buses without replacing any existing devices.

Future improvements Our method of changing the bus voltage works because we are dealing with transmitters that limit the amount of current they can supply. An active implant looks like a short circuit from the perspective of another transmitter. In response to a short circuit, a transmitter can limit its output current, heat up and enter thermal shutdown mode, or overheat and suffer physical damage. None of those options can defeat an implant, and the latter two are not appropriate for critical systems that need to be highly reliable.

Future systems could make this attack obsolete with features that check that each message was sent by an authorized device and has not been modified, maybe using cryptographic authentication.

4.6 Related work

Other MIL-STD-1553 attacks. Other researchers have investigated MIL-STD-1553 and proposed some attack methods. One project uses a similar threat model to this work, with a malicious device that an attacker connects to the bus [22]. The authors prototyped a device that can inject messages when the bus is idle. The injected messages may take precedence over earlier legitimate messages and change the behavior of the devices that receive them.

The effects of the injected messages and messages modified by an implant as described in this work are similar. Our approach is less likely to trigger an intrusion detection system on the bus because it does not need to inject any unexpected messages.

CAN bus attacks. The CAN bus has attracted plenty of security research because all modern cars use it, often for safety-critical communication.

In 2008, an overview of CAN security showed that a malicious device on the bus can change the behavior of motors and lights on a car [33]. After detecting a legitimate command, the malicious device immediately sends another message with a different command. The device processing the commands effectively ignores the legitimate command and follows the malicious command because it arrived later. That is similar to how our 429 implant modifies the text on the MCDU screen by receiving legitimate messages from the FMC and then immediately sending malicious messages with different content. The malicious CAN device could be added on to the bus with physical access to the wires like in this work, or it could be a factory-installed device infected with malicious firmware.

A later project showed how a device on a CAN bus can cause denial-of-service without sending any complete messages [43]. The authors prototyped an implant that identifies CAN messages from a target device and modifies those messages by briefly changing the bus voltage. The target device detects the voltage change as an error and stops sending the message. Their implant has several features in common with the two implants we prototyped for this work. All

three implants use software on a microcontroller to monitor the bus and control the output circuits that change the bus voltage. This lets them detect the beginning of a target message and modify later parts of the same message. The authors speculated that an implant could connect to the OBD-II port in a car to access the CAN bus and a power supply, like our 429 implant uses a connector to access ARINC 429 buses on a Boeing 737.

Around the same time, a different group of researchers made a similar device intended to block malicious messages [26]. This device can create a bus error that makes the malicious device give up on sending a message, like in [43]. The other devices on the bus see this as an error. More interestingly, in some situations their device can manipulate the bus so that the malicious device stops sending a message but other devices on the bus receive a normal zero-length message with no error. Like our implants, this device can sometimes seamlessly block messages.

Prior work uses standard CAN transceivers to drive the bus. The CAN physical layer uses open-drain circuits, which means that any number of nodes can safely concurrently assert the dominant state, but the recessive state can only occur when no node is active [36]—i.e., prior CAN implants can change a '1' bit to '0', but not the reverse. In contrast, Bus Driver implants use custom amplifier circuits that can make arbitrary changes to messages.

Voltage analysis defenses Many projects, most for CAN [54, 39] though also one for ARINC 429 [27], have proposed measuring the voltage over time on a bus and analyzing the readings to detect when a device has been added, removed, or substituted. This approach would be able to detect the type of attack described in this work. Even from casual inspection with an oscilloscope, a message modified by an implant looks different from an unmodified message.

The main problem with this approach is that it requires an analog-to-digital converter running at several million samples per second, and non-trivial computing resources to process the samples and detect anomalies. Practical deployments also must address natural variances, e.g. on vehicles shifts in temperature or primary battery voltage can cause voltage shifts which

fingerprinting must account for [20]. Models also need to be re-calibrated to prevent false positives when a device on the bus is replaced for benign reasons.

Timing defenses Based on a survey of CAN bus intrusion detection systems [40], many intrusion detection systems monitor the timing of messages and expect that all messages will be periodic. They generally do not monitor the content of messages, because that would require detailed knowledge of what each message means.

A similar intrusion detection system for ARINC 429 or MIL-STD-1553 would likely not be able to detect this attack. If the implant changes the content of messages but does not block or inject them, the timing of messages stays the same and does not provide any hint of a problem.

A proposed intrusion detection system for MIL-STD-1553 uses timing features, like the amount of time between messages and the precise delay before a device responds to a command from the bus controller, to find anomalies [29]. On a physically large bus where the implant is far away from other devices, and if an intrusion detection system is measuring time very precisely, the propagation delay through the bus might cause detectable timing differences.

Other buses To attack Trusted Platform Modules (TPMs) in desktop computers, one project described some ways to connect to an I^2C bus and interfere with communication [62]. These methods require disconnecting at least one bus wire to isolate the TPM.

4.7 Ethics

This chapter investigates the feasibility of attacks that amplify mere bus access into attacker-in-the-middle capability on industrial data buses (particularly ARINC 429 and MIL-STD-1553). The potential harms from this work arise from educating attackers interested in such environments, balanced against the benefits from understanding these risks and mitigating these threats in the future. In general, this balance favors publication because our work is speculative

in nature (i.e., we are unaware of any active attackers in the wild who would be enabled by this work) and any risks will be very much application dependent (i.e., there is no specific threat enabled by this work in general), but the understanding of the risk is one that has broad benefits.

In the one domain in which investigated a *specific* attack — overriding the ARINC 429 buses communicating between the GE FMC and MCDU on the Boeing 737 aircraft and doing so via a particularly accessible socket — we have provided advance disclosure to The Boeing Company. We first disclosed this information April of 2020 and have since provided multiple briefs and documentation to the company about the details of the attack and how we tested it in our lab. We have also chosen to omit certain details not critical to understanding the issue, but important to mounting the attack in practice (e.g., the name and location of the vulnerable access port and documentation of the proprietary MCDU protocol that we reverse engineered). Finally, while our end-to-end attack is significant (effectively allowing the attacker to misconstrue information to the pilot and the input to the FMC) we note that it does not directly impact the control surfaces and an able pilot can safely fly the aircraft with both units disabled.

4.8 Conclusion

We have shown how a small implanted device on a wired communication bus can block and modify messages, achieving attacker-in-the-middle capabilities without needing to physically disconnect the bus. This flexibility means that an attacker can install an implant in seconds on any accessible maintenance port and gain complete control over messages on the bus. Our proof-of-concept implants use high-current output amplifiers that can supply enough current to control the voltage on the bus and overpower other transmitters. The task of driving the bus is fairly simple for ARINC 429, where the implant and all other devices are connected directly. It is more difficult with MIL-STD-1553, because the implant can only access the bus through transformers. We described and demonstrated in a lab an attack on a Boeing 737 airplane that

can modify the information shown to the pilots on one MCDU and can secretly send commands to the FMC. Finally, we suggested some defenses that can detect this type of attack and may be practical to add on to existing systems without extensive design changes.

Chapter 4, in full, is a reprint of material under submission to USENIX Security 2023: Sam Crow, Stephen Checkoway, Patrick Mercier, Pat Pannuto, Stefan Savage, and Aaron Schulman. Bus driver: No-cut attacker-in-the-middle capability on aviation data buses. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Implementing High-Speed Scanning for Wireless Devices

5.1 Introduction

Scanning to find nearby wireless devices is an important tool for security testing and auditing. The results can reveal legitimate wireless devices that have been configured to be accessible when they should not be. On deployed systems, scanning can also find hidden malicious hardware.

Many devices communicate using classic Bluetooth. Conventional Bluetooth scanners are very slow, taking tens of seconds to find all the devices nearby. This slows down the security auditing process and makes it easy to miss devices.

We set out to solve this problem by using a low-cost software-defined radio to scan and find all the Bluetooth devices in range much more quickly than a conventional scanner.

5.2 Bluetooth scanning

For some protocols, a scanner only needs to listen on a single channel and decode the received signals. Scanning for devices using classic Bluetooth, however, is more difficult for two reasons. First, devices do not regularly send messages to announce that they exist. A scanner needs to send scan request messages first and then wait for responses from other devices. Second, Bluetooth uses frequency hopping across 79 different channels that span 79 megahertz of frequency [16]. Of the 79 channels, 32 are used for scanning. Most devices can only send and receive on one channel at a time, and a scanner can only communicate with another device if they are lucky enough to be using the same channel at the same time.

A conventional Bluetooth scanner periodically sends a message on one channel and then listens for a response on one other channel, and then moves to another pair of channels. The other devices listen on different channels at different times. Because the scanner and devices are not synchronized, they do not have any way to quickly end up on the same channel at the same time. Eventually, the timing works out and the scanner finds a device. This process can go on for tens of seconds before it finally discovers the last nearby device.

5.2.1 Multi-channel Bluetooth scanning

Our scanning method finds devices more quickly by sending scan messages on many channels at the same time and then listening for responses on many channels at the same time. Any device that is listening will respond, regardless of which channel it is listening on. As figure 5.1 shows, this significantly increases the rate of responses and reduces the time needed to find all the devices.

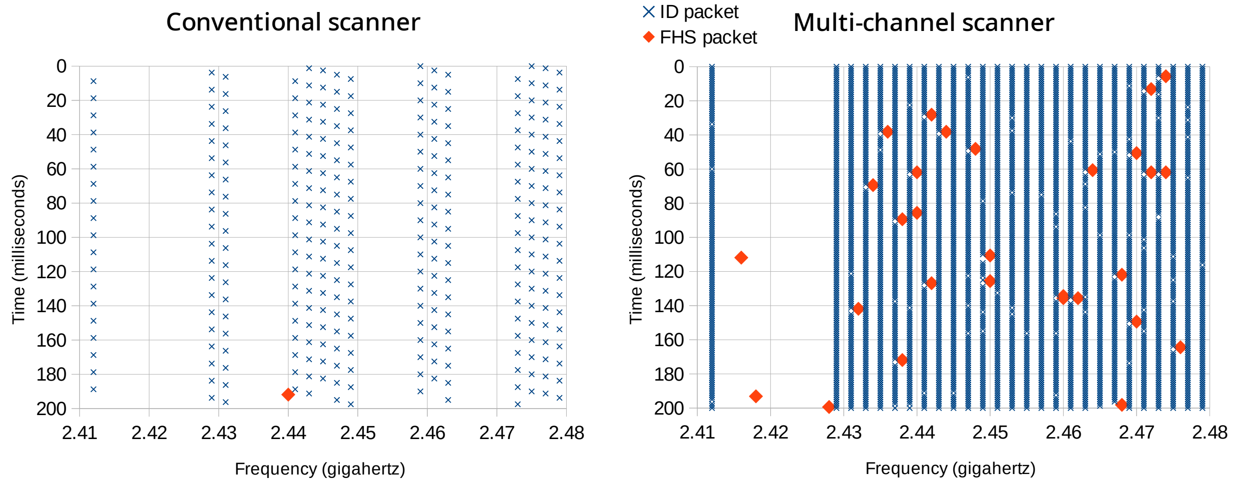

Figure 5.1: The messages sent and received by a conventional scanner and a multi-channel scanner over the same duration. Each blue X is an ID packet that the scanner sends. Each orange square is an FHS packet that a device sends in response. The multi-channel scanner sends packets on many channels at the same time and gets far more responses.

5.3 Hardware limitations

To implement multi-channel scanning, we needed a platform with wide bandwidth to transmit and receive on a wide range of frequencies. Many off-the-shelf Software-Defined Radio (SDR) devices would work, but they generally cost thousands of dollars.

To show that this scanning method works even on low-cost hardware, we decided to use an Analog Devices ADALM-PLUTO (”Pluto”) software-defined radio. The Pluto costs only $230, compared to thousands of dollars for other common SDRs. It includes analog filters and analog-to-digital and digital-to-analog converters, as well as a Field-Programmable Gate Array (FPGA) and an ARM processor. Normally, a separate computer controls the Pluto and does all the signal processing work. However, the processor on the Pluto runs Linux and can support custom signal processing software.

The Pluto has 512 megabytes of memory and two processor cores running at 667 megahertz. For comparison, that is half the memory, half the cores, and half the processor speed of a Raspberry Pi single-board computer. From previous experience processing and decoding radio

signals on a Raspberry Pi, it was clear that making the entire scanning process run smoothly on the Pluto would not be trivial.

5.4 Receiving using SparSDR

The normal way to receive a wide range of frequencies using an SDR is to run the analog-to-digital converter (ADC) at a high sample rate and then use software filters to separate the samples into different channels. A decoder for each channel detects messages and extracts their content. The filters and decoders all need to run constantly, even when there are no useful signals.

The Pluto's maximum sample rate of 61.44 million samples per second is enough to capture most of the Bluetooth advertising channels at the same time. The problem is that the built-in processor is too slow to filter and decode the resulting 245 megabytes of samples per second in real time.

To solve this problem, we took advantage of our previous project, SparSDR [35]. The first part of the process runs on the FPGA. We compress the samples from the ADC using a Fast Fourier Transform (FFT). This divides the signals into 1024 bins at different frequencies. The bins that have active signals above a threshold continue to the next step. On the processor, we use an inverse FFT to reconstruct the compressed samples back into the time domain. This produces one stream of samples for each Bluetooth channel that can be decoded normally.

The most important benefit of SparSDR is that the software does not have to do anything unless there is a received signal above the threshold. When there are more signals received, the software has to spend more time reconstructing and decoding them.

5.5 Scanner implementation

We implemented multi-channel Bluetooth scanning by customizing the software and FPGA on a Pluto software-defined radio. To send scan request messages, our software simply reads pregenerated samples from a file and sends them to be transmitted. To receive responses, we use SparSDR compression on the FPGA and then reconstruct and decode the signals in software.

5.5.1 Timing Challenges

The Bluetooth specification requires that messages are aligned to 625-microsecond time slots [16]. During scanning, the slots alternate between transmit slots, when the scanner sends messages, and receive slots, when the nearby devices send responses. The scanner can send messages as often as every 1250 microseconds.

If the scanner received signals all the time, it would receive the same signals it is transmitting and waste time processing them. To avoid this, the software needs to stop receiving while it is transmitting.

We found experimentally that we can get good scanning performance by transmitting every 10 milliseconds instead of the maximum rate of once every 1250 microseconds. We decided to make our scanner transmit for 312.5 microseconds and receive for the next 937.5 microseconds. The scan messages it transmits do not use up the full 625-microsecond slot. By switching to receive mode earlier, we are more likely to successfully receive responses if the timing is not perfect. For the rest of the 10-millisecond period, the software can process the received signals. Figure 5.2 shows a timeline of this process. This repeats until all the nearby devices have been found.

The easiest way to transmit samples periodically is to load the samples into memory and then repeatedly send the samples from memory to the digital-to-analog converter (DAC). If the software waits the correct amount of time before sending the samples each time, the messages

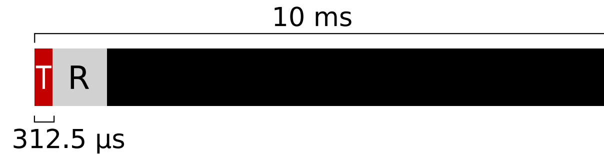

Figure 5.2: A timeline of the transmit and receive periods within a 10-millisecond scanning cycle

should be transmitted with the correct timing. The software can also use this process to stop receiving while it is transmitting.

Unfortunately, this method is not precise enough for two reasons. First, the functions that software uses to wait are inconsistent and sometimes wait much longer than desired. Second, there is a significant and variable delay after the software asks to send samples before the DAC starts running and the signals are actually transmitted.

I solved the first problem by making the transmit timing independent of the software. The software creates a buffer with exactly 1250 microseconds of samples and asks the Direct Memory Access (DMA) hardware to send those samples to the DAC repeatedly. This makes the scanner send messages exactly once every 1250 microseconds, and the software does not have to intervene.

That change makes the transmit timing correct, but it also means that the software does not know anything about the timing, so it has no way to stop receiving while transmitting messages.

To give the software information about the timing, we changed the FPGA configuration and added a simple timer that counts the number of transmitted samples. Every 312.5 microseconds, the timer sends an interrupt to the processor and the software can respond by enabling or disabling the transmit and receive modes. With this approach, unlike with a software timer, the software stays perfectly synchronized with the transmitted samples and never suffers from clock drift.

Because the interrupts are not aligned with the beginning of the transmitted messages, the software needs to make some adjustments. As figure 5.3 shows, misalignment can prevent the

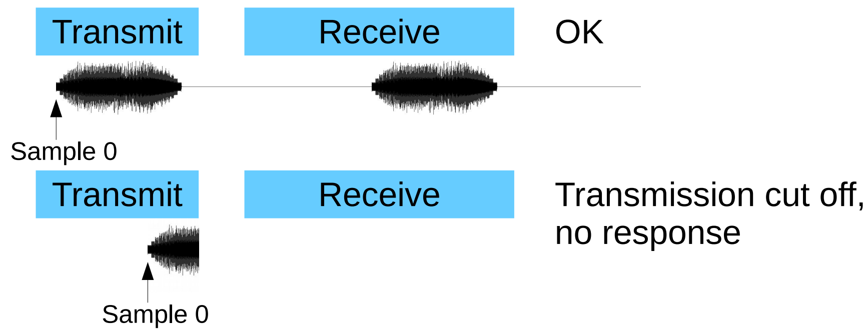

Figure 5.3: A timeline of what can go wrong if the sample interrupts are not correctly aligned. If the software enables transmit mode before or after the DMA sends sample 0 to the digital-to-analog converter, the transmitted messages get cut off and no devices respond.

scanner from correctly sending messages. The software periodically reads one of the DMA status registers to find what part of the sample buffer is being transmitted. It adjusts the offset of the timer interrupts so that it gets one interrupt exactly at the beginning of each transmit slot (just before the scan messages are transmitted). This lets the scanner transmit messages at the correct times and receive signals only when it expects responses from the nearby Bluetooth devices.

Because the software handles the interrupts is running in the user space of a non-real-time operating system, it sometimes misses interrupts when another process is using the processor. This makes the timing of a transmit or receive period incorrect, but the software can determine how many interrupts it missed and keep an accurate understanding of the time. Missed interrupts happen rarely and do not disrupt the timing of later transmit and receive periods.

5.5.2 Performance Challenges

The Pluto radio was designed to work with a separate host computer generating and decoding all the samples, so its built-in processor only needs to copy samples. For this project, we need to receive signals on 32 different channels, reconstruct the compressed samples, and decode them all using the Pluto's built-in processor. This all needs to happen in real time. It needs to handle responses from all the nearby devices, and also interference from non-Bluetooth signals.

A few years earlier, I set up a similar system using a different radio and a Raspberry Pi 3

B+ single-board computer. The Raspberry Pi had four processor cores running at 1.4 gigahertz and 1 gigabyte of RAM. With those computing resources, it was able to reconstruct and decode Bluetooth Low Energy messages on three different channels.

In comparison, the Pluto has only two cores running at 667 megahertz and 512 megabytes of RAM. With only a fraction of the resources and more channels to monitor, I did not know if it was possible to process all the signals in real time.

The early tests showed that the amount of available RAM was not a problem. Without any special optimizations, all the software used about 225 of the 512 megabytes of available RAM.

The performance of the processor, however, quickly turned out to be a significant problem. I started with a simple setup that kept the full receive chain enabled all the time. The software spent a long time reconstructing and decoding all the received signals. The receive chain became backed up within a few seconds and had to discard some of the received signals. We made several changes to improve the performance and keep the system working even when it gets temporarily overwhelmed.

The most important change was disabling the receive process during the 9.4-millisecond period when the scanner does not expect a response. Because the radio only receives signals for a fraction of the time, this eliminates most of the unwanted non-Bluetooth signals before the software processes them. The software then only needs to spend time on the actual scan responses and other signals that are received at the same time.

Later in the testing process, I noticed that the decoding step was usually the bottleneck that limited the throughput of the system, so I focused my efforts there.

The decoding software was intended to handle a stream of samples covering a wide range of frequencies. It does extra work to filter individual channels and adjust the rate of incoming samples. For this project, our reconstruction software produces a separate stream of samples for each channel with the correct sample rate, so the decoder does not need to do those steps. Another member of the team made a modified version of the decoder with the extra steps removed, which

worked more quickly.

We also realized that there might be a way to detect Bluetooth transmissions before the reconstruction step, to avoid running the reconstruction and decoding software on non-Bluetooth signals that our system cannot use.

The compressed samples that the FPGA produces have information about which frequency ranges have active signals. Each 1-megahertz Bluetooth channel is divided into 17 bins, each representing 20 kilohertz of bandwidth. I started by looking at the number of bins with active signals. Real Bluetooth signals show up in the middle of the channel and span about 8 bins (500 kilohertz). Narrow-band signals that only activate a few bins are usually noise, and signals that activate all 17 bins are usually Wi-Fi or something else that with a large bandwidth. I tested some simple heuristics that checked that the number of active bins was within the expected range. These methods partially worked, but had many false negatives, discarding signals that were actually decodable Bluetooth transmissions.

We later developed and tested a more complex method that detects the preamble at the beginning of each Bluetooth message. This method uses an FFT, making it more processor-intensive than the simple bin heuristics, but it is more accurate.

5.6 Evaluation

To check that the entire scanning process can run in real time without dropping samples, I set up a simple experiment. I put the Pluto and eight discoverable Bluetooth devices in a metal box, isolated from Wi-Fi and other external signals. I first ran the scanning software with all the Bluetooth devices off, so the software would not have any incoming signals to process. Next, I progressively turned on the Bluetooth devices to increase the rate of scan responses for the software to process. At each step, I recorded the total CPU and memory usage averaged over five minutes.

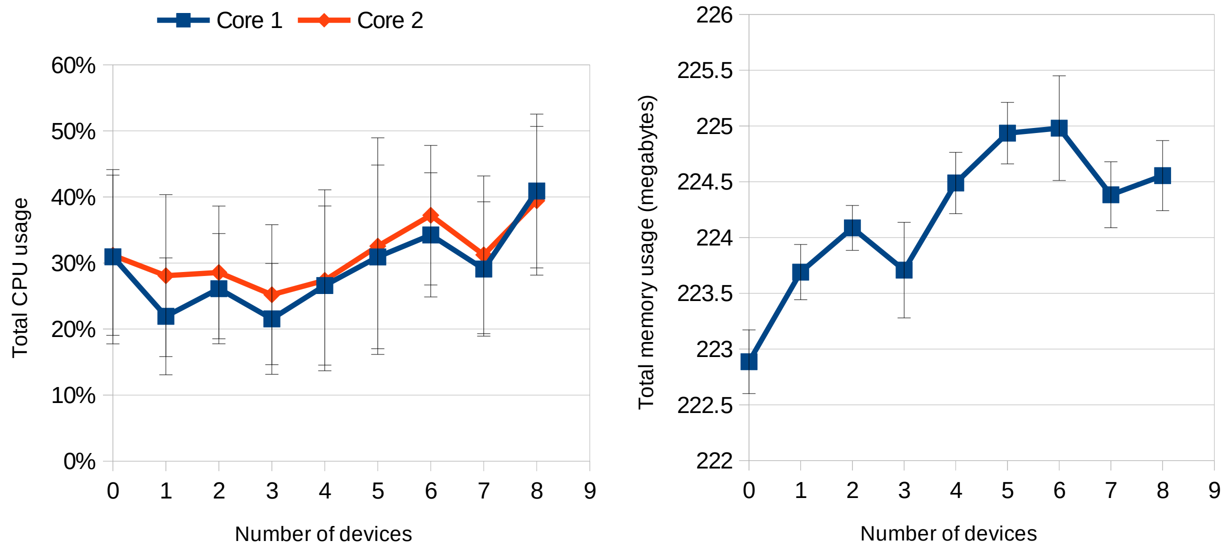

Figure 5.4: The total CPU (left) and memory (right) usage of all the scanning software with different numbers of discoverable devices. The error bars represent one standard deviation.

The total CPU usage increased only marginally with each additional enabled Bluetooth device. The variation of CPU usage over time was greater than the variation caused by increasing the number of enabled devices. With all eight devices enabled, the average CPU usage was only 40%.

The total memory usage remained within a range of two megabytes (223 to 225 megabytes) as the number of enabled devices increased. The maximum memory usage was still less than half of the 512 megabytes of total available memory. Figure 5.4 shows the detailed CPU and memory usage results.

5.7 Conclusion

We have developed a low-cost scanner that can find Bluetooth devices more quickly than conventional single-channel scanners. It transmits and receives on multiple channels at the same time so it can communicate with more devices sooner. By precisely timing when it transmits and receives, it follows the Bluetooth protocol and minimizes the performance impact of processing

non-Bluetooth signals.

This approach could be used with other protocols to make a multi-protocol scanner. By finding devices more quickly and consistently, our scanner can help make security auditing easier and more comprehensive.

Chapter 6

Conclusion and Future Directions

The previous chapters described three projects that help with security testing for complex cyber-physical systems. Chapter 3, *Flexible Avionics Testing*, explains how we combined physical, emulated, and simulated components in a flexible test setup. Chapter 4, *Bus Driver: No-cut Message Modification on Aviation Data Buses*, describes an attack that can also be used to test how devices respond to unexpected or modified messages. Finally, chapter 5, *Implementing High-Speed Scanning for Wireless Devices*, explains some challenges involved in multi-channel scanning for wireless devices using a software-defined radio.

Chapters 3 and 4 describe projects that are mostly complete and usable for testing. The Bluetooth scanning project in chapter 5, however, does the same job as any Bluetooth device but more quickly. For a scanner to be more useful than existing off-the-shelf devices, it needs to support many different wireless protocols beyond Bluetooth, including Wi-Fi and IEEE 802.15.4. We are continuing our work to enable more protocols. Depending on the number of protocols and channels, this might require a more capable software-defined radio receiver and a more powerful processor to decode the signals. Although this would increase the cost, the system would still cost less and use less power than a conventional software-defined radio approach.

For some tests, it might be useful to use two different testing methods together. For

example, a test could modify a message on a wired communication bus and monitor several channels of wireless communication to see how the system responds. If the test setup is large and involves testing devices spread out over a wide area, the testing devices could be interconnected and controlled from one place.

The testing methods described in this dissertation can help test the security features of complex cyber-physical systems. More effective security testing contributes to making transportation, communication, power distribution, and other infrastructure safer and more reliable.

Bibliography

[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., December 1998. ARINC Characteristic 739A-1.

[3] ARINC. *Loadable Software Standards*. Aeronautical Radio, Inc., January 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., August 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., January 2011. ARINC Characteristic 758-3.

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

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

[12] Marie Baezner and Patrice Robin. Stuxnet. Technical report, ETH Zurich, October 2017.

[13] Jorge Barrera. Hacker told F.B.I. he made plane fly sideways after cracking entertainment system. *APTN National News*, 2015.

[14] Nishant Bhaskar, Maxwell Bland, Kirill Levchenko, and Aaron Schulman. Please Pay Inside: Evaluating Bluetooth-based Detection of Gas Pump Skimmers. In *28th USENIX Security Symposium (USENIX Security '19)*, pages 373–388, Santa Clara, CA, August 2019. USENIX Association.

[15] Calvin Biesecker. Boeing 757 Testing Shows Airplanes Vulnerable to Hacking, DHS Says. *Avionics International*, November 2017.

[16] Bluetooth SIG. *Bluetooth Core Specification*, December 2016. Rev. 5.0.

[17] John Bull. You Hacked: Cyber-security and the Railways. *London Reconnections*, 2017.

[18] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *20th USENIX Security Symposium (USENIX Security '11)*, August 2011.

[19] Bo Chen, Vivek Yenamandra, and Kannan Srinivasan. Tracking Keystrokes Using Wireless Signals. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, May 2015.

[20] Kyong-Tak Cho and Kang G. Shin. Viden: Attacker Identification on In-Vehicle Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1109–1123, Dallas, TX, USA, October 2017. Association for Computing Machinery.

[21] Data Device Corporation. MIL-STD-1553 High-Reliability Three Stub Box Coupler. `https://www.ddc-web.com/en/about/news/press-releases/MIL-STD-1553_High-Reliability_Three_Stub_Box_Coupler_`.

[22] D. De Santo, C.S. Malavenda, S.P. Romano, and C. Vecchio. Exploiting the MIL-STD-1553 avionic data bus with an active cyber device. *Computers & Security*, 100, January 2021.

[23] Eurocontrol LINK 2000+ Programme. The LINK2000+ Test Facility Presentation. 2004.

[24] Bernard Ferguson, Anne Tall, and Denise Olsen. National Cyber Range Overview. In *Proceedings of the IEEE Military Communications Conference*, Baltimore, MD, USA, October 2014.

[25] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and Vulnerable: A Story of Telematic Failures. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*, Washington, D.C., August 2015. USENIX Association.

[26] Hristos Giannopoulos, Alexander M. Wyglinski, and Joseph Chapman. Securing Vehicular Controller Area Networks: An Approach to Active Bus-Level Countermeasures. *IEEE Vehicular Technology Magazine*, 12(4):60–68, 2017.

[27] Nimrod Gilboa-Markevich and Avishai Wool. Hardware Fingerprinting for the ARINC 429 Avionic Bus. In *European Symposium on Research in Computer Security 2020*, Guildford, UK, September 2020.

[28] Shyamnath Gollakota, Haitham Hassanieh, Benjamin Ransford, Dina Katabi, and Kevin Fu. They Can Hear Your Heartbeats: Non-Invasive Security for Implantable Medical Devices. In *Proceedings of the ACM SIGCOMM 2011 Conference*, Toronto, ON, Canada, August 2011.

[29] Sébastien J. J. Généreux, Alvin K. H. Lai, Craig O. Fowles, Vincent R. Roberge, Guillaume P. M. Vigeant, and Jeremy R. Paquet. MAIDENS: MIL-STD-1553 Anomaly-Based Intrusion Detection System Using Time-Based Histogram Comparison. *IEEE Transactions on Aerospace and Electronic Systems*, 56(1):276–284, February 2020.

[30] Brad Haines. Hackers + Airplanes: No Good Can Come Of This. Presented at DEFCON 20, 2012.

[31] Jon Hemmerdinger. Boeing delivered final commercial 737NG in January, ending 23 years of production. *FlightGlobal*, April 2020.

[32] Dustin Hoffman and Semon Rezchikov. Busting the BARR: Tracking "Untrackable" Private Aircraft for Fun & Profit. Presented at DEFCON 20, 2012.

[33] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security Threats to Automotive CAN Networks – Practical Examples and Selected Short-Term Countermeasures. In Michael D. Harrison and Mark-Alexander Sujan, editors, *Computer Safety, Reliability, and Security*, pages 235–248, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] IEEE. IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks–Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, 2021.

[35] Moein Khazraee, Yeswanth Guddeti, Sam Crow, Alex C. Snoeren, Kirill Levchenko, Dinesh Bharadia, and Aaron Schulman. SparSDR: Sparsity-Proportional Backhaul and Compute for SDRs. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, Seoul, South Korea, June 2019.

[36] Ronn Kliger and Sean Clark. APPLICATION NOTE AN-770: iCoupler Isolation in CAN Bus Applications. Technical report.

[37] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, Oakland, CA, USA, May 2010.

[38] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*, Washington, DC, USA, August 2015.

[39] Nathan Liu, Carlos Moreno, Murray Dunne, and Sebastian Fischmeister. vProfile: Voltage-Based Anomaly Detection in Controller Area Networks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, February 2021.

[40] Siti-Farhana Lokman, Abu Talib Othman, and Muhammad-Husaini Abu-Bakar. Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):1–17, July 2019.

[41] Jelena Mirkovic, Terry V. Benzel, Ted Faber, Robert Braden, John T. Wroclawski, and Stephen Schwab. The DETER Project: Advancing the Science of Cyber Security Experimentation and Test. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, Waltham, MA, USA, November 2010.

[42] Harry F. Olson and Everett G. May. Electronic Sound Absorber. *The Journal of the Acoustical Society of America*, 25(6):1130–1136, 1953.

[43] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206. Springer International Publishing, 2017.

[44] Phil Polstra and Captain Polly. Cyberhijacking Airplanes: Truth or Fiction? Presented at DEFCON 22, August 2014.

[45] Curtis Risley, James McMath, and Brian Payne. Experimental encryption of aircraft communications addressing and reporting system (ACARS) aeronautical operational control (AOC) messages. In *20th Digital Avionics Systems Conference*, October 2001.

[46] Aloke Roy. Secure Aircraft Communications Addressing and Reporting System (ACARS). In *20th Digital Avionics Systems Conference*, October 2001.

[47] Ruben Santamarta. *SATCOM Terminals: Hacking: By Air, Sea and Land*. IOActive, 2014.

[48] Ruben Santamarta. *A Wake-up Call for SATCOM Security*. IOActive, April 2014.

[49] Ruben Santamarta. *Last Call for SATCOM Security*. IOActive, August 2018.

[50] Ruben Santamarta. *Arm IDA and Cross Check: Reversing the 787's Core Network*. IOActive, August 2019.

[51] Securaplane. *Component Maintenance Manual with Illustrated Parts List, WELS Control Unit, Part No. 100-2601-01*. Securaplane, 2009.

[52] Securaplane. Wireless emergency lighting system, 2014.

[53] Matthew Smith, Daniel Moser, Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. Undermining Privacy in the Aircraft Communications Addressing and Reporting System (ACARS). In *The 18th Privacy Enhancing Technologies Symposium*, Barcelona, Spain, July 2018.

[54] Muhammad Tayyab. Authenticating the Sender on CAN Bus using Inimitable Physical Characteristics of the Transmitter and Channel. Master's thesis, University of Michigan-Dearborn, 2018.

[55] Teledyne Controls. PMAT 2000 System: Portable Maintenance Access Terminal 2000, November 2017.

[56] Hugo Teso. Aircraft Hacking: Practical Aero Series. Presented at Hack In The Box Security Conference, April 2013.

[57] United States Department of Defense. *MIL-STD-1553C: Digital Time Division Command/Response Multiplex Data Bus*, 2018.

[58] Chris Valasek and Charlie Miller. *Remote Exploitation of an Unaltered Passenger Vehicle*. IOActive, August 2015.

[59] David Van Cleave. Perspectives: RCAT-Tool to Achieve GATM. *Avionics Magazine*, pages 30–32, September 2003.

[60] Christopher B. Watkins. Modular Verification: Testing a Subset of Integrated Modular Avionics in Isolation. In *2006 IEEE/AIAA Digital Avionics Systems Conference*, pages 1–12, 2006.

[61] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, USA, December 2002.

[62] Johannes Winter and Kurt Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 65(5):748–761, March 2013.