ARINC-825TBv2: A Hardware-in-the-loop Simulation Platform for Aerospace Security Research

Derek Yu
Michael Vaquier
Evan Laflamme
Gabrielle Doucette-Poirier
Justin Tremblay
Brett H. Meyer
derek.yu2@mail.mcgill.ca
michael.vaquier@mail.mcgill.ca
evan.laflamme@mail.mcgill.ca
gabrielle.doucette-poirier@mail.mcgill.ca
justin.tremblay@mail.mcgill.ca
brett.meyer@mcgill.ca
Department of Electrical and Computer Engineering, McGill University
Montréal, Québec

ABSTRACT

CAN sees wide use in the aerospace industry, and is proposed as the link-layer implementation of the proposed Aeronautics Radio, Incorporated 825 protocol standard (ARINC-825). Unfortunately, CAN receives far less attention in aerospace than in the automotive domain; consequently, the aerospace security research community has been unable to assess the vulnerabilities of this protocol, due to a lack of access to data and real-time simulation infrastructure. We introduce ARINC-825TBv2, a hardware-in-the-loop link-layer simulation platform that allows aerospace security researchers to observe ARINC-encoded CAN traffic with realistic data. We present a novel, predictive attacker Gaslighter that we integrate into the testbench, and demonstrate its effectiveness. Furthermore, we implement an intrusion detection algorithm from the literature, Z-Score, and validate it using Gaslighter and realistic CAN data. We show that as the number of messages in the simulation increases, the accuracy of Z-Score drops to below that of a random predictor.

ACM Reference Format:
Derek Yu, Michael Vaquier, Evan Laflamme, Gabrielle Doucette-Poirier, Justin Tremblay, and Brett H. Meyer. 2019. ARINC-825TBv2: A Hardware-in-the-loop Simulation Platform for Aerospace Security Research. In *30th International Workshop on Rapid System Prototyping (RSP'19), October 17–18, 2019, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3339985.3358489

RSP'19, October 17–18, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6847-6/19/10…$15.00
https://doi.org/10.1145/3339985.3358489

1 INTRODUCTION

Controller area networks (CAN), are low-cost bus-based communication solutions, and are widely used in automotive, aerospace, and industrial automation systems. CAN was first introduced in 1986 by Bosch for use in cars, and has been adapted since for other markets. It is the basis for: ARINC-825, a proposed aerospace protocol standard; CANopen, with applications in medical equipment, maritime electronics, and railway applications; ISObus, which covers tractors and agricultural applications; and, others.

The CAN protocol was developed before cybersecurity was a concern in electronic system design, and as such lacks many of the standard features that exist in other link-layer protocols, such as Ethernet. While CAN employs CRC for message integrity, it does not support encryption, source authentication, etc. Consequently, there is tremendous need today to assess CAN, and any higher level protocols built on top of it, for security vulnerabilities.

While research on attacks on and countermeasures in automotive CAN has experienced tremendous growth, aerospace systems have received relatively little attention. As a consequence, aerospace researchers lack access to the platforms needed to conduct security research, principally, a real-time simulation platform for (a) data collection, (b) vulnerability assessment, and (c) validation of mitigation techniques. Existing data collection strategies use application-level data, but are unable to replicate link-layer level communication. However, simulation at the link-layer is critical: automotive system attacks show that key vulnerabilities exist in CAN and related interfaces with electronics. As ARINC-825 exists at a higher level of abstraction, it too is potentially vulnerable at the link-layer.

We therefore propose ARINC-825TBv2, a hardware-in-the-loop simulation platform for aerospace cybersecurity research. The ARINC-825TBv2 platform implements the ARINC-825 protocol and simulates how it encodes and decodes realistic flight data transmitted between different actors over CAN. Realistic data is provided by FlightGear, an open-source flight simulator [3]. The testbench manages the exchange of this data with models of aerospace electronics,

which communicate amongst themselves over CAN (virtual or physical), and FlightGear, in a closed feedback loop. The testbench has been carefully designed to be extensible, facilitating the development of attackers that mimic aerospace subsystems, and system monitoring, enabling the implementation of mitigation strategies.

This combination of features makes it uniquely possible for researchers to quickly implement and evaluate attacks and corresponding mitigation strategies, testing them against realistic flight data, running on real-time, virtual, or physical CAN buses. The ARINC-825 protocol is currently only a draft, and no hardware implementations exist to our knowledge. The ARINC-825TBv2 implements the proposed application protocol entirely through software, although the underlying link-layer CAN traffic supports both hardware and software implementations. Other aerospace simulation frameworks exist, but none of them incorporate hardware-in-the-loop compatibility and link-layer message resolution.

The rest of this paper is organized as follows. We briefly introduce existing data collection, attack, and intrusion detection methodologies in Section 2. We then present our proposed platform in Section 3. The following sections present our novel attack, Gaslighter, and how we integrated it (Section 4) and a state-of-the-art intrusion detection mechanism, Z-Score [2] (Section 5).

2 RELATED WORK

In embedded system security intrusion detection research, three main areas are important: data collection or simulation, attacks on aerospace systems, and intrusion detection systems. In the aerospace community, data predominantly comes from the US National Aerospace System flight recordings, which is generally unavailable to researchers, or the X-Plane flight simulator. X-Plane uses Monte Carlo methods to simulate random flight variation, and has been used to simulate the detection of anomalous flight behavior [9]. There are no related investigations of attacks on aerospace systems, however; instead, the focus is on the detection of anomalies leading to aircraft failure [7].

In broader CAN community, the story is radically different. As automobiles are ubiquitous, researchers armed with a CAN reader and a laptop can easily gather CAN data straight from the OBD-II port of a car [2, 10, 11]. Simulations have also been proposed, such as the J1939 CANoe framework [6], and OCTANE, which not only simulates CAN traffic but also LIN and FlexRay [5].

There have also been a plethora of validated attacks on cars, using a variety of chained exploits on weakly defended embedded systems to gain access to the CAN bus [8]. However, most attacks use an augmented, pre-recorded dataset, with attacks that fall into the following categories: dropped packets, injected control packets, and injected data packets [1, 2].

Intrusion detection techniques in CAN typically focus on system invariants: voltage patterns, message timing, and the steady-state behavior of the system [4, 10]. Message timing is particularly popular, with a focus on using inter-message arrival times as a means to detect malicious behaviour [6].

Existing CAN security research has proven without a doubt that CAN is vulnerable to attack. Given that aviation networking communication protocols use CAN as an underlying resource, we are certain that attacks can occur in aircraft. However, existing CAN security research can not be directly applied to aerospace systems because the ARINC-825 application abstraction layer, which interfaces between aircraft data and CAN data. Our proposed testbench uniquely meets this need by simulating CAN at the link-layer with real-time, ARINC-encoded, flight data.

3 ARINC-825TBV2

The ARINC-825TBv2 platform, illustrated in Figure 1 consists of three key components:

(1) FlightGear, an open source flight simulator; FlightGear satisfies the requirement that we simulate using realistic data.
(2) ARINC-825 message profiles, or *line replaceable unit* (LRU) profiles; LRU profiles model software tasks that produce and/or consume data from FlightGear and other LRUs; and,
(3) A custom testbench, which sources flight dynamics data from FlightGear, and other signals from LRUs, and implements the ARINC-825 encoding scheme for all communication.

The simulation implements only the ARINC-825 specification; in software, CAN traffic is abstracted away using the python-can library, which interacts directly with the Linux networking stack, of which CAN is natively supported. In hardware, CAN traffic support is provided by existing CAN-compliant microcontrollers that interact using a standard physical CAN bus. Additional support is provided for coordinating communication across redundant buses, as recommended by the ARINC-825 draft standard.

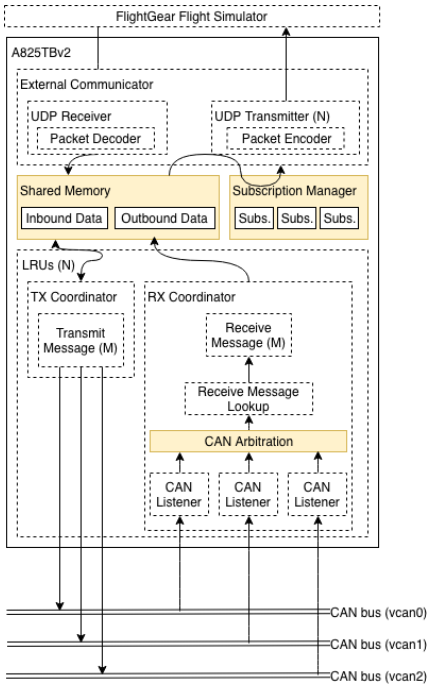


Figure 1: ARINC-825TBv2 integrates the FlightGear flight simulator and LRU profiles. FlightGear communicates with LRUs over UDP; LRUs communicate with each other over CAN. Within the testbench, data is exchanged asynchronously.

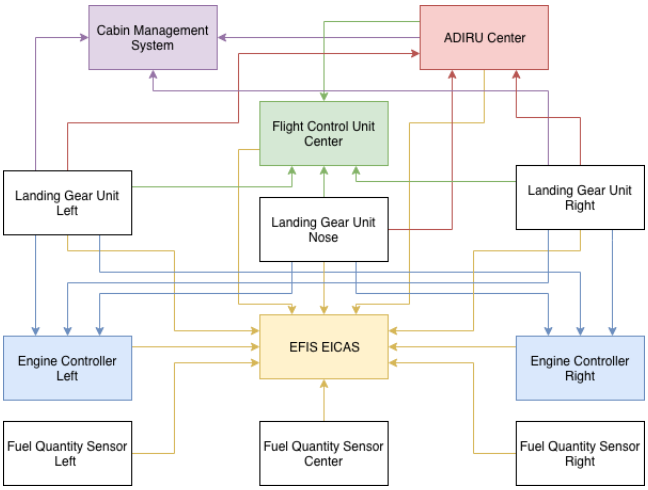


Figure 2: Every LRU which consumes a message from another LRU is colored; edges from sources of consumed messages have the same color as the LRU in question.

3.1 FlightGear

FlightGear [3] is a sophisticated, open-source, flight simulator. All state is maintained internally in FlightGear in its *property tree*, e.g., the position of the airplane, data related to weather simulation, and control state. Information from the property tree is used to drive the *flight dynamics model*, the fundamental control loop of the simulation. FlightGear periodically updates this internal state; every 33ms, this internal state is published to the testbench over UDP, and every 33ms, we aggregate a new global state based on the testbench ARINC-825 traffic and push this data directly to FlightGear's internal state. Internally, the testbench stores FlightGear's state in Shared Memory, which is periodically consumed by LRUs. These LRUs exchange ARINC-825 encoded traffic, and the resulting decoded traffic across all LRUs is what is written back to FlightGear. In this manner, both FlightGear and the testbench are receiving and transmitting UDP packets at 30Hz.

3.2 LRU Profiles

We model *line-replaceable units* (LRUs), aerospace electronic control subsystems, using profiles that describe their function (e.g., altimeter sensor, or engine speed actuator) and the characteristics of their data payload, such as: frequency of communication, communication data-type, and fields related to arbitration.

We model the *air data inertial reference unit* (ADIRU), *electronic flight instrument system* (EFIS) *engine-indicating and crew-alerting system* (EICAS, a cockpit instrument panel), engine controllers, landing gear, and fuel sensors. These subsystems exchange information with each other, as illustrated in Figure 2. Data originating at LRUs is sourced from FlightGear through the Shared Memory module (using UDP). Data destined for FlightGear from LRUs also passes through the Shared Memory module. The different LRUs each send and/or receive one or more periodic messages. This communication is initiated on new data anywhere from every 40 to every 100 ms; the messages vary in payload from 4 to 8 bytes.

3.3 Testbench Data Flow Management

Data flow between components of the testbench is facilitated by two modules, the *Shared Memory*, and the *Subscription Manager*. The Shared Memory module acts as the interface between the external and embedded components of the testbench, and eliminates data race problems by duplicating state during read and write operations. As noted above, Shared Memory receives FlightGear state every 33ms, and is updated by LRUs asynchronously.

The Subscription Manger listens to all transactions that occur at the Shared Memory module in order to collect information used for tracking the behavior of the testbench, e.g., monitoring and logging. In addition to supporting debugging and performance monitoring, the Subscription Manager is the key to modeling intrusion detection in ARINC-825TBv2. Subscriptions are delivered in the form of *News* objects that encapsulate the desired data. News is delivered to subscriptions in real-time using a combination of asynchronous queues and batch processing. Each subscription has its own queue. We implement support for two types of subscriptions: low and high performance. Low-performance (i.e., low frequency) subscriptions are managed by the same process running the testbench.

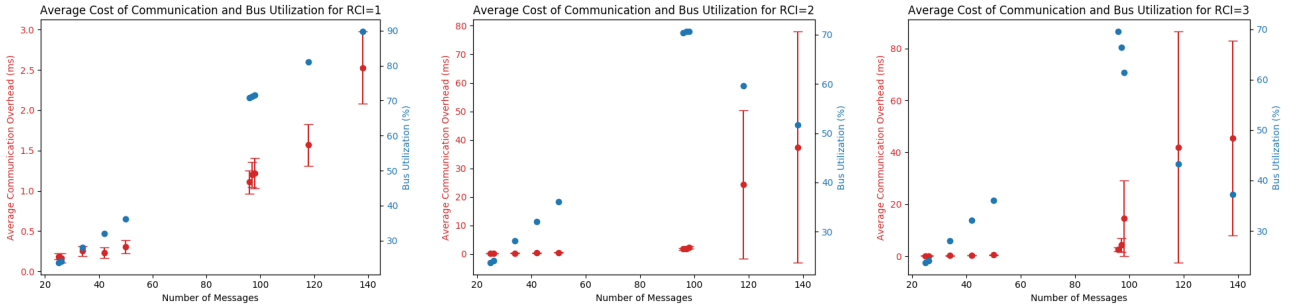
High-performance (i.e., high frequency) subscriptions share their own independent process that communicates with the host process. Without this separation, a number of problems arise: the Subscription Manager begins to interfere with the performance of the testbench because of the number of calls to enqueue, dequeue, and process News; queues begin to fill; the testbench runs out of memory, and ultimately the system crashes. Giving high-performance subscriptions their own process allows them to consume News nearly as quickly as it is produced. Because of the overhead of encapsulating objects for interprocess communication (*pickling*, in Python), News objects are packaged and unpackaged in batches.

3.4 Message Timing

When the success of failure of an attack (or mitigation technique) depends on the timing and data of the malicious messages, it is critical that these systems operate with the highest fidelity possible. This is particularly important for CAN-based systems, as CAN is arbitrated, not scheduled, and differences in simulated vs real message timing re-orders message arrival, potentially affecting intrusion detection: one common way to identify attacks is by looking at message interarrival times [2]. However, since FlightGear state is captured every 33 ms, if it takes the testbench longer than 33 ms to produce all the necessary CAN messages related to this state snapshot, data races are possible.

We performed an experiment to validate that our simulation runs quickly enough to allow system components (e.g., FlightGear and LRUs) to interact in real-time simulating a CAN bus operating at 250 Kbps. We varied the number of active LRUs in the system to measure the average communication overhead of communication using our asynchronous subscriber model. This overhead includes: reading from and writing to the Shared Memory module; ARINC-825 encoding and decoding; python-can library calls; and, data arbitration across multiple redundant buses.

Experiments were conducted on a system with a 2.8 GHz, four-core Intel Xeon E5-1603 v4, with 16 GB RAM, and Nvidia Quadro K1200. We ran FlightGear 2019.1 under Ubuntu 18.04 LTS. Each



(a) One bus. **(b) Two redundant buses.** **(c) Three redundant buses.**

Figure 3: Real-time behavior is possible for a wide range of bus utilizations when simulating a single bus. However, as the number of redundant buses increases, data value arbitration becomes more costly; simulating 100 periodic messages or more is not recommended.

trial was run for 200 seconds, which includes airplane startup, and takeoff. Measurements of average communication overhead and bus utilization are averaged the entire 200 second trial.

The results of these experiments are illustrated in Figure 3(a)-(c). For a single bus (Figure 3(a)), average communication overhead remains low–less than 3 ms–even when the number of active messages reaches 140. Above 100 messages, however, bus utilization rises above 50%, the recommended maximum for ARINC-825; higher utilization risks high priority messages being delayed in arbitration. Regardless, real-time simulation is clearly possible in this case, with average message overhead representing less than 10% of even the most aggressive messages timing budget.

When two or three buses are simulated (Figures 3(b) and Figure 3(c) respectively), the overhead of (a) waiting for all redundant copies to be received, and (b) deciding what value to use, contributes significant delay to communication, on average: with 50 active messages, average communication overhead rises above 20 ms, compared with less than 1 ms for a single bus. This delay may begin to inhibit real-time simulation, depending on the sorts of computational tasks that are modeled; in this work, LRUs do no computation of their own, but only forward data from FlightGear to other LRUs, or vice versa. Furthermore, while ARINC-825 supports up to four buses, real-time simulation of such complex systems, may require further work streamlining the process of arbitration. We would expect arbitration to scale better in hardware.

4 GASLIGHTER, A PREDICTIVE ATTACKER

Gaslighter listens for ground truth, changes it, and attempts to preempt the transmission of future ground truth; in the best case, this results in systems that observe both the true and falsified data. In the worst case, only the falsified data is observed by the system.

4.1 Attack Methodology

An overview of *Gaslighter* is illustrated in Figure 4. First, Gaslighter observes CAN network traffic, and identifies a stream between the source LRU to be compromised, and the destination LRU to be targeted. For example, fuel quantity data exchanged between the fuel sensor and the EFIC EICAS (Figure 2). Gaslighter measures the inter-message arrival time for messages in the stream, and decodes and records the data of those messages. Gaslighter then predicts the timing of the next legitimate message (using the past two observations only), and places a malicious message on the bus *before* the next valid message.

In the case of regular messages, the target will accept the malicious message, as well as the subsequent legitimate message, potentially resulting in rapid, confusing variation in flight data. In the case of high-integrity messages, the target will accept the malicious message, and ignore the legitimate message. This is facilitated by decoding the sequence number (SNo), which is used to order high-integrity messages. High-integrity messages are only consumed in order; when Gaslighter sends a malicious messages with SNo = n prior to the legitimate message with the SNo, the legitimate will be discarded as erroneous.

4.2 Implementation

Gaslighter is implemented in ARINC-825TBv2 in a similar way to LRUs, with the key difference that rather than coordinate with Shared Memory to observe changes in FlightGear application state, it maintains its own Gaslit memory, and other functionality for tracking SNo, and carefully timing message transmission. These differences are illustrated in Figure 5.

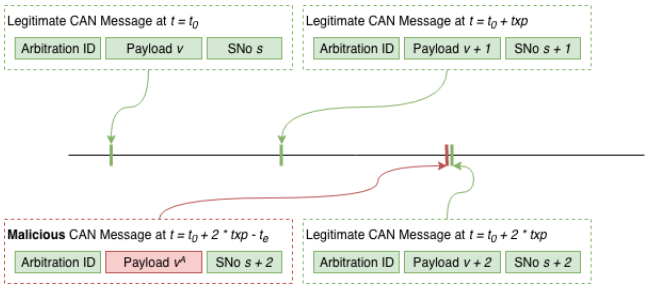


Figure 4: *Gaslighter* observes past messages in order to place a malicious message with valid encoding and correct SNo onto the CAN bus *before* the next legitimate message.

Gaslighter implements a flexible framework for manipulating flight data. Given a past observed value v_i, Gaslighter will transmit malicious value $m = K_1 + K_2 v_i + K_3(v_i - v_{i-1})$. This captures a wide variety of potential behavior intended to deceive operators or other equipment, from offset attacks (K_1), to attacks on rate of change (K_2 and K_3).

4.3 Experimental Setup

We conducted experiments to test Gaslighter's effect on flight data transmitted over ARINC-825. We attacked regular and high-integrity messages; the testbench was run on the same hardware configuration described in Section 3.4.

Attacks were conducted during the takeoff of a flight from one side of mainland Hawaii to the other. The target aircraft, a Boeing 757-200, begins on the ground; we manually performed the takeoff procedure. Once the aircraft has taken off and has reached a minimal altitude, the autopilot is engaged, bringing the 757 to a cruising altitude of 10,000 feet. To reduce the effect of human variation in take-off procedures, *Gaslighter* begins 100 seconds after simulation initialization, *after* the autopilot has been engaged. *Gaslighter* then attacks for another 100 seconds before disengaging. The simulation is concluded after another 40 seconds and the logs are written to file. The total simulation time from beginning to end is 240 seconds.

To test the effectiveness of Gaslighter against regular messages, we adopt the attack the communication of altimeter data between the ADIRU and the EFIS EICAS; we subsequently attack the communication between the left fuel quantity sensor and the EFIS EICAS.

4.4 Results

We performed 16 experiments varying the magnitude of K_1, K_2, and K_3 for both regular and high-integrity messages; for the sake of brevity, we only include the most interesting five, three for regular messages, and two for high-integrity messages. These results are illustrated in Figures 6(a)-(c) (attacks on regular messages) and Figures 7(a)-(b) (attacks on high-integrity messages) respectively.

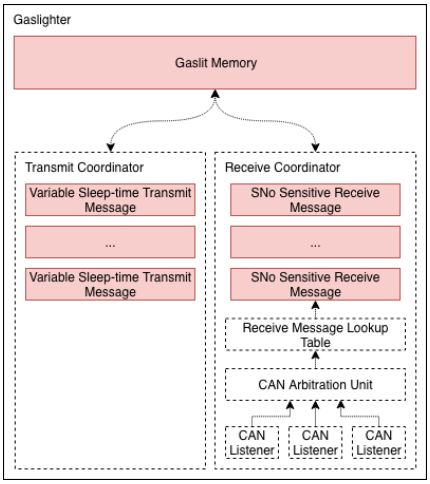


Figure 5: Gaslighter is implemented like an LRU, but does not interact with FlightGear, only the targeted LRUs.

Figure 6 illustrates the effect of attacks on altimeter data. In Figure 6(a), we observe a vertical translation in the value of the altitude (by varying K_1). In FlightGear, this attack will manifest itself as the flickering of the altimeter between the two values (one legitimate, one by *Gaslighter*), both of which respond to changes in altitude equally. When the attack changes the slope of increase in altitude (by varying K_2), the result looks like that in Figure 6(b), where the falsified altitude value increasingly diverges from the actual value as a function of time: one set of data follows the actual takeoff altitude of the aircraft, and the second shows that the plane is climbing at a far shallower angle of attack.

Varying K_3 (Figure 6(c)) appears to have a similar effect to varying K_1. However, the stratification of values is increased, and there appears to be noise injected into the stream of malicious altimeter values. We suspect that numerical stability in the calculations made by Gaslighter is responsible for this noise.

For attacks on high-integrity messages (Figures 7(a)-(b)), we see that while the node is under attack, the operator is only able to observe the malicious payloads; legitimate messages are ignored. As a result, the observed fuel quantity drops continuously and rapidly. The main exception is when the system occasionally resynchronizes to the correct data stream. This occurs whenever Gaslighter mispredicts the appropriate timing of its attack, and transmits *after* the correct data. This could be mitigated by giving Gaslighter more sophisticated timing analysis capabilities; at the moment, it only considers the last two messages to determine the time to insert its malicious data. As bus utilization increases, the need for more sophisticated timing analysis also increases.

Another anomaly occurs in Figure 7(a), when the malicious value appears constant (starting at $t = 100$). We suspect this has occurred by Gaslighter improperly synchronized with the target's SNo; as the SNo has a maximum value of 255, this is quickly resolved.

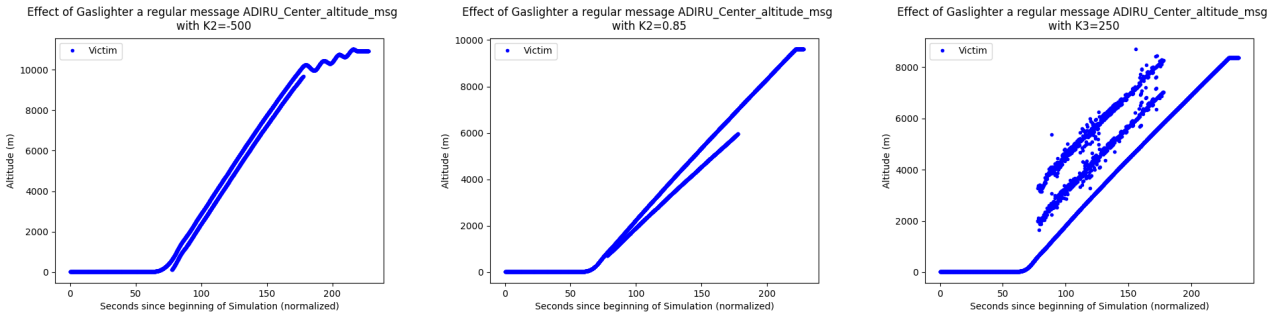
5 IMPLEMENTING Z-SCORE

The key value of our testbench is that different components can be easily integrated into the platform. In the previous section we demonstrated a novel link-layer attack on application-layer data, *Gaslighter*; in this section, we present an implementation of a state-of-the-art intrusion detection scheme, Z-Score [2], and use it to attempt to detect Gaslighter. Z-Score has been used previously to attack synthetic data; ours is the first demonstration of it on realistic, real-time data.

5.1 Z-Score

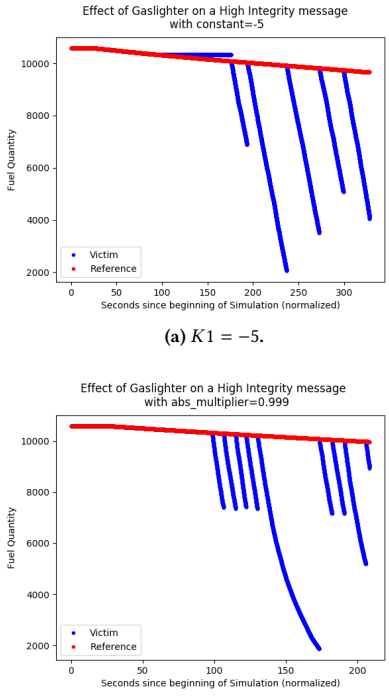
Z-Score tracks message interarrival time in order to detect intrusions in CAN-based systems. CAN is a broadcast medium, and any node may transmit any message with any arbitration ID. As such, one natural way to attack a CAN-based system is to masquerade as a node and transmit falsified data (a la Gaslighter). This can be effective because one policy that ARINC-825 suggests for data arbitration is "first good data;" in this case, as long as the attacker preempts the targeted node, its data will be accepted.

However, unless the targeted node is first disabled, e.g., using a targeted denial-of-service attack [1], the attacked arbitration ID will be appear more often than expected (e.g., twice per 40 ms, rather than once); this can be coarsely monitored to determine when an



(a) $K1 = 500$. **(b)** $K2 = 85$. **(c)** $K3 = 250$.

Figure 6: When Gaslighter attacks altimeter data, a *regular* message, the result is manifest as flickering on the EFIS EICAS: both values, real and attacked, appear.



(a) $K1 = -5$.

(b) $K2 = 0.999$.

Figure 7: When Gaslighter attacks fuel quantity data, a *high-integrity* message, the EFIS EICAS usually displays falsified data; at times, the flight system (briefly) resynchronizes with the correct data.

attacking node attempts to inject falsified data. Z-Score observes the arrival time of each message in a sequence of non-overlapping windows of communication. When the average interarrival time for a given arbitration ID deviates from that of the average interarrival time for all messages in the window, by more than a pre-defined threshold, Z-Score flags the window as having an intruder.

5.2 Implementation

We implement Z-Score as a high-performance subscription (see Section 3.3). Each non-overlapping window contains a specific number of News objects. The batch of News is first separated by parameter name (such as altitude, fuel gauge, or longitude). Each News also includes a timestamp corresponding to when it was received. These separated sub-batches then have their timestamps differenced to produce their average inter-message arrival times, x. Given the average inter-message arrival time across all messages in the window, μ, the Z-score of a message is $Z = (x - \mu)/\sigma$, where σ is the standard deviation of the interarrival times of all messages in the window. If this ratio exceeds a pre-defined threshold factor of Th, then that message is flagged as attacked.

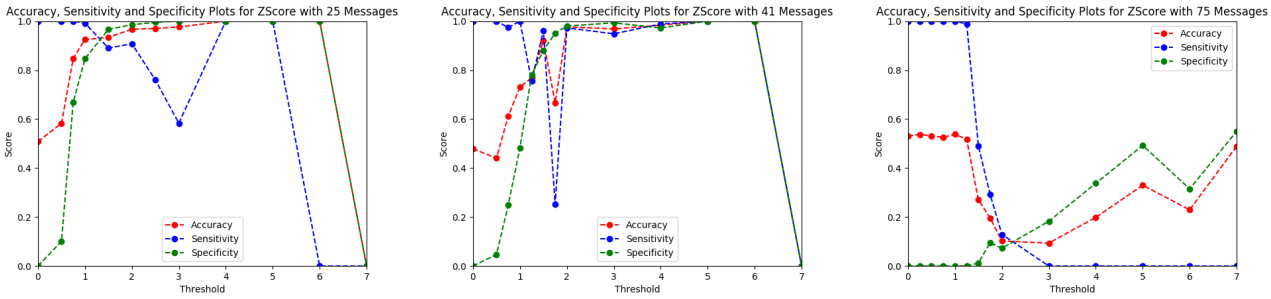
5.3 Experimental Setup

We conducted experiments to test Z-Score's capacity to detect Gaslighter attacking flight data transmitted over ARINC-825. We used the same hardware configuration described in Section 3.4. Each trial we conducted was divided into four stages. In the first stage, the simulation is active but intrusion detection is not. In the second, the intrusion detection subscription is active, but Gaslighter is not. In the third, Gaslighter is attacking the simulation; in the fourth, Gaslighter is disabled again. Stage 3 lasts for 100 seconds.

The observational period for Z-Score is set to 0.1 s; we varied the threshold Th from 0 to 7, as in literature [2]. We explore low thresholds with higher resolution than higher thresholds in order to carefully document changes in performance. We varied the number of messages N from 25 (the smallest possible, representing messages from just two LRUs), to 41, to 75.

The performance of Z-Score has been previously evaluated *accuracy*, *sensitivity*, and *specificity*, which, given the rate of *true positive*, *false positive*, *true negative*, and *false negative*, are defined as:

$$\begin{aligned} \text{accuracy} &= \frac{TP + TN}{TP + FP + TN + FN}, \\ \text{sensitivity} &= \frac{TP}{TP + FN}, \\ \text{specificity} &= \frac{TN}{FP + TN}. \end{aligned} \tag{1}$$



(a) $N = 25$. **(b)** $N = 41$. **(c)** $N = 75$.

Figure 8: Z-Score performs about as expected when the number of active messages is low. However, as the number of active messages, and resulting bus utilization increases, the approach fails to outperform a policy of randomly classifying messages as attacked or not.

In general terms, *accuracy* is the rate of correct overall classification, including both messages that are attacked and those that are not. *sensitivity* is the rate of detection of attacks; *specificity* is the rate at which messages that aren't attacked are identified as such.

In the literature, when evaluating using synthetic CAN data, the accuracy and specificity curves of Z-Score tend to rise rapidly from 0%, when $Th = 0$, to 100%, where they hold relatively stable: as the threshold increases, Z-Score does a better job of identifying messages that have not been attacked, and given that these messages dominate, this drives up overall accuracy. The threshold must be high enough to separate attacks from background noise in interarrival times. Sensitivity, on the other hand, tends to begin at 100%, and decrease exponentially to 0%: as the threshold increases, Z-Score's ability to distinguish attacked from non-attacked messages decreases. Likewise, the threshold must be low enough that the variation between messages under attack and not is measurable.

5.4 Results

Our results for simulations with 25, 41, and 75 active messages are illustrated in Figures 8(a)-(c). We observe that when the number of messages, and resulting bus utilization, is low Z-Score performs relatively well. Accuracy begins at about 50%, indicating that attacks introduce sufficient deviation in inter-message arrival time that the approach can do better than random guessing even when the threshold is small. Specificity also rises rapidly to 100%; sensitivity falls, but in general not until the threshold is high.

When $N = 75$, and bus utilization is approximately 50%, Z-Score breaks down. Accuracy falls, after holding steady at about 50%; when $Th > 1.25$, random guessing is expected to outperform Z-Score on this metric. Sensitivity likewise falls off rapidly; specificity never rises above about 50%. In short, Z-Score, in the presence of the naturally occurring timing variability of a highly utilized real-time system, cannot distinguish friend from foe.

6 CONCLUSIONS

We have presented ARINC-825TBv2, a configurable, hardware-in-the-loop link-layer simulation platform for aerospace security researchers, which uses realistic data from FlightGear, an open-source flight simulator, and ARINC-825 profiles provided by industry partners, to produce realistic aerospace CAN traffic simulations. ARINC-825TBv2, within the confines of the ARINC-825 draft specifications, has real-time capability with redundant buses, making it possible to integrate, test, and evaluate cyber-attacks and countermeasures.

We also introduced *Gaslighter*, a novel predictor attack that leverages the ability to decode ARINC-traffic and to preempt legitimate CAN packets targeted at any LRU. Gaslighter is successfully able to attack regular as well as high-integrity messages in real-time on a CAN bus. Finally, we implemented the state-of-the-art intrusion detection algorithm, Z-Score, to determine its effectiveness against Gaslighter. We observed that when the system implements relatively few messages, resulting in low bus utilization, Z-Score works as presented in the literature. However, when ARINC-825TBv2 is configured with >75 messages, we observe that Z-Score is unable to predict with an accuracy greater than that of a random policy.

REFERENCES

[1] A. Palanca et al. [n. d.]. A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks. In *DIMVA 2017*. 185–206.
[2] A. Tomlinson et al. [n. d.]. Detection of Automotive CAN Cyber-Attacks by Identifying Packet Timing Anomalies in Time Windows. In *DSN-W 2018*.
[3] C. L. Olson, et al. 2018. About FlightGear. https://www.flightgear.org/about/. (2018).
[4] Kyong-Tak Cho and Kang G. Shin. 2016. Fingerprinting Electronic Control Units for Vehicle Intrusion Detection. In *USENIX Security 2016)*.
[5] Christopher E. Everett and Damon McCoy. 2013. OCTANE (Open Car Testbed and Network Experiments): Bringing Cyber-Physical Security Research to Researchers and Students. In *CSET 2013*.
[6] B. Groza and P. Murvay. 2019. Efficient Intrusion Detection With Bloom Filtering in Controller Area Networks. *IEEE Trans. Inf. Forensics Security* 14, 4 (April 2019).
[7] V. M. Janakiraman and D. Nielsen. 2016. Anomaly detection in aviation data using extreme learning machines. In *IJCNN 2016*.
[8] K. Koscher et al. 2010. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*.
[9] A. Nanduri and L. Sherry. 2016. Generating Flight Operations Quality Assurance (foqa) data from the X-Plane Simulation. In *2016 Integrated Communications Navigation and Surveillance (ICNS)*. 5C1–1–5C1–9. https://doi.org/10.1109/ICNSURV.2016.7486355
[10] H. M. Song, H. R. Kim, and H. K. Kim. 2016. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *ICOIN 2016*.
[11] H. Suda, M. Natsui, and T. Hanyu. [n. d.]. Systematic Intrusion Detection Technique for an In-vehicle Network Based on Time-Series Feature Extraction. In *ISMVL 2018*.