5-2016

Prevention of Drone Jamming Using Hardware Sandboxing

Joshua Mead
University of Arkansas, Fayetteville

Prevention of Drone Jamming Using Hardware Sandboxing

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Joshua Mead
University of Arkansas
Bachelor of Science in Computer Engineering, 2014

May 2016
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

Dr. Christophe Bobda
Thesis Director

Dr. Jia Di
Committee Member

Dr. Qinghua Li
Committee Member

Abstract

In this thesis, we concern ourselves with the security of drone systems under jamming-based attacks. We explore a relatively new concept we previously devised, known as hardware sandboxing, to provide runtime monitoring of boundary signals and isolation through resource virtualization for non-trusted system-on-chip (SoC) components. The focus of this thesis is the synthesis of this design and structure with the anti-jamming, security needs of drone systems. We utilize Field Programmable Gate Array (FPGA) based development and target embedded Linux for our hardware sandbox and drone hardware/software system.

We design and implement our working concept on the Digilent Zybo FPGA, which uses the Xilinx Zynq System. Our design is validated via simulation-based tests to mimic jamming attacks and standalone, stationary tests with commercial transmitter and receiver equipment. In both cases, we are successful in detecting and isolating unwanted behavior. This thesis presents the current work performed, observations, and the future potential of hardware sandboxing in drone systems.

Table of Contents

List of Figures

Chapter 1 Introduction

1.1 Overview

In today's society, unmanned aerial vehicles (UAVs), commonly known as aerial drones, have become prevalent and useful across many domains. Drones are used in the agriculture industry to give farmers an aerial perspective on their crops, which allows for better understanding of crop performance and thus, better yields [1]. Drone use has grown in the commercial and consumer recreation sector and is pertinent in military space for surveillance and precision strikes [2]. In fact, military drone usage is expected to continue to grow by 50% in the next four years [3]. This rise in usage demands security.

Remotely controlled drones are operated indirectly through a user by transmitting radio frequency (RF) waves allowing a receiver on the drone to receive and decode the transmission into signals for processing in the system. Because control happens indirectly through this RF manner, an attacker can potentially hijack and affect the control of a drone by sending their own RF signals. Naturally, we wish to prevent this attack from manifesting to the best of our ability.

In this thesis, we explore a method in which we can achieve the detection of such a control attack and the prevention of the spread of improper control signals. Our method utilizes the hardware technology of Field Programmable Gate Arrays (FPGAs), which allow for quick, efficient development due to their programmable nature through the use of HDL languages such as VHDL or Verilog. The method we use in this thesis is a relatively new concept we created, known as hardware sandboxing. Our goals of this thesis are to study the prior, practical synthesis of the hardware sandbox for drone use, design a secure drone system on FPGAs that can detect control attacks, and verify our implementation through simulation-based testing.

1.2 Motivation

In the RF world, such aforementioned control attacks are referred to as frequency jamming, which is a very real and common threat seen today. Jamming is defined as the disruption of RF signals by an attacker transmitting in the same frequency band as the target. As shown in [4], RF jamming, like in our drone systems, has a strong likelihood of occurring. This has already been seen in use during the Iraq War with the Warlock RF jamming systems to effectively block UAV systems.

With regards to frequency jammers, there are many different types with varying degrees of effectiveness. There are simple designs, such as the constant jammer, which transmits a jamming signal continuously, or the random jammer, which only transmits intermittently [5]. For more useful jammers, there exist others, such as the reactive jammer where a signal is only sent when target transmission is sensed, which has proven to be effective [5], [6], [7], or other intelligent jammers which know the protocols and modulation used. Traditional means to limit the effects of jamming have turned to spread spectrum techniques, most notably frequency hopping (FHSS) and direct sequence (DSSS). In FHSS, a sender will quickly hop along channels across a wide frequency band in a specified order, while in DSSS, a sender will distribute their message on a narrow band across a wide frequency range through the use of pre-determined spreading codes. The original message is multiplied by the spreading codes to achieve this level of spread-spectrum. A receiver can use the same codes to “de-spread” the received transmission back to its original message. While both of these techniques are useful, a powerful or intelligent jammer can still effectively jam at least parts of the transmission, with FHSS being more susceptible to simple, narrow-band jamming. With great power devoted to jamming, full jamming of these spread spectrum signals is possible [8].

In order to detect jamming, comparing signal strength or signal-to-noise-ratio (SNR) is a common method [6], [9], however this has trouble when dealing with energy efficient, low-power jamming such as reactive or intelligent jammers. To combat the reactive jammers, other techniques such as in [10] have looked at comparing received packet loss at the preamble of the received transmission to the rest of the message. This however does not take into account other intelligent jammers, such as powerful ones where modulation techniques are known and can know to jam the start of the sent message.

Our end-goal is to be able to detect jamming regardless of jammer in RF drone systems. In jamming, an attacker transmits their signal on the same frequency band to interfere with the correct signal. The attacker's goal is to distort the signal to the extent where either the receiver is unable to correctly receive the transmission, resulting in total denial-of-service, or parts of the transmission are incorrect, resulting in loss of integrity and possible denial-of-service for certain aspects of the system. Thus, the drone has been compromised and can be brought down by the attacker due to loss of control. In a drone system, the controller transmits the signals to be read by the receiver to generate the correct control bit signals for flight control. Because in a valid transmission these bit signals follow a predefined protocol, we can monitor these incoming signals to continuously check that no deviation occurs. We assume any deviation from this predefined behavior indicates denial-of-service, and thus, indicates jamming is present. In order to do this, we can utilize a hardware component we designed known as the hardware sandbox.

While more detail regarding the design and structure of the hardware sandbox is given in Chapter 2, the concept of the hardware sandbox was first conceived from the need to combat hardware Trojans. Hardware Trojans are malicious alterations or additions in intellectual property (IP) hardware cores which lay dormant until activation. They can be inserted by non-trusted entities at any of the various stages of IP integration, such as the design, verification, and manufacturing stages [11][12], due to the design and production typically being outsourced. A Trojan is said to contain both a trigger and a payload, such that when a specific input or condition is satisfied for the trigger, the Trojan is activated and its damaging effects, or payload, are released. While difficult to detect prior to runtime, if we consider that the Trojan's payload will have a discernible effect on the output signals of the IP, we can monitor the signals at runtime at the boundary of the component and check to ensure they satisfy the original design description and protocol. We can isolate any IP we deem non-trusted through the monitor process and virtualization of any system-level resources the IP utilizes.

This same concept can be applied in drone jamming. In this case, the trigger is the drone jammer and the payload is the denial-of-service and deviation of control signal behavior. Due to this, we can treat parts of the drone system as non-trusted and build a hardware sandbox to isolate and protect the remaining system from potentially compromised elements.

1.3 Scope of Thesis

This thesis presents the first system design utilizing the hardware sandbox to detect and isolate jamming attacks on drones. The goal here is to design, implement, and study a hardware sandbox for a drone flight system that has the capability to detect and prevent compromised signals from reaching and adversely affecting the rest of the system. To do this, we must first delve into the structure, general design, and functionality of the hardware sandbox to provide the necessary means for this task. Therefore, sufficient prior background and tests are given in addition to ones in drone jamming to show both *how* the hardware sandbox works and prove that it *does* work on testable hardware Trojan benchmarks.

While we based our system and hardware sandbox design around a functioning drone and its necessary elements, such as the consideration of booting an embedded operating system for running flight software, building a fully-working jammer and testing on a flying aircraft is out of the scope of this thesis. Our goal is focused more on the design and study of the hardware sandbox, and therefore, we primarily utilized simulation-based testing to simulate jamming attacks and verify the results from our hardware sandbox. We have, though, performed basic tests on stationary hardware utilizing solely the transmitter and receiver to verify performance under real-world timing constraints and ensure accurate communication between software and hardware.

For the drone system, we utilized commercially available equipment for the controller, receiver, and testing platform. We used the Spektrum DX7SE transmitter and AR7010 receiver and built our hardware sandbox around the protocols used in this equipment. We also used the Digilent Zybo FPGA, a Xilinx Zynq-based board, for implementation and testing. Through the board's FPGA fabric for programmable logic and the Zynq's processing system featuring a dual-core Cortex A9 processor, we were able to accurately create our hardware/software-based design. The hardware portions of the drone system and hardware sandbox were written in VHDL, and the drivers and software-based tests were implemented in C.

1.4 Summary of Chapters

The remainder of the thesis is outlined as follows: Chapter 2 (Hardware Sandbox) details the concept, design, and validation of the hardware sandbox, Chapter 3 (System Design and Implementation for Jamming Prevention**)** goes in depth on the design of the hexacopter system and the hardware sandbox tailored for detecting and preventing jamming, Chapter 4 (Tests and Results) shows the test results obtained in order to validate the anti-jamming design presented in

Chapter 3, and Chapter 5 (Conclusion and Future Work) gives the conclusion and offers a few avenues for the future work of hardware sandboxing in drones.

Chapter 2 Hardware Sandbox

2.1 Overview

The hardware sandbox was born on the concept of isolation in software sandboxing and targets potentially malicious activity in hardware IPs and components. As discussed in [13], there are a number of different techniques used by sandboxes in software to enforce isolation. Among these, the *inline reference monitor*/*system call sandbox* and the *hardware memory isolation* categories are of particular interest for the nature of sandboxing hardware IPs and components. In the first set of categories, the goal is to place run-time checks on resource accesses inside or on the boundary of the application to enforce a given security policy, and in the second, segments of resources are provided to the application itself and a manager handles the data transfer in the sandbox to the rest of the system. We can utilize these two ideas in hardware to create a sandbox which is capable of monitoring a non-trusted hardware IP for malicious behavior while still ensuring isolation from the rest of the system.

2.2 Design and Components

In order to provide the necessary isolation and monitor of a potential malicious IP, we need to consider and include multiple components in our overarching hardware sandbox structure. Figure 1 displays a representation of what the hardware sandbox must consist of to achieve the sandbox goals. These components include a properties checker for the monitoring of the sandboxed IP's signals, virtual resources for isolating possible attacks on shared, physical resources in a system, status and configuration registers for processor control, and a sandbox manager for controlling the data flow inside the hardware sandbox. These components are explained in further detail in the following subsections.

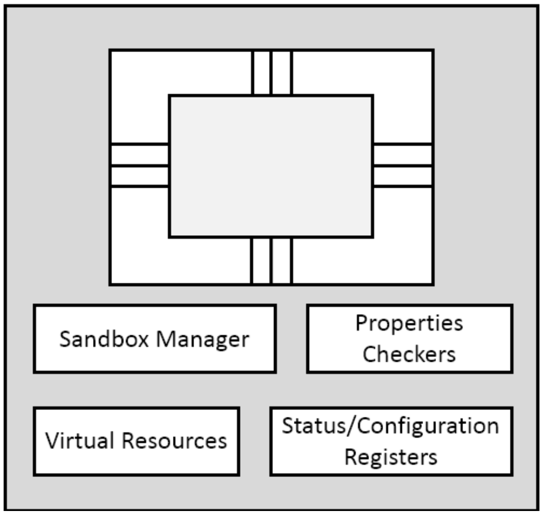


Figure 1. Hardware Sandbox Structure

2.2.1 Properties Checker

This component consists of either one or more checkers used for runtime monitoring and enforcement of security rules defined by the system integrator at compile time. A checker is created based off the incoming or outgoing signal properties of an IP component in the sandbox and can be limited to only a subset of IP signals and properties for overhead reduction. Our implementation of the properties checkers is currently based around the Open Verification Library by Accellera [14]. While this library gives us the ability to create our own expressions to define the necessary properties of the signals to actively enforce and monitor, a more formal method of defining signal properties exists in verification languages, such as the Property Specification Language [15].

2.2.1.1 Open Verification Library

The Open Verification Library (OVL) consists of a series of parameterizable assertion checkers which take in some form of an input signal and monitors and enforces the property needing to be checked. There are many OVL checkers, some more specific in the property or policy being checked; however, there are only a handful of components that can be synthesized directly into hardware. Of these, the most powerful in terms of assertion checking is the OVL Cycle Sequence, which we commonly use for our properties checkers in the hardware Sandbox. The OVL Cycle Sequence takes in primarily an input vector signal, known as *test_expression*, and outputs a signal known as *fire*. Essentially, when a check is active the Cycle Sequence samples the MSB of the *test_expression* sequence on the rising clock edge. As long as this bit value is high, the check passes, and the sequence is left-shifted by one bit. This process repeats for as many clock cycles as specified by the system integrator. If at any point in time during the check the sampled bit is low, the check ends and *fire* is asserted, indicating a problem has occurred. In order for a check to begin, the MSB of the original *test_expression* must be high. If pipelining is specified by the system integrator, multiple checks can take place at the same time; otherwise, the previous check must finish before beginning a new one. Through its customization and by generating *test_expression* from multiple signals based on the temporal and spatial properties specified or observed, the Cycle Sequence can be a very powerful runtime assertion checker and suitable for our needs. For example, consider the system consisting of the signals *req*, *busy*, and *done*. If in this system our signals are observed such that when *req* is asserted, *busy* is asserted on the following clock cycle, and after *busy* is finished, *done* is asserted, we can utilize the OVL Cycle Sequence in VHDL and build *test_expression* through concatenation in the following way: "*req* & (*busy* and not *done*) & *(done* and not *busy)*". In VHDL, this expression is three bits wide and represents our observed protocol.

2.2.1.2 Property Specification Language

The Property Specification Language (PSL) by Accellera, based off the Sugar Language developed by IBM, is an assertion based verification language used for model checking, and evolved into an IEEE standard (1850-2005) [15]. At its core, PSL is used to temporally specify and define the properties of a system under verification through a combination of the temporal logic Linear Time Logic (LTL) and regular expressions. PSL consists of 4 layers: the Boolean layer, the temporal layer, the verification layer, and the modeling layer. Basic relationships among observable interface signals and state variables are defined at the Boolean layer, which take the form of Boolean expressions in the chosen HDL flavor used: VHDL (*reset* and *ena*) or Verilog (*reset* && *ena*). The temporal layer is used to describe signals' behavior over finite or infinite sequence of states. A sequence is built from basic Boolean operators from the below layer combined with specific sequence operators. PSL's temporal layer supports both Sequential Extended Regular Expression (SERE)-style operators and LTL-style operators, which allow for the evaluation of an expression across multiple clock cycles. Said system properties are built utilizing these expressions and Boolean operators to define behavior. For example, taking into consideration two signals in a system which must be asserted at some time, *req* and *ack*, for an LTL-style expression to indicate that whenever *req* is asserted at some time point, *ack* must be asserted on the very next time point, we write our expression as "always *ack* -> next *req*." Utilizing SEREs, with the inclusion of the signals *busy* and *done*, the expression "always {*req*} |=> {*ack*; *busy*[+]; *done*}" indicates that when *req* is asserted on some time point, the following sequence occurs starting at the very next time point: *ack* is asserted for one time period, *busy* is asserted for at least one time period after *ack*, and after *busy*, *done* is asserted, signaling the end of the sequence. Because PSL is primarily a verification language, the verification layer exists to provide all necessary directives for a verification tool to check for the validity of a property defined from the two above layers. A common directive, assert, for example, will instruct a verification tool to report a failure if a certain defined property does not hold. Lastly, the modeling layer utilizes the underlying HDL language to model combinational signals or other aspects of the system that are not directly part of the design, but are needed for accurate verification purposes.

2.2.2 Virtual Resources

Regarding this component, the concept of a sandbox requires that resources needed by IPs are provided in virtual form within the sandbox where they can be used by an IP without negatively affecting the rest of the system. Typically, the virtual resources considered for inclusion are those that are shared between the trusted system and the potentially malicious component because we want to limit the effects of a compromised IP. The main advantage here is that the interface between virtual resources within the sandbox and physical resources follows a secure protocol and can never cause a denial-of-service. Any attempt from a malicious entity to alter a system's resource or peripheral will be nullified by the virtual resource. Examples of virtual resources include a virtual UART, virtual VGA, or virtual memory.

2.2.3 Status and Configuration Registers

Next, the hardware sandbox features a set of status and configuration registers to be used for the communication between the sandbox manager and the rest of the system. Statistics on the behavior of IPs in the sandbox can be recorded for further analysis. If misbehavior occurs and is caught via a checker, this event can be indicated in a status register for the processor to read and react accordingly.

2.2.4 Sandbox Manager

The final aspect to consider in the hardware sandbox is a manager. The manager is simply in charge of all data exchange and handling inside the sandbox. This includes the exchange between virtual resources and their physical counterparts, as well as the handling of the results from the checkers and the potential configuration of the sandbox.

2.3 Validation Tests and Implementation

To validate our design and demonstrate how the hardware sandbox works in practice, we previously implemented a series of UART module-based hardware Trojans from trust-HUB [16]. Trust-HUB, founded by experts from the University of Connecticut, is seen as the benchmark resource in hardware Trojans today. All of the UART modules are based off the RS232 protocol with parity. Therefore, it is first necessary to understand how this protocol works.

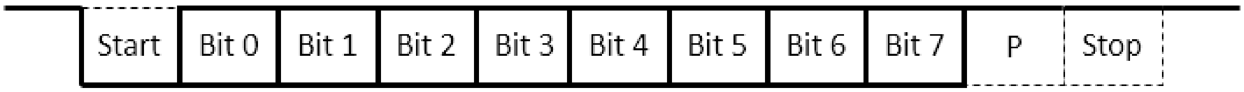


Figure 2. RS232 Protocol with Parity Used by the UART Modules

Figure 2 outlines the RS232 transmission protocol. When a transaction begins taking place, RS232 synchronizes using the start bit, which is logic low, to signal the start of the frame before sending the data. Once the synchronization period is over, RS232 sends the data, bit-by-bit, beginning with the LSB. After, the parity bit is sent, followed by the end-of-frame stop bit, both of which are logic high. Any future transaction must repeat this process by de-asserting the line for the start-of-frame.

The hardware sandbox was customized and tailor-made for the RS232-UART module. Using the protocol above, a checker was built from the OVL Cycle Sequence module to monitor the transmission signals. A virtual UART component was included to receive the expected transmission and only transmit to the rest of the system if the protocol was followed correctly.

For brevity, this thesis will only focus on one of the many implemented RS232-UART Trojans, named the T-900. This Trojan solely impacts the UART transmission through denial-of-service after activation by transmitting a specific sequence of data. The following subsections give the simulation results of the RS232-UART without the Trojan, with the Trojan, and with the hardware sandbox in use.

2.3.1 Trojan-Free RS232-UART Transmitter

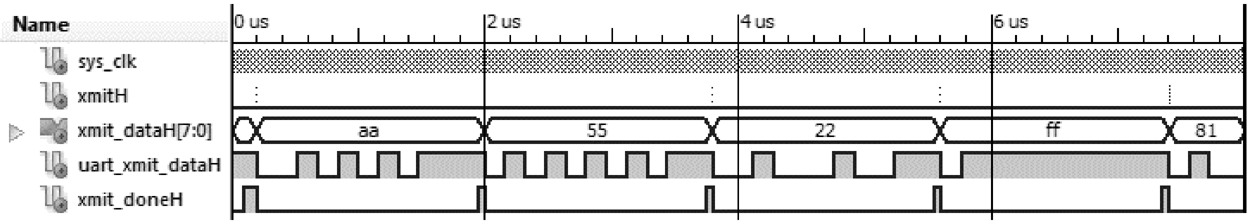


Figure 3. Trojan-Free RS232-UART Transmission Waveform

From Figure 3, we can adequately see the valid, uninterrupted RS232 transmission protocol the UART module uses. This protocol is based on the following signals: *xmitH*, used to signal the start of a transmission, *xmit_dataH*, used to denote the 8-bit data to transmit, *uart_xmit_dataH*, representing the serialized, UART data line, and *xmit_doneH*, used to signal when the transmission is complete. For an RS232 transmission, after *xmitH* is asserted, the *xmit_dataH* signal is sampled and the transmission begins. The *uart_xmit_dataH* line begins low for 16 clock cycles representing the start bit. After, the serialized line follows the data in the sampled *xmit_dataH*, bit-by-bit, for 16 clock cycles each, starting from the LSB. Once finished with sending all 8 bits, the *uart_xmit_dataH* line will be asserted for 31 clock cycles representing both the parity bit (16 cycles) and the stop bit (15 cycles). Once complete, *xmit_doneH* is asserted to indicate the full transmission is finished on the 16th cycle for the stop bit.

2.3.2 RS232-UART Transmitter with the T-900 Hardware Trojan

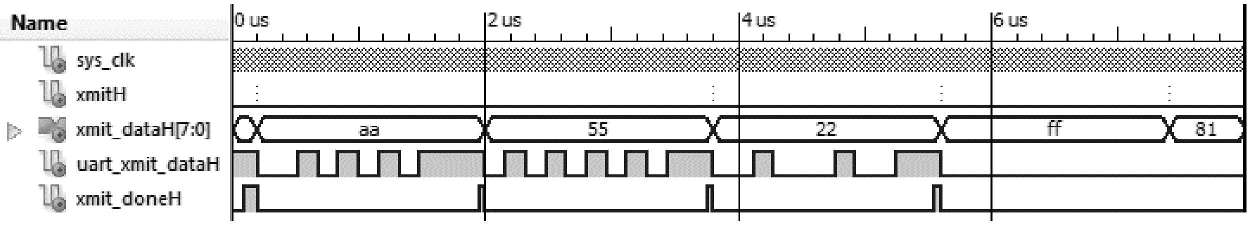


Figure 4. T-900 Trojan-Injected RS232-UART Transmission Waveform

Figure 4 shows how the T-900 Trojan affects the RS232 transmission sequence. In the T-900 Trojan, when the user transmits the data 0xAA, 0x55, 0x22, and 0xFF, in that order, the

Trojan's payload is triggered and blocks all future transmission. This is clearly seen in Figure 4. After the start of the transmission of 0xFF by asserting *xmitH*, nothing is ever serialized on *uart_xmit_dataH,* and *xmit_doneH* is never asserted. Thus, this UART module has effectively been placed under a denial-of-service. Also, because *uart_xmit_dataH* and *xmit_doneH* are both low, indicating in normal operation data is currently being transmitted, any IP in the full system this Trojan-injected module is connected to, such as other UART modules, would experience problems and loss of integrity and availability from this malicious behavior. Therefore, the invalid behavior needs to be contained.

2.3.3 Hardware Sandboxed RS232-UART Transmitter with the T-900 Hardware Trojan

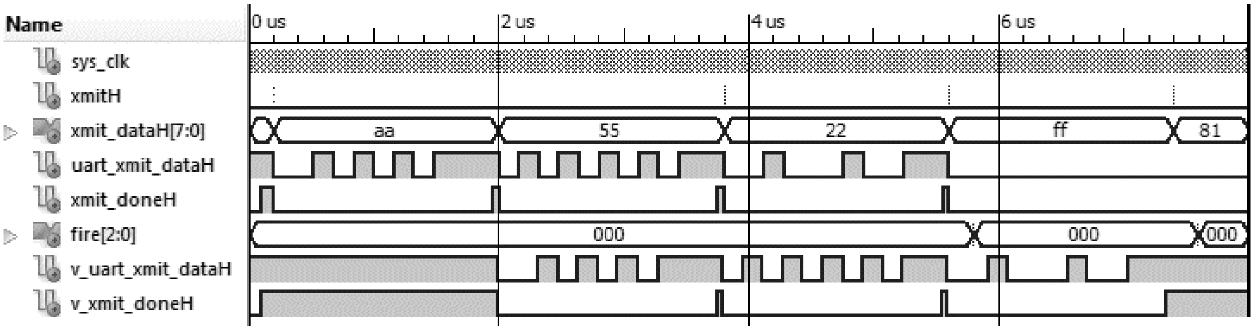


Figure 5. T-900 Trojan Injected RS232-UART Transmission with the Hardware Sandbox Waveform

The hardware sandbox is demonstrated in Figure 5. The signals *v_uart_xmit_dataH and v_xmit_doneH* represent the virtual UART signals coming from the hardware sandbox that interface to the rest of the system. As we can see, when the T-900 Trojan becomes active in the same manner as seen in Figure 4 with the transmission sequence, the *fire* signal from the OVL Cycle Sequence is asserted for a single clock cycle in the two locations shortly after *xmitH* is asserted, indicating the aforementioned problem in the transmission. Because in addition to checking the properties of the signals we include the virtual UART module in our hardware sandbox, the rest of the system is not negatively affected by the Trojan's misbehavior. This is seen in the virtual UART signals. The real, valid transmission is still received virtually and transmitted out when the protocol is correct. Once a problem is found in the sandboxed module, the transmission ends. However, instead of incorrectly outputting that the serial line is busy and transmitting to the rest of the system as the Trojan would, *v_uart_xmit_dataH* and *v_xmit_doneH* output correct values for no transmission. Thus, through the hardware sandbox, we have effectively isolated the reach of the hardware Trojan from affecting the availability and integrity of the entire system, and we have detected the first moment when an error in protocol has occurred.

Chapter 3 System Design and Implementation for Jamming Prevention

3.1 Overview

In the previous chapter, we demonstrated how the design of the hardware sandbox is capable of preventing malicious hardware Trojan behavior from affecting a system's integrity and availability through the monitoring and checking of defined signal properties and inclusion of virtual resources. Utilizing the same structure, we can define a hardware sandbox to detect and prevent RF jamming from compromising a drone system. The following sections outline our current drone system and how we can leverage and design the hardware sandbox for our anti-jamming needs.

3.2 Materials and Platforms

3.2.1 Hexacopter Drone

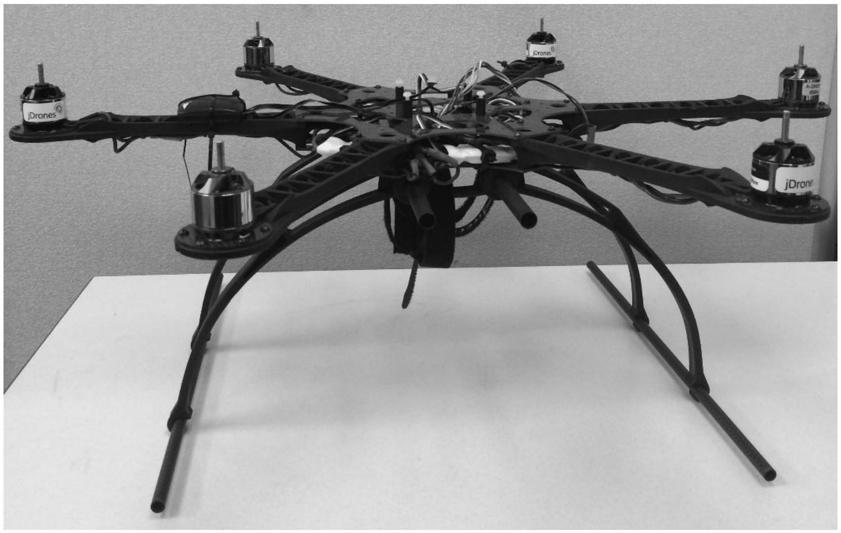


Figure 6. Skeleton Hexacopter Drone

Our base drone design is given in Figure 6. For this project, we use the commercially available multicopter drone, the XAircraft DIY Hexa. This drone features six arms, and thus six motors for flight control, providing added stability and fault tolerance compared to a typical quadcopter drone with only four motors. In multicopter systems, the steering control of the drone is based around three axes: yaw, pitch, and roll [17]. The copter is able to rotate at will around these three axes in order to ascend, descend, turn, or fly accordingly. A flight controller must be capable of sending a transmission that can be interpreted to adjust these parameters for the drone.

3.2.2 Control Transmitter and Receiver

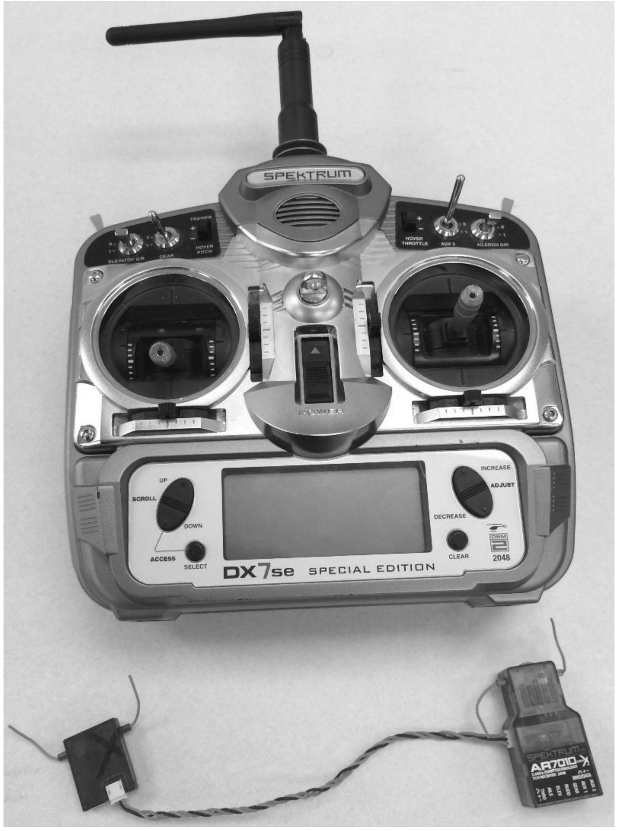


Figure 7. Spektrum DX7SE Transmitter and AR7010 Receiver

Our current drone uses the Spektrum DX7SE transmitter and the Spektrum AR7010 receiver, as pictured in Figure 7. This transmitter/receiver combo operates in the 2.4 GHz RF band using Spektrum's proprietary DSM2 DSSS-based technology. Once paired and transmitted, the receiver gathers the transmitted signals and decodes them into seven channels to control the drone: throttle, ailerons, elevators, rudder, gear, auxiliary-1, and auxiliary-2. For direct control of the drone, the key signals here are the throttle and the ailerons, elevators, and rudder, which control the roll, pitch, and yaw of the drone, respectively. Under normal conditions, when the controller is paired and transmitting to the receiver, these seven channels exhibit the same properties: each signal follows a pulse width modulation (PWM) with pulse length between around 5% and 9%, depending on user input, at a frequency around 45.45 Hz. While the pulse length can be programmed on the controller to be modified by the user, we keep the parameters at these values. These seven signals are cascaded, one after the other, with the following pattern: ailerons, auxiliary-1, gear, elevators, auxiliary-2, throttle, and rudder. Once one signal finishes its PWM pulse, the next in the pattern begins immediately after. Once all are complete, all signals are low for the remaining period time. When the transmitter and receiver are disconnected, however, such as when the controller is turned off after transmitting, the receiver still generates the PWM-based signals, albeit at an initial, slower frequency that decreases with time spent disconnected.

3.2.3 Digilent Zybo

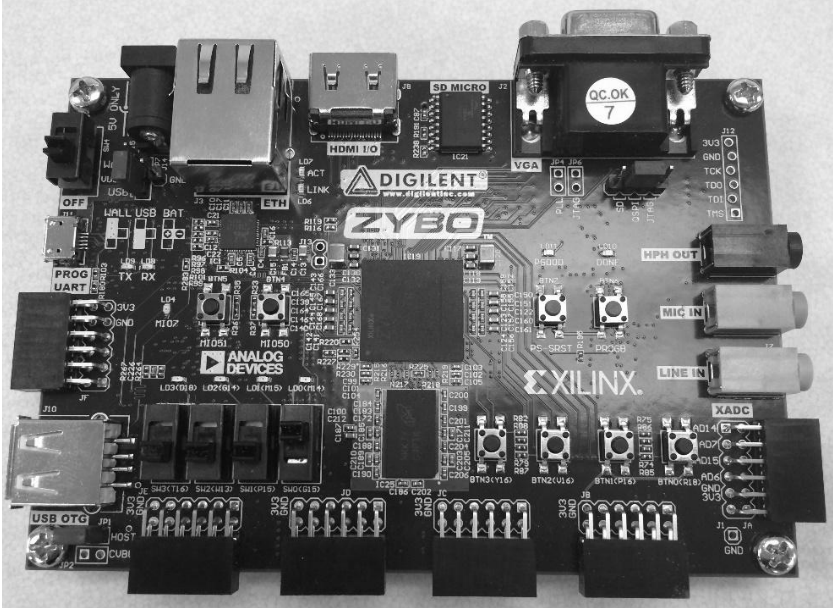


Figure 8. Digilent Zybo FPGA

The Zybo (Zynq Board) FPGA by Digilent, pictured in Figure 8, provides us the necessary features and functionality for a hardware/software system, such as our drone system. The Zybo includes the Xilinx Zynq-7010 System-on-chip, featuring a dual core ARM Cortex-A9 processor, along with Xilinx 7-Series FPGA logic [18]. The board features a number of multimedia and connectivity peripherals, of which the UART and SD are instrumental to this thesis. The UART peripheral was used for terminal communication and obtaining various design information, such as the sequence of cascaded signals from the receiver, while the SD was used for running embedded Linux and storing the necessary kernel drivers. This board also features six PMOD connectors, which are also vital for this drone system to interface to various, external devices, including the Spektrum receiver and the drone's motors. While it is a fairly small Zynq-based FPGA board compared to others such as the Digilent Zedboard or the Xilinx ZC702, the Zybo is perfectly suitable for our design requirements.

3.2.4 ArduCopter Software

For flight control, this drone system uses the open source ArduCopter software by ArduPilot. This software was originally designed for Arduino-based boards, but through a hardware abstraction layer, it can be ported for many different boards, including those featuring the Zynq system-on-chip, such as the Zybo. Prior to this thesis project, the ArduCopter code had already been ported and compiled for use on the ARM processor found on the Zybo. However, because of the dependencies in the code, such as threading, an operating system had to be built for the Zybo to run ArduCopter.

3.2.5 Embedded Linux

In order to run the ArduCopter software, we turned to using embedded Linux. We built the Linux kernel for the Zybo, provided by Digilent, and used Buildroot for creating a root file system to include the required libraries and C++ support for running the software. The main takeaway for this thesis, however, is that because we run the flight software on top of the Linux kernel, a device driver had to be written and loaded into the kernel for the hardware sandbox and other IPs in the system to access their respective hardware registers.

3.3 Drone Design

Combining the previous information about our thesis-relevant materials and platforms, our simplified, high-level system design for the unsecured drone is given in Figure 9 below.

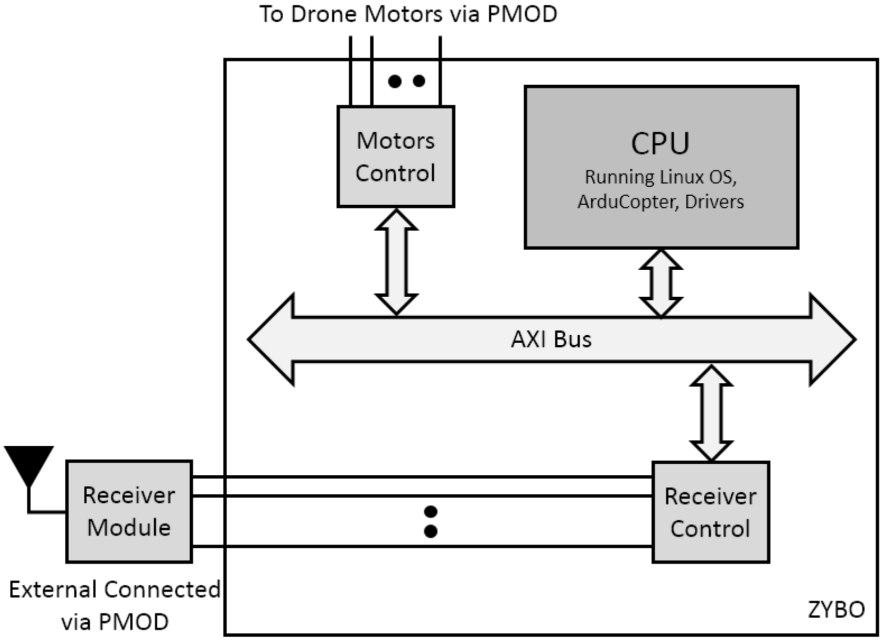


Figure 9. High-Level Drone System Design

Of the modules and components to this system, only the two control IPs were not previously mentioned: the Receiver Control IP and the Motors Control IP. These two hardware IPs simply provide an interface for the ArduCopter software to read and write to and from the receiver's signals and the directly-connected drone motors. Please note, however, that do to the nature of this thesis, only the pertinent hardware and system-level components for flight control and its security are considered here.

For control of the copter, the user transmits the controller's input from the Spektrum DX7SE to the AR7010 receiver. After decoding the RF transmission into the seven control channels, these signals are fed to the Receiver Control IP for register storage. These control register values are fetched regularly by the CPU in the ArduCopter software running on Linux and must be processed to determine the state of the six motors of the hexacopter drone. These motor values are then written from software to the hardware registers of the Motor Control IP and sent out via PMOD connectors to the physical motors of the drone.

This design, however, is unsecure and susceptible to jamming attacks on the transmission and reception process to negatively impact or take down the flight. Note that if an attacker wishes to disrupt this flight control flow, the only means to do so would be through an RF receiver like the Spektrum AR7010 used. While preventing the attacker from jamming our real transmission signal is impossible, if we can protect the system from a potentially compromised receiver, we can secure the flight of the drone. This follows the nature of hardware sandboxing.

3.4 Hardware Sandbox Integration

Because we treat the receiver as a non-trusted component with the possibility of being compromised by a jammer, we can utilize the hardware sandbox in a similar fashion as seen in Chapter 2 with respect to isolating hardware Trojans. We place the hardware sandbox on the boundary of the external RF receiver to monitor its signals, as seen in our new, secure system design in Figure 10 below.

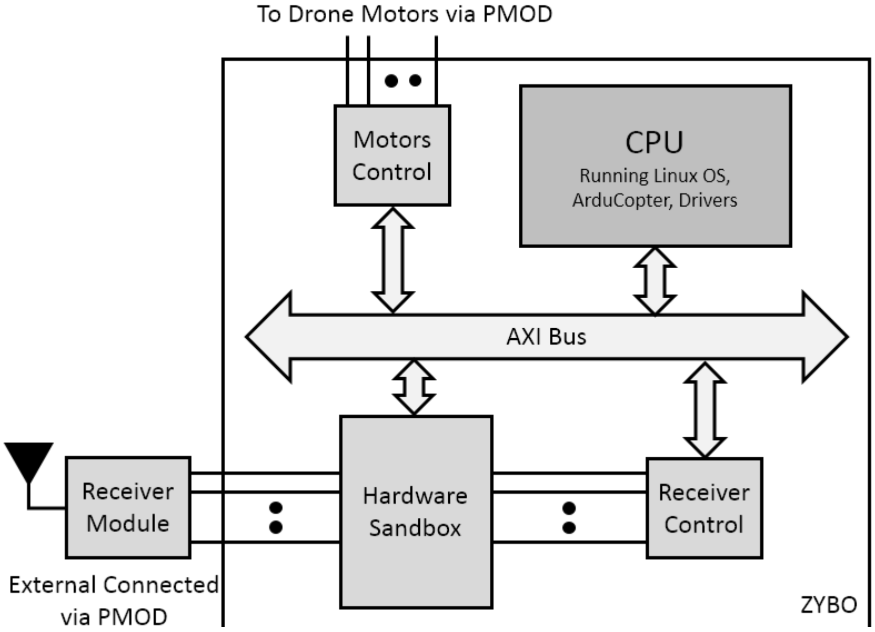


Figure 10. High-Level Drone System Design with Hardware Sandboxing

The signals from the receiver are monitored for the possibility of malicious behavior and, through a virtual receiver, only valid, protocol-following signals are output to the rest of the system for flight control. In this way, we can know when a jamming attack is taking place and that the rest of the system will be as least negatively affected as possible. A more detailed, high-level diagram is given in Figure 11 and more detail regarding the principal components is given in the subsections below.

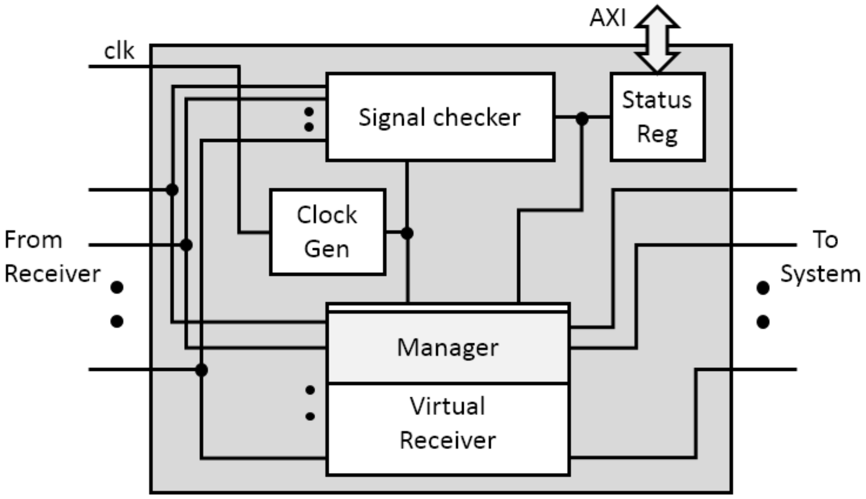
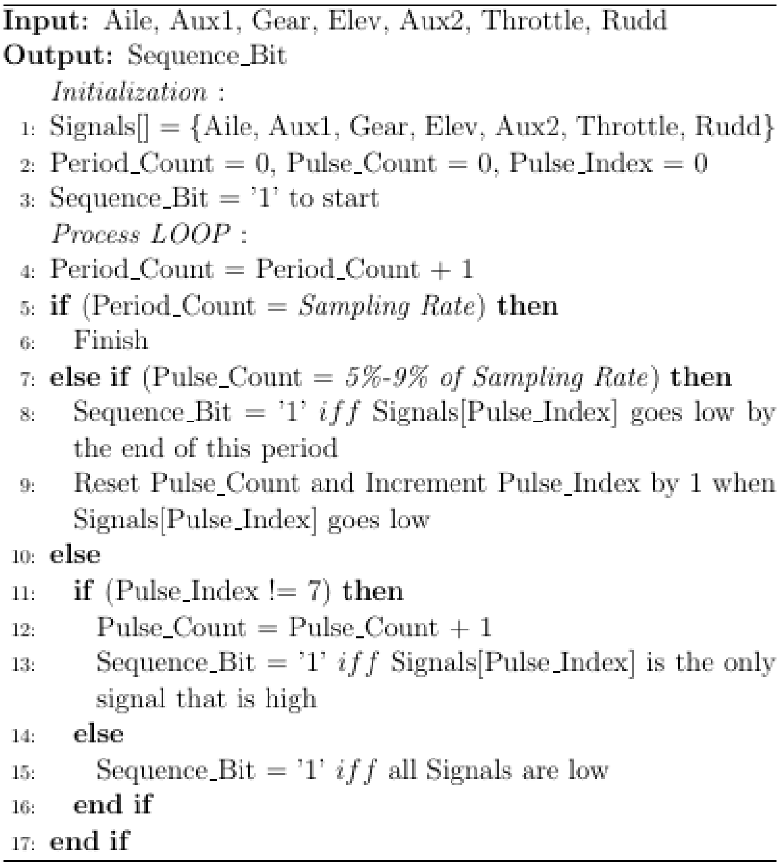


Figure 11. Receiver Hardware Sandbox Diagram

3.4.1 Signal Checker

Because the protocol of the correct, paired receiver is known, as described in 3.2.2, we can build a checker using the OVL Cycle Sequence to sample the seven PWM-based signals for valid behavior at a constant rate, multiple times during a single PWM period. To do this, we must divide the system clock (100 MHz) such that the frequency matches the original frequency of the PWM signals, around 45.45 Hz, multiplied by the sample rate. For this thesis, we use a 200x sample rate for a good balance between checker resolution and design area. A 100x, 500x, 1000x, and 2000x sample rate were also considered and their results are shown in Chapter 4. Because our checker works on the temporal nature of the signals, it is imperative to get as close to the correct frequency as possible. Therefore, we employ the Xilinx Mixed-Mode Clock Manager (MMCM) Clocking primitive in the Xilinx Clocking Wizard IP core [19] to first generate a clock as close to 45.45 MHz and divide this down to 9.09 kHz (200x 45.45 Hz), which is now possible due to the two clocks being an even multiple of one another. The closest, initial clock possible with the MMCM primitive is 45.445 MHz. While not the exact frequency, this is okay due to our necessary window-based checker design described in more detail shortly below.

With the right clock frequency and number of times to sample in a period, we now turn our attention to generating the proper checking sequence for the OVL Cycle Sequence's *test_expression*. This sequence is based on the seven channels from the receiver: throttle (*throttle*), ailerons (*aile*), elevators (*elev*), rudder (*rudd*), gear (*gear*), auxiliary-1 (*aux1*), auxiliary-2 (*aux2*). Unlike the static RS232-UART examples that follow the same sequence every transmission, every clock cycle, the receiver has the dynamic property described in 3.2.2 where the start of the pulse of the current control signal is dependent upon the previous pulse length, not a specific time value. Therefore, a sequence generation algorithm had to be devised. This is given below in Figure 12.



```
Input: Aile, Aux1, Gear, Elev, Aux2, Throttle, Rudd
Output: Sequence_Bit
  Initialization :
Signals[] = {Aile, Aux1, Gear, Elev, Aux2, Throttle, Rudd}
Period_Count = 0, Pulse_Count = 0, Pulse_Index = 0
Sequence_Bit = '1' to start
  Process LOOP :
Period_Count = Period_Count + 1
if (Period_Count = Sampling Rate) then
  Finish
else if (Pulse_Count = 5%-9% of Sampling Rate) then
  Sequence_Bit = '1' iff Signals[Pulse_Index] goes low by
  the end of this period
  Reset Pulse_Count and Increment Pulse_Index by 1 when
  Signals[Pulse_Index] goes low
else
  if (Pulse_Index != 7) then
    Pulse_Count = Pulse_Count + 1
    Sequence_Bit = '1' iff Signals[Pulse_Index] is the only
    signal that is high
  else
    Sequence_Bit = '1' iff all Signals are low
  end if
end if
```

Figure 12. Sequence Generation Algorithm

Utilizing this sequence generation algorithm and our sample rate of 200x, our sequence vector for *test_expression* would be 200 bits long, thus taking the full 200 clock cycles for the receiver PWM period to complete. Before starting the algorithm, however, we must first synchronize with the receiver, waiting for the first rising edge of *aile*, as this signal is the first in the cascaded sequence described in 3.2.2. Once synchronized, we start by first initializing the sequence bit with an always '1' to indicate that, because we should always be receiving a signal from the transmitter in a non-jammed environment, a check should always be running. From there, we start checking the first pulse, *aile*, to ensure that only it is high for the first 5%. After, because the user's control dictates the length of the PWM, the next 4% of the period is independent, and we can only check to ensure that both the signal goes low during this time and the other signals remain low. Once the active signal goes low, we move onto the next signal's pulse and repeat this process. Finally, after ensuring all seven signals followed their correct pulse pattern, the remainder of the sample period is devoted to ensure all signals stay low.

While this sequence bit generation represents the correct check to be used in our sandbox, in practice it needs to be made more complex. Due to potential for jitter and inconsistent clock periods, this algorithm will not hold as it assumes a rigid PWM structure and period. In fact, jitter is present on the AR7010 receiver's clock, typically deviating by a margin of up to 0.3 Hz from the 45.45 Hz baseline. Therefore, we need to allow for this margin by adding a short window to the time before and after the assumed end points of each signal pulse and total period. For the signal pulses, this window is added onto the 4% PWM length that is user-dependent. As long as each signal pulse goes low during this period, we assume the signals are still behaving correctly. Once the pulse does go low, we look to check the next signal pulse for correct behavior. The same concept is applied for the full PWM period ending and starting anew with the rise of the signal, *aile*. While this does not provide us with quite as fine-grain resolution as the validation UART tests in 2.3, any major deviance from the base protocol will still be detected, especially if the window size is small. For this thesis, we use a window size of two clock cycles. At a 200x frequency rate, this gives an approximate 0.45 Hz jitter tolerance to the base 45.45 Hz signal, which covers the observed 0.3 Hz maximum jitter.

With respect to the length of the window tolerance, we add the number of clock cycles onto the length of our sequence vector, *test_expression* – in our case, this is two cycles, and thus, two extra bits. Because the signal is allowed to finish its full period during the window, we add one more extra bit at the end to indicate whether or not the period actually finished during the window and a new one started or not. Therefore, in total, our *test_expression* input is 203 bits long, and the Cycle Sequence takes 203 clock cycles to finish. This is too long for a normal PWM signal period at our sample rate, and thus, we use the pipelined setting of the OVL Cycle Sequence to allow multiple checks to happen at the same time. Whenever a new *aile* pulse begins during the end window, we start the next check while still allowing the current check to finish the remaining clock cycles. This is not a problem as the generated sequence bits at the end during the window are independent from the rest of the sequence generation at the beginning, and therefore, there will never be overlapping, conflicting values between checks.

3.4.2 Virtual Receiver/Manager

Next, we turn to including virtual resources in our drone receiver hardware sandbox to ensure our system cannot be compromised. For our drone, this resource must be the receiver as both the attacker and the user share this hardware. However, because all communication with the receiver happens via airwaves off the board, we cannot utilize our virtual resource in the same manner as other virtual resources: isolating the attack to prevent the physical resource from being compromised and affecting the rest of the system. Instead, we can do just the opposite: allow the best, correct communication possible in spite of a compromised physical resource. Thus, we can include a virtual receiver to generate the PWM signals following the correct protocol as described in 3.2.2. So long as the *fire* from our OVL cycle sequence-based checker is low representing correct behavior, we can simply pass the received signals to the rest of the system; otherwise, each channel defaults to a base PWM value from the point of error to the end of a full period. We use the basic 5% PWM value as our default. From there, we wait until we experience a full valid period without *fire* being asserted, indicating the jamming problem is gone. If this happens, we switch back to the received signals for the next period until *fire* is reasserted.

Extra care is needed to make the transition from received signals to our correctly-generated signals seamless. Therefore, we monitor the position of our sample during a full period and determine which signal is currently active and for how many clock cycles. Based on this knowledge, if *fire* is asserted, we know exactly how to proceed for the remaining period. For example, if *fire* is asserted due to a signal pulse not lasting long enough, we can seamlessly take over and generate the correct pulse length without the rest of the system seeing the problem. The reverse is also true if the pulse remains high for too long. The virtual receiver can begin the next signal's pulse, outputting the correct protocol-behaving signals to the rest of the system. Due to this necessary ability, we must buffer the incoming received signals to align with the results from the OVL Cycle Sequence and our virtual receiver logic.

Our virtual receiver's manager assumes the same flexible design that the sequence generation uses with regards to window size. Because our virtual receiver's output is based solely on the Cycle Sequence's *fire* signal during a period, it is imperative that the incoming signals, the current check, and the virtual receiver's position are all in sync. For example, if the period of the signals were to deviate too far and invoke *fire*, our virtual receiver generation would start. However, because of clock jitter, if the jamming were to be stopped, it is possible for the signals to make up the margin lost or gained by the frequency change when jammed and re-match our sequence generation logic. Doing so, would cause *fire* to stop being asserted. If the virtual receiver was rigid, the changeover from virtual to real signals would be incorrect.

3.4.3 Status Register

Lastly, we connect *fire* to a status register to be read via software to indicate if a problem has occurred. We interface with this register through the AXI bus and protocol. Since *fire* is only at most asserted for one clock cycle during a single check period, in order to guarantee that we can read the register to know a problem has occurred, we connect this signal to a D Flip-Flop with Enable, setting the enable to be a signal indicating whether the value has been read or not. When *fire* is asserted, this read signal goes low, meaning this value has not been read. When a valid read has occurred in the AXI protocol, the read signal is asserted, and new values, such as *fire* being low, will now be stored. Because embedded Linux was used, a character device driver was written to be able to read this register in the software running on Linux for potential logging.

Chapter 4 Tests and Results

4.1 Overview

With our full design in place, we look to verifying our method through our tests, observations, and results. We use the Vivado Design Suite and Flow provided by Xilinx for synthesis and implementation for the Zybo FPGA. We synthesize and implement using the Vivado Synthesis and Implementation Defaults. We give our implementation results in Section 2, provide assumptions related to the jamming impact on signals in Section 3, display simulation waveforms based on these assumption in Section 4, and finally, describe two short, real-world experiments to validate the signal checker's ability to detect invalid behavior and the hardware sandbox's ability to perform in a Linux environment in Section 5.

4.2 Implementation Results

After implementation, we compare various performance metrics to show the resource overhead when utilizing the hardware sandbox. Table 1 shows the utilization results of the individual, principle hardware sandbox components.

	Slice LUTs (Util%)	Slice Registers (Util%)
Signal Checker	438 (2.49%)	379 (1.08%)
Virtual Receiver/Manager	88 (0.5%)	55 (0.16%)
Status Register	134 (0.76%)	171 (0.49%)

Table 1. Resource Overhead for Each Component in the Hardware Sandbox

While we use the full hardware sandbox in our design, depending on the design goals of the system, it is possible to strip elements, such as the status register, from the sandbox and save on resource usage. As we can see, our signal checker makes up the majority of the required area of the hardware sandbox. This makes sense as this component takes both the sequence generation logic and the OVL Cycle Sequence. As described in 3.4.1, the length needed for our checker at 200x sample rate is 203 bits. The OVL Cycle Sequence pipes these 203 bits one register at a time for its logical tests, and therefore, as the sample rate increases, so does the area. Table 2 gives a few results of considered sample rates versus resource usage.

Sample Rate	Window Size	Bit Length	Slice LUTs (Util%)	Slice Registers (Util%)
100x	1	102 bits	290 (1.65%)	273 (0.78%)
200x	2	203 bits	438 (2.49%)	379 (1.08%)
500x	5	506 bits	867 (4.93%)	688 (1.95%)
1000x	10	1011 bits	1597 (9.07%)	1198 (3.40%)
2000x	20	2021 bits	3033 (17.23%)	2213 (6.29%)

Table 2. Utilization Results of Various Signal Checkers

This table only displays the resource usage of the checker. Because the sample rate increases, and thus the clock frequency in the sandbox, the window size must also increase to allow for the same total window time. Bit length gives the total number of bits required for our sequence-based *test_expression* for the OVL Cycle Sequence. As the sample rate increases, the resource utilization increases roughly linearly. For additional security, an integrator may opt for a higher sample rate. However, 200x offers ample coverage of our PWM signal while using fewer resources, and therefore, in this thesis, we utilize this rate.

In order to show total resource overhead of our drone system design with and without hardware sandboxing, we present Table 3 below.

	Slice LUTs (Util%)	Slice Registers (Util%)
Drone System	851 (4.84%)	938 (2.66%)
Secure Drone System	1511 (8.59%)	1543 (4.38%)

Table 3. Utilization Results for the Drone Design with and without Hardware Sandboxing

As we can see in Table 3, the drone design with hardware sandboxing uses only 660 more lookup tables and 605 more registers, representing an additional 3.75% and 1.72% of the Zybo's total available resources, respectively. While the resource utilization in this context compared to the base drone system appears costly, with a 77.6% increase in LUTs and 64.5% increase in register slices, we argue this is reasonable. First, the original design was not large to start so any resource increase through additional logic will make the design of magnitudes higher quickly. Second, while the drone system can increase in size though additional functionality and feature set, the hardware sandbox will remain a fixed size. This is due to the hardware sandbox being tailored for a specific protocol, which remains unchanged. Therefore, in larger designs the hardware sandbox will not appear as obtrusive. For the additional security, this small tradeoff is reasonable.

	Worst Negative Slack (ns)	Power (W)
Drone Design	3.201 ns	1.657 W
Secure Drone Design	2.663 ns	1.658 W

Table 4. Performance Metrics for the Drone Design with and without Hardware Sandboxing

Table 4 lists the timing and power result comparison between designs. The worst negative slack displays the lowest setup slack in the system from a 10 ns period (considering the 100 MHz system clock frequency) with the larger value meaning a greater cushion to meet timing. First, while our drone design with hardware sandboxing meets the same timing requirements as that of the base design, it has a longer critical path by 0.538 ns in the path to the memory mapped registers found in the IPs and status register. This increase is not from the logic of the signal checker or the virtual receiver, but rather found in a path in the AXI memory mapped registers logic, and largely due to a longer routing delay from the larger design.

The second aspect in Table 4 is the power consumption. This was performed at the default settings in Vivado's power estimator at 25 degrees Celsius ambient temperature. As evident, the power consumption is largely the same, with both having the Zynq processing system consume roughly 92% of the reported power. Even though there is added logic with the hardware sandbox, because it operates at many magnitudes slower frequency than the rest of the system (on the order of kHz versus MHz), the power consumption impact is negligible.

4.3 Jamming Assumptions

Because building and testing a full jammer for our transmitter/receiver combination is out of the scope of this thesis, we make a couple of assumptions and hypotheses regarding the impact of jamming on the seven cascaded control signals output from the receiver before testing our design. First, because jamming aims to disrupt and even fully block communication to the receiver, a behavior of the signals under full jamming would be the equivalent of turning off the transmitter to prevent the receiver from receiving any form of valid transmission. In our case, as mentioned in 3.2.2, when the controller is turned off and disconnected from the receiver, the frequency of the PWM signals decreases. Therefore, full jamming has the ability to alter the frequency of the signals output from the controller and must be prevented.

Second, because it is possible to only jam parts of a transmission such that while the receiver still receives communication, parts of it could be incorrect, as demonstrated in jamming bits in WLAN communication [20], a possibility of signal behavior under partial jamming would be wrong PWM pulse lengths stemming from the disrupted transmission. We do not believe this to be a strong assumption due to the controller's ability to initially program the sensitivity range of control aspects, which in turn would affect the PWM pulse of the signals. While we keep the pulses between 5% and 9%, we assume a jammed transmission has the potential to alter these lengths. However, unlike the first assumption regarding full jamming, which has solid support from manual observation, we note that this assumption would require future testing and research to accurately verify. For the sake of this thesis project, however, we include it.

4.4 Simulation Testing

Based on our assumptions, we simulate various test cases in Xilinx ISim to show the hardware sandbox is capable of detecting and preventing the malicious behavior from affecting the rest of the system. All of the following waveforms depict the same signals. *Jam* represents the period in which the transmission is being jammed, and thus, the impact on the signals is felt, and *fire* is the output from the signal checker's OVL Cycle Sequence to indicate bad behavior. The seven control signals under the header "From RX" represent the potentially compromised signals being output from the receiver before the hardware sandbox, and the seven signals under the header "To System" represent the protocol-conforming signals coming from our hardware sandbox and virtual receiver to the rest of the system.

4.4.1 Jamming Impact – Short Pulses

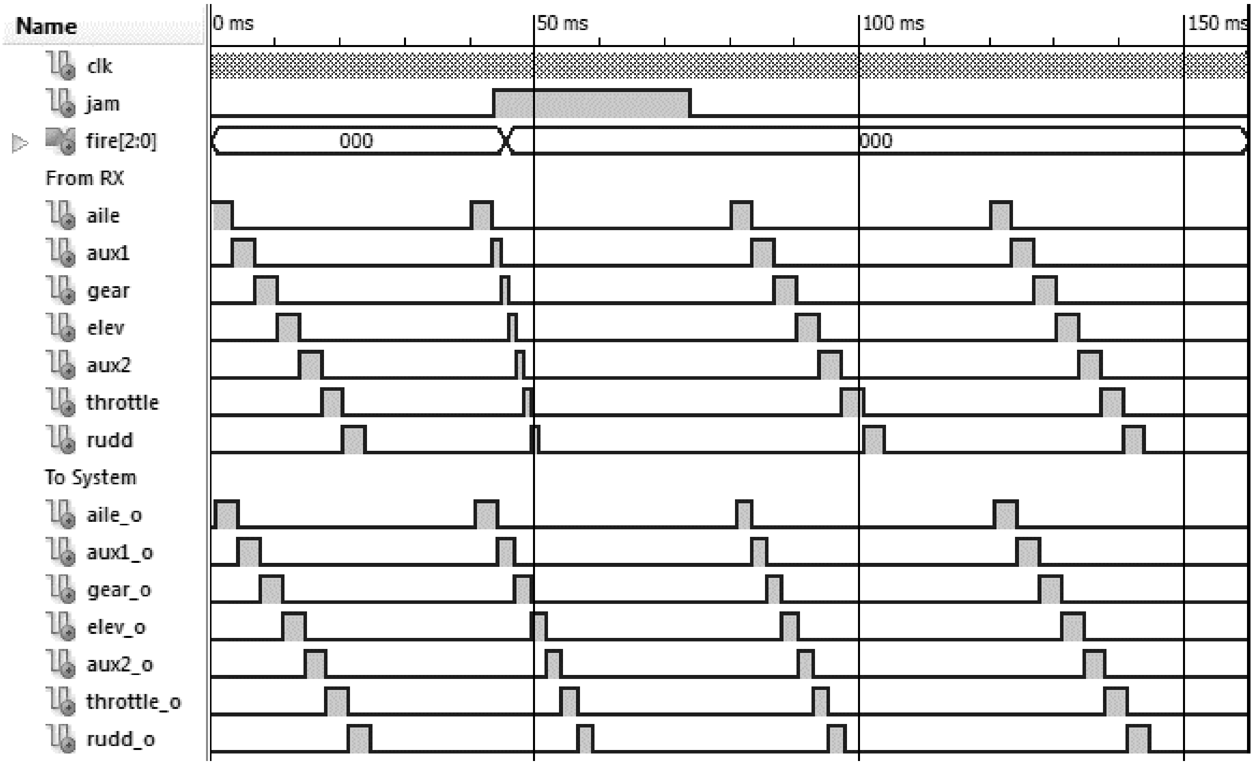


Figure 13. Jamming with the Hardware Sandbox (Short Pulse) Waveform

In Figure 13, we look at the hardware sandbox's ability to fight off jamming which impacts the PWM signals from reaching minimum pulse length. In this example, when *jam* is high, the remaining signals to be sent are changed from an 8% pulse to that of a 2.5% pulse. When this happens, we detect the incorrect protocol from our signal checker, *fire* is asserted, and our virtual receiver seamlessly takes over from this spot generating the remaining signals at a 5% pulse length. We wait one additional period to ensure the jamming is finished and the incoming signals are behaving correctly before switching back to the signals coming from the receiver.

4.4.2 Jamming Impact – Long Pulses

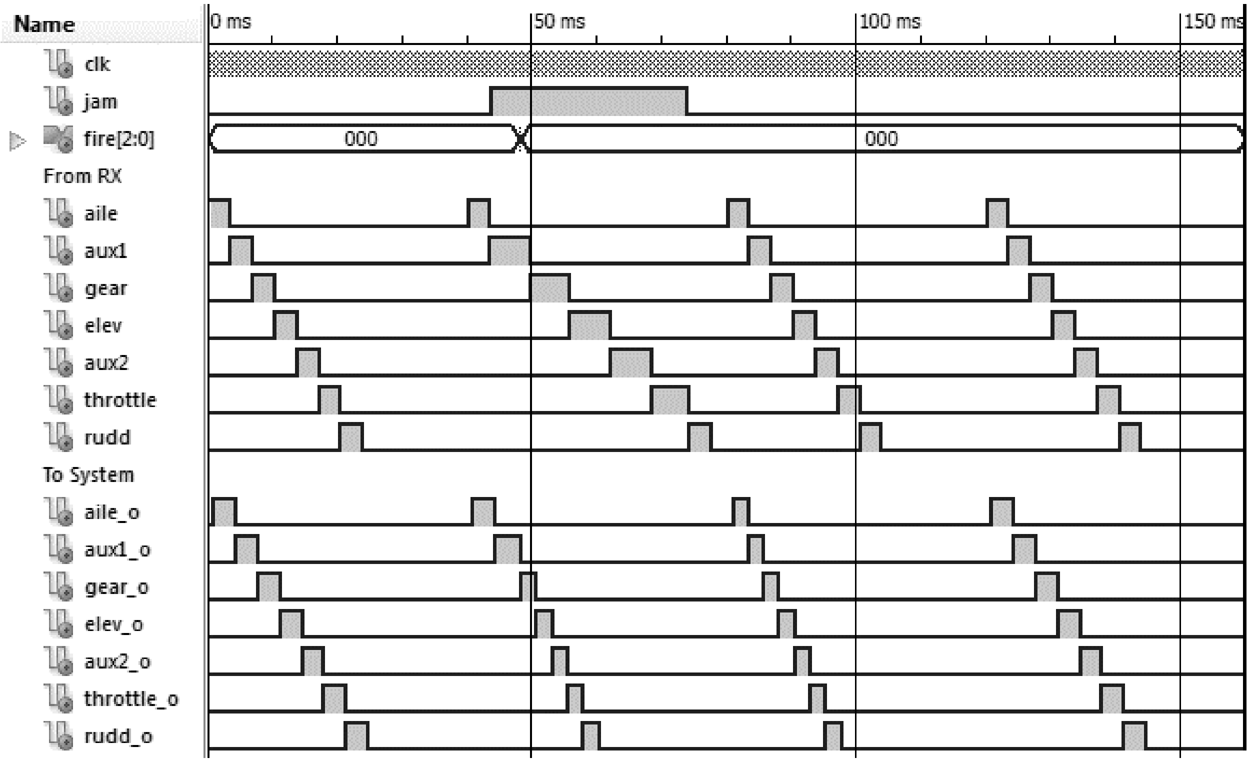


Figure 14. Jamming with the Hardware Sandbox (Long Pulse) Waveform

The reverse is true for pulses which exceed the maximum pulse length as in Figure 14. Again, we are capable of detecting this behavior, as seen in the *fire* signal. The remaining period is spent generating the valid signals. Note that prior to the jamming occurring, the signal to the reset of the system, *aux1_o*, reaches the maximum pulse length, 9%, compared to *aile_o* being sent for 8% and *gear_o* being sent for the default 5%. This shows the seamless transition from still correctly behaved signals, prior to the OVL Cycle Sequence detection, to the generated signals from the virtual receiver. We allow only the minimum to maximum pulse length to be output to the rest of the system.

4.4.3 Jamming Impact – Short Period

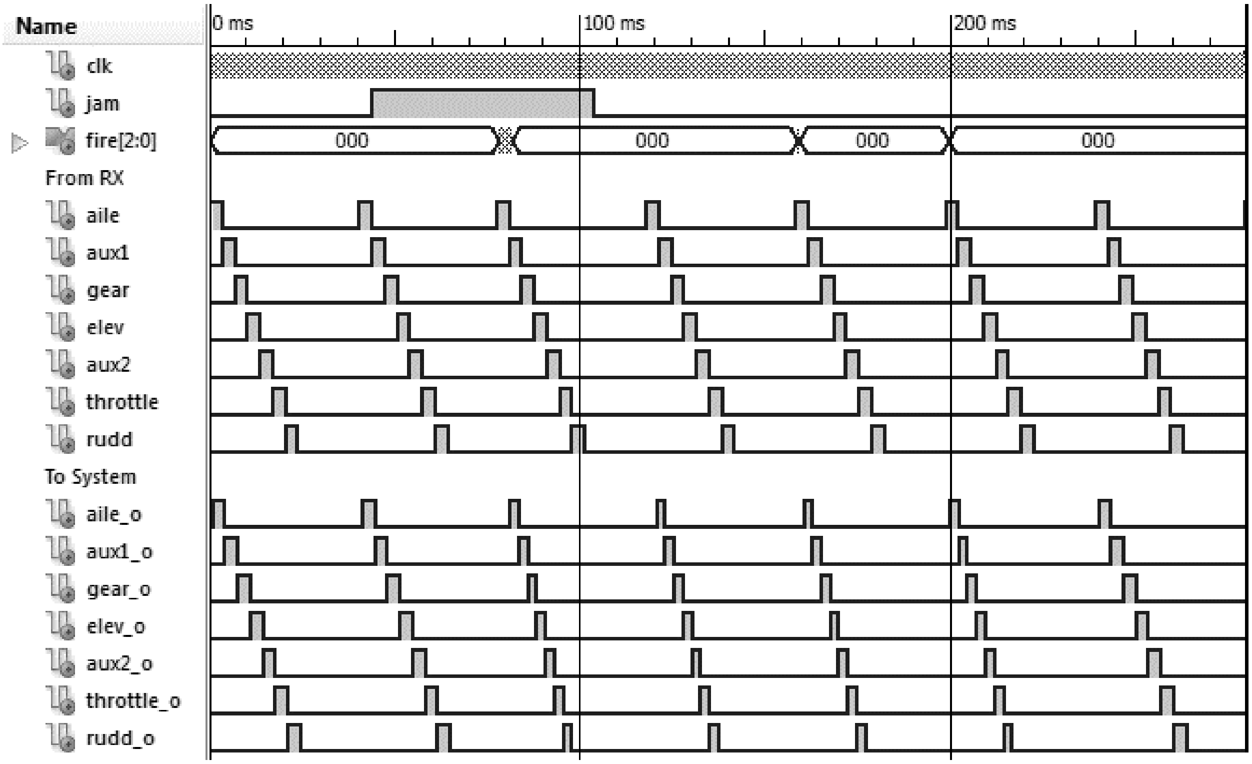


Figure 15. Jamming with the Hardware Sandbox (Short Period) Waveform

For Figure 15, when *jam* is enabled, we decreased the period of the receiver signals by 10% to indicate a faster PWM frequency and observed the results. As you can see, again, we are capable of detecting this change in protocol. Notice, in this case, *fire* is asserted twice in a very short interval. This is due to the signal checker detecting *aile* being asserted (due to the short period, a new period beginning with *aile* starts again) during the interval where all signals should remain low in the correctly-conforming protocol. Thus, *fire* is asserted. *Fire* is reasserted when the next check automatically begins and discovers that *aile* is not high for the required 5% pulse length due to *aile* already being asserted prior due to the shorter period. In either case, the faster signals are never output to the rest of the system, as we can see in the "To System" signals. The virtual receiver takes over to generate signals at the correct frequency.

In this waveform, we simulate possible jitter in our receiver by decreasing the PWM signal frequency by 1% under normal conditions. Due to the short window allowed in both the checker and the virtual receiver, this gives us the ability to dynamically adapt to correct signals. As the waveform shows, after *jam* goes low, the signals are still off, and therefore, trigger *fire* a couple more times. However, after enough time has passed for the signals to become back in sync, the system is corrected for a full period, and the virtual receiver manager switches back to the real signals to be output to the system.

4.4.4 Jamming Impact – Long Period

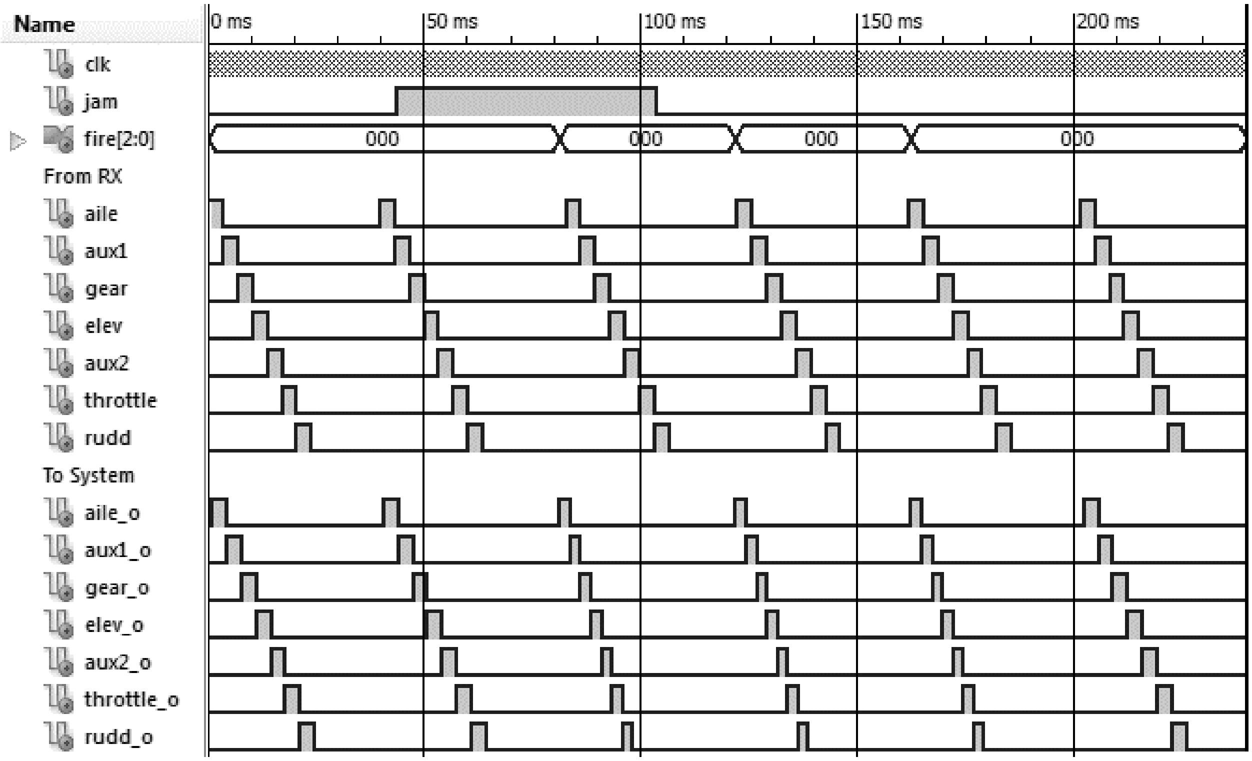


Figure 16. Jamming with the Hardware Sandbox (Long Period) Waveform

In our next simulation test, in Figure 16, we look at the behavior opposite to that in Figure 15. Here we increased the period of our base signals by 7.5% to demonstrate a slower-received PWM frequency. Again, our signal checker detects this behavior when *aile* is never reasserted during the end window to signal the correct start of a new period. Thus, our virtual receiver takes over and begins the signal generation at the correct time. We simulate possible clock jitter in this example with a 1% faster PWM frequency than normal to show how the signal can resynchronize back with our sequence generation logic over time after the jamming signal is no longer present. Again, after the first correct check, we switch from our virtual receiver generation to the real signals seamlessly.

Thus, in all simulation tests, we were successful in detecting the jamming behavior from our assumptions and in the generation of correctly-followed PWM protocol signals to the rest of the system. We demonstrated the seamless takeover from the virtual receiver and back to the real signals when jamming is no longer present.

4.5 Real-World Testing

Outside of simulating, we performed two real-world tests utilizing the Spektrum DX7SE transmitter and AR7010 receiver with the hardware sandbox elements implemented on the Zybo FPGA. In both tests, our objective was to be able to send all valid inputs on the controller and observe the hardware sandbox performing correctly. Our first test, however, focused solely on the signal checker's accuracy, while the second ensured the full hardware sandbox worked correctly in a Linux environment. We simulated a jammed environment by turning off the controller during the test with the expected behavior that *fire* of the OVL Cycle Sequence should be asserted due to the transmission not being received by the receiver. We also tested the ability to resynchronize the system after jamming by turning the controller back on and observing the output.

4.5.1 Signal Checker LED Test

To perform this test, we modified our drone design with only the signal checker included to allow *fire* from the OVL Cycle Sequence to be output to the LEDs of the Zybo FPGA, such that when *fire* is asserted, the LEDs will turn on. We plugged the receiver's seven channels into one of the PMOD connectors on the FPGA and turned on the transmission controller to find the paired receiver. Our new modified, design bitstream was then downloaded onto the board for observation.

Initially, the LEDs were turned off, indicating that *fire* is not asserted and that a problem has not occurred. This matches our expectation as we have not modified anything at this point. Next, we adjusted the various control inputs on the controllers, moving the flight sticks and toggling the gear, elevators, and auxiliary switches. When doing so, again, the LEDs on the Zybo did not blink, meaning our OVL Cycle Sequence is accepting the changed PWM values correctly. Next, we turned off the controller and observed the LEDs. Here, the LED turned on, indicating that *fire* is continually being asserted. This again matches our expectation because as described in 3.2.2, when the controller and receiver are not linked, the receiver outputs the PWM signals at a slower frequency. Finally, we turned the controller back on. When this occurred, the LED on the board stayed lit for a brief period of time before turning off. This behaves exactly like our simulation results where after jamming has finished, there exists an interval where *fire* is still being asserted while it resynchronizes from the deviating clock period. Thus, our real world test with the controller and receiver matched our initial hypothesis and expected behavior of the signal checker under various conditions.

4.5.2 Full Hardware Sandbox Test

For the last real-world test, we added all elements of the hardware sandbox back to the drone design and removed the LED-*fire* output connection. Now, *fire* is being output only to the status register to be read. Because the working drone would utilize the embedded Linux OS to run its software, we created a test C application to read a byte from the contents of the device file associated with the status register IP and its physical address space on a loop every one second. We cross-compiled and placed this application, the character device driver for the status register IP, and all necessary files for booting Linux on the Zybo onto an SD card. When booted, we loaded our device driver kernel module, ran our test application, and observed the results from the still-connected AR7010 receiver and paired DX7SE transmitter.

The results of this test validate our design. As expected, because the controller is paired and sending, the values being read and outputted to the terminal were 0, because *fire* is low. When changing the inputs on the controller, like the previous test, the outputted value was still 0. However, when the controller disconnected, and *fire* was asserted, we were able to read 255 (0xFF), which represents the value from *fire* being connected to every bit in the status register. When we reconnected the controller by turning it back on, the value eventually changed back to 0, meaning *fire* is low again. As a result, our hardware sandbox is capable of reading the deviated protocol assertion in software in Linux for logging.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

Protecting against jamming attacks to hijack unmanned aerial vehicles, or drones, is of utmost priority due to their recent, widespread surge in both civilian and military space. In this thesis, we designed and implemented a system that has the capability of detecting and reacting to jamming attacks to better secure drones. We leveraged our current work of the hardware sandbox on hardware Trojans for the monitor and rule enforcement of the Spektrum AR7010 receiver's control PWM signal protocol from the DX7SE transmitter in our system. We tailored our hardware sandbox specifically for this receiver/transmitter pair. This was accomplished by studying and analyzing the behavior of the Spektrum AR7010 receiver and the protocol it followed. From the established protocol, we designed an algorithm in VHDL to generate the valid sequence bits based on the seven control signals to be used in conjunction with the OVL Cycle Sequence module. In addition, we created a virtual receiver signal generator in VHDL to isolate the potentially jammed receiver from the rest of the system. Our virtual receiver is capable of seamlessly switching from real, transmitted signals to and from the virtually, generated signals without notice from the rest of the system. While our virtual receiver gives us an ability to react via hardware, we also employed memory-mapped status registers to give us an ability to react to attacks via software in a Linux environment.

Our hardware sandbox-based drone design was targeted for use on the Xilinx Zynq processing system and FPGA fabric. We implemented the design for the Digilent Zybo FPGA and analyzed the performance and resource metrics. We proved that despite the addition of hardware in the middle of the receiver and the control IP, we consume virtually identical power and do not significantly affect the timing of the system, while only using a comparatively small amount of board resources.

We have shown functional verification of our system through a number of possible jamming simulation tests of both full and partial jamming and a checker/status register verification using real-world delay and timing with the two real-world tests. Our hardware sandbox performs successfully in the detection of an improper protocol, an achievement also demonstrated via our previous hardware Trojan tests. Our simulation results also demonstrate the isolation requirement of sandboxing by disallowing the incorrect signals from spreading to the system. While future research would need to be performed in order to fully and accurately understand the behavior of our specific transmitter and receiver under partial jamming, our methods and results are still valid for full jamming and can be applied and analyzed for partial jamming in our receiver and future others. Regardless, as jamming aims to disrupt the communication and flight path of a drone, it is anticipated that the control signals to the system would be disrupted too in the case of partial jamming.

While this thesis project was performed utilizing the Spektrum model AR7010 receiver and DX7SE transmitter, our same design approach and philosophy for the hardware sandbox is applicable to other RF drone receiver and transmitters. The only condition required for use is for the receiver to follow a well-defined, consistent protocol when interfacing to the system, which is expected in any control system. With this, a hardware sandbox signal checker and virtual receiver can be defined and designed specifically for the equipment used to detect and prevent malicious denial-of-service and other incorrect functional-based attacks. Therefore, our work in this thesis project on hardware sandboxing has universality.

5.2 Future Work

Our work on hardware sandboxing and its use in drone systems to prevent jamming is still young and offers many avenues for future development. In this thesis, we discuss a few possible ways to improve and advance the current system, design, and implementation of the hardware sandbox.

5.2.1 Automatic Generation of a Hardware Sandbox

While our hardware sandbox is designed specifically for the Spektrum AR7010 receiver, there are many other possible commercial or custom receivers to use, each possibly using a different control protocol for interfacing to the system. Designing a hardware sandbox, in particular, the properties checker for the IP under sandbox, can require an extensive amount of time and testing to ensure correctness. We would like to eliminate this cost by automatically generating the sandbox. As mentioned earlier, the design of a hardware sandbox first requires knowledge of the specific protocol and properties of the signals under monitor. Therefore, if we can define the signal properties in a universal manner through a language, we can use tools to parse and generate the required principle components of the sandbox. This future work has implications beyond drone attacks, as the hardware sandbox can be applied in many situations, such as the aforementioned hardware Trojan isolation.

There are some pre-existing, open-source tools which can accomplish the compilation of a universal property language into HDL sources. One such tool in particular, Lily [21], while based on Linear Temporal Logic synthesis [22], allows for the compilation of LTL-style PSL-defined properties to output, if possible, a synthesizable Verilog module. We can extend and use this work as a base for hardware sandbox generation.

5.2.2 Further Exploration of Partial Jamming

As discussed earlier, the effects of only partially jamming a transmission are assumed and not fully known for our receiver model. Therefore, a further analysis and study would need to be performed to fully understand the behavior in order to validate our assumptions. Doing so requires building a jammer that is capable of jamming a small subset of the controller's transmission and observing the effect.

5.2.3 Adaptive Flight Measures in Conjunction with Hardware Sandboxing

Our hardware sandbox offers the ability to respond to jamming behavior in both hardware and software after detection from the signal checker. At the current moment, we utilize in hardware a virtual receiver to generate correct, yet basic PWM signals to the system in case of jamming, while in software, use the status register to make a log if jamming occurred. However, more advanced adaptive flight measures are possible after detection. For example, routines in the ArduCopter software could be devised to return the drone to its original source away from the attacker when jamming occurs.

In hardware, we can use adaptive techniques such as partial reconfiguration to thwart attackers when jammed. This measure would require the integration of the receiver and its ability to decode a transmission into the seven PWM channels into our hardware from the current external module. For example, if jammed, we could load a new receiver's transmission decoding module IP in a reconfigurable partition that utilizes a different set of spreading codes. While this would mitigate jamming for attackers who know at least a subset of the spreading codes and how to jam a certain frequency channel, this would also affect the legitimate controller and transmitter. Thus, additional exploration would be needed on how to securely notify the flight controller that the spreading codes for the transmission have been changed.

Bibliography

[1] Anderson, Chris. "Agricultural Drones." *MIT Technology Review*. Web. 11 Apr. 2016.

[2] "Drones: What Are They and How Do They Work?" *BBC News*. 31 Jan. 2012. Web. 11 Apr. 2016.

[3] Lubold, Gordon. "Pentagon to Sharply Expand U.S. Drone Flights Over Next Four Years." *WSJ*. The Wall Street Journal, 16 Aug. 2015. Web. 11 Apr. 2016.

[4] A. Y. Javaid, W. Sun, V. K. Devabhaktuni and M. Alam, "Cyber Security Threat Analysis and Modeling of an Unmanned Aerial Vehicle System," *Homeland Security (HST), 2012 IEEE Conference on Technologies for*, Waltham, MA, 2012, pp. 585-590.

[5] Grover, Kanika, Lim, Alvin & Yang, Qing (2014). Jamming and Anti-jamming Techniques in Wireless Networks: A Survey. *Int. J. Ad Hoc Ubiquitous Comput., 17*, 197-215.

[6] Wenyuan Xu, Ke Ma, W. Trappe and Yanyong Zhang, "Jamming Sensor Networks: Attack and Defense Strategies," in *IEEE Network*, vol. 20, no. 3, pp. 41-47, May-June 2006.

[7] Bayraktaroglu, Emrah, King, Christopher, Liu, Xin & Noubir, Guevara and (2013). Performance of IEEE 802.11 Under Jamming. *Mob. Netw. Appl., 18*, 678-696.

[8] S. Fang; Y. Liu; P. Ning, "Wireless Communications under Broadband Reactive Jamming Attacks," in *IEEE Transactions on Dependable and Secure Computing* , vol.PP, no.99, pp.1-1

[9] C. S. Tsang, "Jamming Detection and SNR/SNJR Estimation," *Aerospace Conference, 2011 IEEE*, Big Sky, MT, 2011, pp. 1-7.

[10] D. Giustiniano, V. Lenders, J. B. Schmitt, M. Spuhler, and M. Wilhelm, “Detection of Reactive Jamming in DSSS-Based Wireless Networks,” in *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec ’13. New York, NY, USA: ACM, 2013, pp. 43–48.

[11] M. Tehranipoor and F. Koushanfar, “A Survey of Hardware Trojan Taxonomy and Detection,” Design Test of Computers, IEEE, vol. 27, no. 1, pp. 10–25, Jan 2010.

[12] S. Bhunia, M. Hsiao, M. Banga, and S. Narasimhan, “Hardware Trojan Attacks: Threat Analysis and Countermeasures,” Proceedings of the IEEE, vol. 102, no. 8, pp. 1229–1247, Aug 2014.

[13] W. Venema, “Isolation Mechanisms for Commodity Applications and Platforms,” IBM, Tech. Rep. RC24725 (W0901-048), 01 2009.

[14] O. W. Group. “Open Verification Library (OVL) Working Group.” *Accellera*. Web 11 Apr. 2016.

[15] “IEEE Standard for Property Specification Language (PSL),” IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005), pp. 1–182, April 2010.

[16] “Resources.” *Trust-HUB*. Web. 11 Apr. 2016.

[17] “Aircraft Rotations.” *Glenn Research Center NASA*. Web. 11 Apr. 2016.

[18] “ZYBO™ FPGA Board Reference Manual.” *Digilent*. 11 Apr. 2016. Web. 12 Apr. 2016.

[19] “Clocking Wizard v5.1.” *LogiCORE IP Product Guide.* Xilinx. 1 Apr. 2015. Web. 12 Apr. 2016.

[20] D. Giustiniano, V. Lenders, J. B. Schmitt, M. Spuhler and M. Wilhelm, "Detection of reactive jamming in DSSS-based wireless networks", *Proc. 2013 ACM WiSec*, pp. 43-48

[21] “Lily - a LInear Logic sYnthesizer.” *IAIK – TU Graz.* Web. 12 Apr. 2016.

[22] B. Jobstmann and R. Bloem, "Optimizations for LTL Synthesis," *2006 Formal Methods in Computer Aided Design*, San Jose, CA, 2006, pp. 117-124.