A Defense-In-Depth Security Architecture for Software Defined Radio Systems

Seth D. Hitefield

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

December 2, 2019
Blacksburg, Virginia

Keywords: Wireless Communications, Software Defined Radio, Security, Isolation

A Defense-In-Depth Security Architecture for Software Defined Radio Systems

Seth D. Hitefield

(ABSTRACT)

Modern wireless communications systems are constantly evolving and growing more complex. Recently, there has been a shift towards software defined radios due to the flexibility software implementations provide. This enables an easier development process, longer product lifetimes, and better adaptability for congested environments than conventional hardware systems. However, this shift introduces new attack surfaces where vulnerable implementations can be exploited to disrupt communications or gain unauthorized access to a system. Previous research concerning wireless security mainly focuses on vulnerabilities within protocols rather than in the radios themselves. This dissertation specifically addresses this new threat against software radios and introduces a new security model intended to mitigate this threat. We also demonstrate example exploits of waveforms which can result in either a denial-of-service or a compromise of the system from a wireless attack vector. These example exploits target vulnerabilities such as overflows, unsanitized control inputs, and unexpected state changes.

We present a defense-in-depth security architecture for software radios that protects the system by isolating components within a waveform into different security zones. Exploits against vulnerabilities within blocks are contained by isolation zones which protects the rest of the system from compromise. This architecture is inspired by the concept of a microkernel and provides a minimal trusted computing base for developing secure radio systems. Unlike other previous security models, our model protects from exploits within the radio protocol stack itself and not just the higher layer application. Different isolation mechanisms such as containers or virtual machines can be used depending on the security risk imposed by a component and any security requirements. However, adding these isolation environments incurs a performance overhead for applications. We perform an analysis of multiple example waveforms to characterize the impact of isolation environments on the overall performance of an application and demonstrate the overhead generated from the added isolation can be minimal. Because of this, our defense-in-depth architecture should be applied to real-world, production systems. We finally present an example integration of the model within the GNU Radio framework that can be used to develop any waveform using the defense-in-depth security architecture.

A Defense-In-Depth Security Architecture for Software Defined Radio Systems

Seth D. Hitefield

(GENERAL AUDIENCE ABSTRACT)

In recent years, wireless devices and communication systems have become a common part of everyday life. Mobile devices are constantly growing more complex and with the growth in mobile networks and the Internet of Things, an estimated 20 billion devices will be connected in the next few years. Because of this complexity, there has been a recent shift towards using software rather than hardware for the primary functionality of the system. Software enables an easier and faster development process, longer product lifetimes through over-the-air updates, and better adaptability for extremely congested environments. However, these complex software systems can be susceptible to attack through vulnerabilities in the radio interfaces that allow attackers to completely control a targeted device. Much of the existing wireless security research only focuses on vulnerabilities within different protocols rather than considering the possibility of vulnerabilities in the radios themselves. This work specifically focuses on this new threat and demonstrates example exploits of software radios. We then introduce a new security model intended to protect against these attacks.

The main goal of this dissertation is to introduce a new defense-in-depth security architecture for software radios that protects the system by isolating components within a waveform into different security zones. Exploits against the system are contained within the zones and unable to compromise the overall system. Unlike other security models, our model protects from exploits within the radio protocol stack itself and not just the higher layer application. Different isolation mechanisms such as containers or virtual machines can be used depending on the security risk imposed by a component and any security requirements for the system. However, adding these isolation environments incurs a performance overhead for applications. We also perform a performance analysis with several example applications and show the overhead generated from the added isolation can be minimal. Therefore, the defense-in-depth model should be the standard method for architecting wireless communication systems. We finally present a GNU Radio based framework for developing waveforms using the defense-in-depth approach.

Dedication

To my family, friends, and colleagues who have always supported me.

Acknowledgments

There are many people I would like to thank for helping me throughout my Ph.D.; it would not have been possible for me to achieve this goal without their support in my life.

First, I would like to thank my advisors Dr. Charles Clancy and Dr. Ali Butt for their vision, support, and encouragement during this process. Your guidance was integral in helping me achieve this goal. I would also like to thank Dr. Allen MacKenzie, Dr. Yaling Yang, and Dr. Jonathan Black for taking the time and effort to serve on my committee. While not on my committee, I would also like to thank Dr. Joseph Ernst for guidance in helping me set up my committee and direct me on a path to finishing and Dr. Dan Doyle for his willingness to discuss my progress over the last semester.

Next, I have been very fortunate to work at the Hume Center and Space@VT during my time in graduate school and have the opportunity to work on multiple research projects with many amazing faculty, staff, and graduate students. I have greatly enjoyed my time at Virginia Tech and look forward to continued collaboration in the future. I want to thank Michael Fowler, Sonya Rowe, Zach Leffke, Bill Clark, Dr. Alan Michaels, Dr. Chris Headley, and Dr. Joseph Gaeddert for all of your constant support and guidance. I also want to thank Bryse Flowers for his help with the testing framework and willingness to give valuable feedback on papers and thank Dr. Jason McGinthy for keeping me grounded when I needed to decompress and get away from the lab.

Finally, I would like to thank my family, friends, and my Northstar Church community for their constant love, support, and encouragement throughout my Ph.D. and my time in Blacksburg. Without you all, this would not have been possible. Thank you all.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ASIC	Application Specific Integrated Circuitry
ASLR	Address Space Layout Randomization
bogo/s	Bogus Operations per Second
C-RAN	Cloud Radio Access Network
CIA	Confidentiality, Integrity, Availability
CORBA	Common Architecture Request Broker Architecture
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DDOS	Distributed Denial of Service
DEP	Data Execution Prevention
DOS	Denial of Service
DSA	Dynamic Spectrum Access
DSP	Digital Signal Processing
FEC	Forward-Error-Correction
FIR	Finite Impulse Response
FM	Frequency Modulation
GFSK	Gaussian Frequency Shift Keying
GMSK	Gaussian Minimum Shift Keying
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communications

HSPA	High Speed Packet Access
IDS	Intrusion Detection Systems
IoE	Internet of Everything
IoT	Internet of Things
IPC	Inter-Process Communication
IT	Information Technology
LTE	Long-Term Evsolution
MBps	Megabytes per Second
Mbps	Megabits per Second
MILS	Multiple Independent Levels of Security/Safety
MIMD	Multiple-Instruction-Multiple-Data
MPU	Memory Protection Unit
Msps	Megasamples per Second
MTU	Maximum Transmission Unit
NFV	Network Function Virtualization
NOP	No Operation
OFDM	Orthogonal Frequency-Division Multiplexing
OSI	Open Systems Interconnection
OTA	Over the Air
PID	Process ID
POSIX	Portable Operating System Interface
RAN	Radio Access Network
RCE	Remote Code Execution
RF	Radio Frequency
RPC	Remote Procedure Call

SCA	Software Communications Architecture
SDN	Software Defined Network
SDR	Software Defined Radio
SIMD	Single-Instruction-Multiple-Data
SNR	Signal-to-Noise Ratio
SRM	Secure Radio Middleware
SSH	Secure Shell
SWaP	Size, Weight and Power
TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TMSI	Temporary Mobile Subscriber Identity
USRP	Universal Software Radio Peripheral
VM	Virtual Machine
VOLK	Vector Optimized Library of Kernels

Chapter 1

Introduction

Wireless communications systems have become ubiquitous in everyday life and many times we are unaware of the considerable number of wireless devices we depend on and interact with on a regular basis. Innovations in embedded hardware and wireless technology have enabled the massive growth of cellular networks, mobile broadband, and the Internet-of-Things (IoT). Devices are growing smaller, yet are more powerful, capable, and connected than ever before. Predictions for the number of Internet connected devices widely vary [1], with many estimates stating 20+ billion devices could be connected within the next few years [2, 3]. A key enabling technology underlying all of these networks and smart devices is wireless communications.

Because of the massive growth of these wireless networks, security should be a primary concern for systems since the possible attack surface is practically unbounded. Unlike traditional Information Technology (IT) networks and systems, the wireless networks have no well defined borders or endpoints that can be secured against attack; there are simply too many devices and inter-connected networks to consider. With the rapid pace of technology, security for these systems is often overlooked or is simply a low priority and is added as an afterthought. Many systems either do not have the proper security capabilities, are poorly designed or configured, or are plagued by poor programming practices. This results in a high probability they are vulnerable to attack [2, 4, 5]. Security is lagging behind in industry as a whole, and these vulnerable systems present a tempting target to attackers who can use them as a vector for attacking other, more valuable systems.

The wireless interfaces themselves are one aspect that can be quickly overlooked from a security perspective. If security is a consideration during development, then the primary focus is usually the protocols implemented by each interface rather than on the implementation itself. However, these implementations themselves can be vulnerable to attack. Smart devices can easily include multiple wireless interfaces and can connect to multiple networks, so this overall problem is only exacerbated. For example, modern smart phones typically support multiple generations of cellular networks (5G, 4G LTE, 3G), local networks (Wi-Fi, Bluetooth), and other receivers (GPS). The research presented in this dissertation focuses specifically on vulnerabilities that can exist within these wireless interfaces and implementations.

1.1 Software Trend

Wireless systems have greatly evolved over recent years; modern systems are exponentially more complex, powerful, and smaller than ever before. There has been a growing push towards implementing wireless systems using software rather than hardware due to this complexity. Software based systems provide many different benefits over traditional hardware systems because of the flexible nature of software [6]. They are easily modified to address changing application requirements and are capable of supporting both current and future wireless standards.

With conventional systems, all implemented functionality is fixed, so new Application Specific Integrated Circuitry (ASIC) must be developed to support new standards and protocols. Developing and testing this new hardware can be a timely and expensive process. On the other hand, software implementations are flexible by nature, allowing multiple different wireless standards to be implemented on the same platform; resources can be efficiently shared and reused when needed. Software defined systems also result in easier and faster development and longer overall product lifetimes, since supporting new standards requires only a software update. Compared to conventional hardware systems, maintaining and upgrading production systems with new functionality is far simpler and cheaper.

Software Defined Radios (SDRs) are a fundamental technology for developing the next-generation of wireless systems. The key concept is the digital signal processing functionality traditionally implemented in hardware is now implemented with software. While the concept of SDRs was first introduced in the 1990s [7], the limited performance of older hardware systems limited SDR use mainly to research, development, and testing roles. Recent embedded systems are now powerful enough to implement production SDRs for real-world systems. For example, NVIDIA's Tegra 4i embedded processor includes an Integrated i500 SDR modem and is capable of supporting multiple cellular standards such as LTE, HSPA+, and GSM [6].

1.1.1 New Attack Surface

However, this shift towards software implementations creates new attack surfaces for wireless systems and introduces new security threats that are not necessarily obvious (specifically the implementation itself). Wireless systems always have certain security threats, but the nature of SDRs creates a new set of threats specific to the software implementation itself. Vulnerabilities can exist in an implementation due to programming errors or faulty designs, and could easily be exploited by an attacker in order to completely compromise the targeted system. The more complex a software implementation becomes, the greater the chance of such a vulnerability existing [8]. An example block diagram of SDR components and this new attack surface is shown in Figure 1.1.

This issue is compounded due to the complexity and rapidly evolving nature of wireless

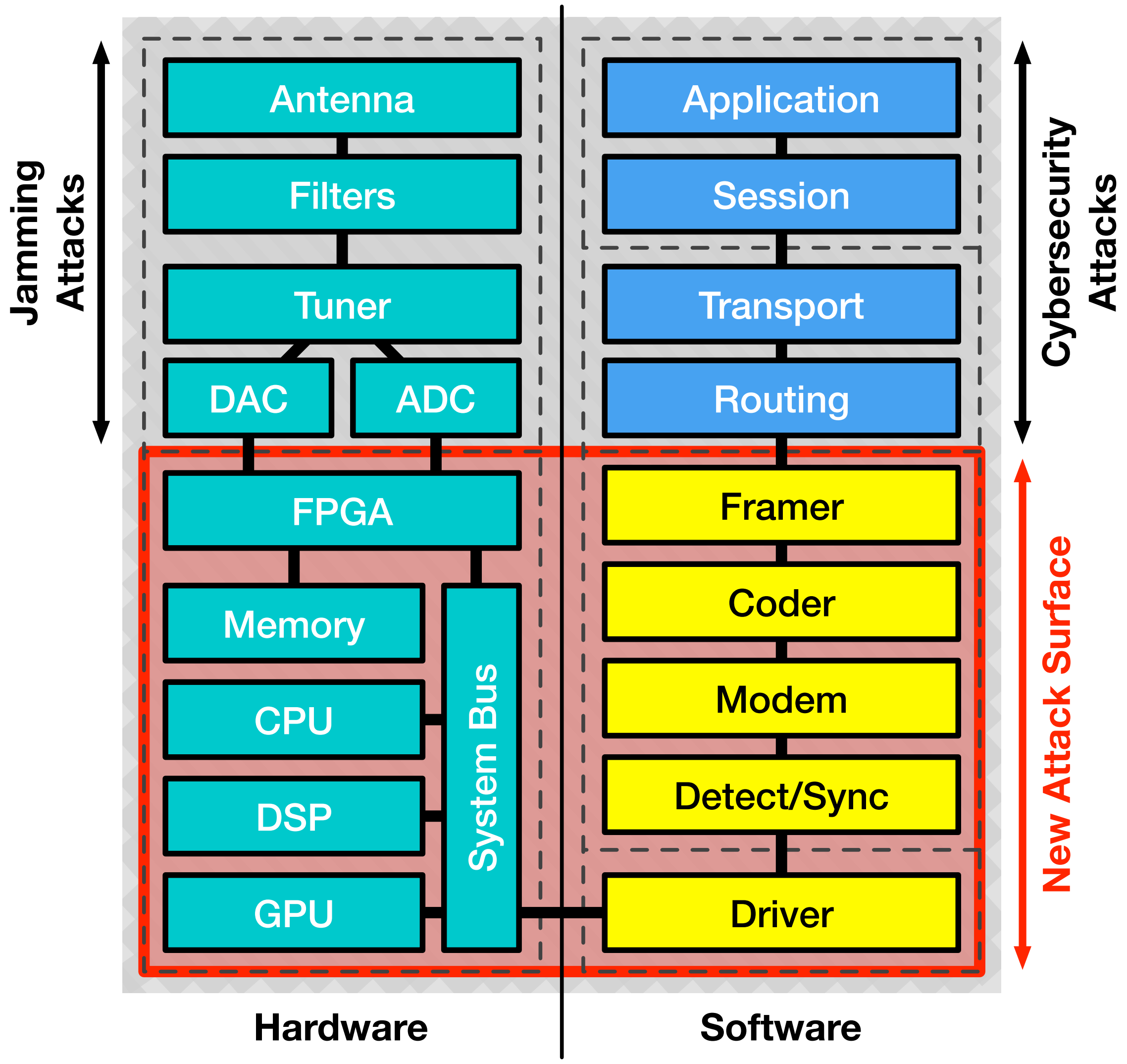


Figure 1.1: Example attack surface of a software defined radio. The highlighted components (which were traditionally implemented in hardware and are now implemented using software) could have vulnerabilities that allow attackers to compromise the system or embedded device.

standards. With faster development timelines, security can be neglected in order to meet deadlines; programming mistakes or design flaws can create vulnerabilities such as buffer overflows that can be easily overlooked during development. There can also be a tendency to focus on end-user functionality during development and only address security as different issues arise after deployment. These systems also do not endure the same level of rigorous testing as a conventional hardware system which increases the chances that vulnerabilities can exist. As standards become more complex and more systems are implemented in software, the probability of serious vulnerabilities existing in implementations only increases.

However, most of the past research in wireless security has been focused toward vulnerabilities in specific protocols and standards rather than the implementation itself [9, 10, 11, 12]. Attackers look for vulnerabilities to exploit in protocols, and defenders attempt to patch any discovered vulnerability to secure the system. The evolution of Wi-Fi security is a prime example of how vulnerabilities were discovered and new protocols were developed to address these flaws [9, 10].

Historically, software and hardware required to test or exploit wireless systems were very expensive, highly specialized, or did not exist. This sometimes leads to wireless security being less of a priority and easily neglected. With software radio and other low-cost hardware now readily available, the barriers to wide-spread penetration testing and exploitation of wireless systems have been significantly lowered which makes security a primary concern when developing systems.

1.1.2 CyberElectronic Warfare

Traditionally, attacks on wireless systems (jamming) attempt to disrupt signals at the lowest layer of the system stack: the physical layer. However, with this new attack surface, systems are now becoming susceptible to traditional cyber-security exploits that allow attackers to compromise the system. Attacks have begun to move up the network stack into the higher layers of the software such as the routing and data layers of the system.

This merging of the traditionally separate electronic-warfare and cyber-security domains results in a new “cyber-electronic warfare” area where traditional cyber-security techniques are used to disrupt wireless communications [13]. Even though standards are heavily vetted prior to production, each specific vendor’s implementation of those standards will be unique and may have weaknesses. Specifically, if the behavior of the system can be modified in a way that causes a denial-of-service (DoS) by a software attack, then this can be considered persistent jamming (knock a receiver offline without the need for constant jamming).

1.2 Motivation

Unfortunately, security still tends to be an afterthought during the development of wireless systems; there can be a tendency to develop and deploy products rapidly and only tackle security issues whenever they arise. The growing shift towards software implementations compounds this since patching security issues can be as simple as publishing a minor software update over-the-air (OTA). With the number of devices constantly increasing, standards becoming more complex, and more systems shifting over to software implementations, threats are becoming more real against systems. There needs to be a fundamental shift in developer's mindsets where security is a primary design requirement from the earliest stages of development. Vulnerabilities can always exist in software even if security is a major focus, but including security from the beginning ensures a stronger foundation for addressing future vulnerabilities once they are discovered.

Additionally, the shift to software defined systems has created a new set of threats and a new attack surface simply due to the nature of software. As systems become more complex, there is a greater chance that programming mistakes or design flaws can lead to vulnerabilities. Any vulnerabilities in the firmware of a radio could be used to exploit and compromise the targeted system. However, much of the existing wireless security research focuses on protocols or privacy issues and not on the security of the implementation itself. A wireless protocol may be reasonably secure, but if the underlying implementation was not properly designed, a system could still be susceptible to attack. There exists a critical need to develop security techniques that specifically address this new class of threats and enable more rigid and secure architectures for wireless systems .

The goal of this dissertation is to highlight the lack of focus in this area and help trigger a change in the development mindset towards prioritizing security as a critical design requirement for SDRs. We specifically address this new class of threats to SDRs rather than focusing on specific protocols or wireless standards. Therefore, the main focuses of this research are: 1) demonstrating example exploits of vulnerabilities within SDR waveforms, 2) presenting a defense-in-depth security architecture that uses isolation environments to separate different components of the system, and 3) understanding and characterizing the generic performance overhead from adding isolation environments to waveforms. This defense-in-depth architecture provides a methodology for protecting SDRs from these types of exploits. By using isolation within the system to separate components, exploits of a specific component are prevented from compromising the entire system. This approach can also be extended beyond SDRs as it provides the underlying methodologies needed for developing any secure, software defined communications systems.

1.3 Contributions

As mentioned, the goal of this dissertation is highlighting a new class of security threats to SDRs and developing methodologies for defending against these attacks. This research provides several contributions toward this goal of secure SDR systems, which are listed below:

1. **Wireless exploitation of SDRs**
   Much of the past work in wireless security research has focused on exploits against protocols rather than exploits against the radio implementations themselves. There is a lack of focus on these types of vulnerabilities, and as more systems shift to software implementations, the risk of these vulnerabilities increases. This contribution

details several examples of vulnerabilities in software defined radio systems and also demonstrates how an attacker can exploit them to compromise the system, modify the behavior of the system, and/or cause a significant denial-of-service against the system. Defending against these types of vulnerabilities is the primary motivating factor for the defense-in-depth SDR architecture presented in this work.

2. **A Defense-in-Depth Architecture for SDRs**
   The defense-in-depth architecture is both a model and guideline for developing wireless systems with security as a critical focus of the entire development process. Security is added to the system from the ground up, which builds a foundation for more secure wireless systems. At its core, the architecture is based on the principles of isolation and least privilege, where all of the components of the system are separated into isolated zones, in order to prevent exploits against single components from affecting other components in the system. Various environments with different levels of isolation (such as containerization and virtualization) can be used, with the highest risk components being placed in the most isolated environments. Adding these multiple layers of isolation into an SDR implementation provides better overall security and resiliency than a monolithic system.

3. **Characterization and analysis of waveform performance in isolation environments**
   Using these different isolation environments to separate components of a radio application creates additional processing overhead that must be taken into account for the overall system implementation. Depending on the defined system requirements and the targeted hardware for the system (for example cloud computing infrastructures versus embedded systems), some isolation mechanisms and their associated overhead may not be acceptable for a given application. For example, cloud infrastructure servers are typically very powerful and contain massive amounts of resources, as well as, hardware support for technologies like virtualization. Embedded systems, on the other hand, are resourced constrained and lack some of these hardware assisted features. This contribution explores the performance impact of added isolation methods on the maximum throughput of various waveforms tested in several different system configurations. We developed a testing framework capable of executing various tests within multiple isolation environments that can also dynamically modify the host system configuration in order to test performance under different system conditions. Specifically, multiple waveforms implemented in GNU Radio and LiquidDSP frameworks were tested on two common environments: virtualization (with Oracle VirtualBox [14]) and containerization (with Docker [15]). Results showed that both environments can have minimal impact on the overall performance of an application and therefore this defense-in-depth approach should be used in production systems.

4. **Example implementation and approaches for secure, SDR applications**
   For this contribution, we developed an example framework that integrates with a

commonly used open source SDR framework (GNU Radio) that allows waveforms to be developed and deployed using our defense-in-depth architecture. This framework is designed to require minimal changes to existing GNU Radio flowgraphs in order to use the defense-in-depth approach. We also explore the advantages of the secure infrastructure, as well as, address disadvantages and limitations of the approach.

1.4 Publications

Below is a list of my current publications:

Directly related conference papers:

- **Seth D. Hitefield**, Zach Leffke, Jon T. Black, "A Open-Source Satellite Communications Simulator Using GNU Radio", IEEE Aerospace 2019.
- **S. D. Hitefield**, M. Fowler, T. Clancy, "Exploiting Buffer Overflow Vulnerabilities in Software Defined Radios", IEEE Conference on Computer and Information Technology (CIT), July 2018 [**Best Student Paper Award**].
- **Seth Hitefield**, T. Charles Clancy, "Flowgraph Acceleration with GPUs: Analyzing the Benefits of Custom Buffers in GNU Radio", Proceedings of the GNU Radio Conference, September 2016.
- **S. D. Hitefield**, M. Fowler, C. Jennette, T. Clancy, "Link Hijacking Through Wireless Exploitation of a Vulnerable Software Defined Waveform", Military Communications Conference (MILCOM) 2014 [Restricted].
- **S. D. Hitefield**, V. Nguyen, C. Carlson, T. O'Shea and T. Clancy, "Demonstrated Physical and LLC Layer Attack and Mitigation Strategies for Wireless Communication Systems", IEEE CNS, Cognitive Radio and Electromagnetic Spectrum Security (CRESS) 2014.
- C. Carlson, V. Nguyen, **S. Hitefield**, T. O'Shea, T. Clancy, "Measuring Smart Jammer Strategy Efficacy Over the Air", IEEE CNS, Cognitive Radio and Electromagnetic Spectrum Security (CRESS) 2014.

Indirectly related conference papers:

- Gustavo Gargioni, **Seth D. Hitefield**, Minzhen Du, Nicholas Angle, Hovhannes Avagyan, Gavin Brown, Zachary Leffke, Stephen Noel, Kevin Shinpaugh, Jonathan Black., "VCC Ceres: Challenges and Lessons Learned in a Undergraduate Cubesat Project", IEEE Aerospace 2020.

- Zach Leffke, **Seth D. Hitefield**, Jonathan T. Black, "Introducing SatMF: The Satellite Metadata Format", IEEE Aerospace 2020.
- Timothy J. O'Shea, **Seth Hitefield**, Johnathan Corgan; "End-to-end radio traffic sequence recognition with recurrent neural networks", 2016 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2016.
- Alan J. Michaels, William C. Headley, Joseph M. Ernst, **Seth D. Hitefield**; "Enhanced PHY-layer security via co-channel underlays", IEEE 35th International Performance Computing and Communications Conference, IPCCC 2016.
- **Seth Hitefield**, Zach Leffke, Michael Fowler, Robert W. McGwier, "System Overview of the Virginia Tech Ground Station", IEEE Aerospace 2016.
- Paul David, **Seth Hitefield**, Zach Leffke, William C. Headley, Robert W. McGwier, "Implementation of an Actor Framework for a Ground Station", IEEE Aerospace 2016.

Technical Reports:

- **Seth Hitefield**, Bill Clark, Zach Leffke, Robert W. McGwier, "Virginia Tech Fox-1 Camera", AMSAT Symposium 2015.
- M.Adams, **S. D. Hitefield**, B. Hoy, M. Fowler, T. Clancy, "Application of Cybernetics and Control Theory for a New Paradigm in Cybersecurity", *arXiv Cryptography and Security*, arXiv:1311.0257 [cs.CR], November 2013.
- Krishan, Neelima; **Hitefield, Seth**; Clancy, T. Charles; McGwier, Robert W.; Tront, Joseph G.; "Multipersona Hypovisors: Securing Mobile Devices through High-Performance Light-Weight Subsystem Isolation", Computer Science Technical Report, Virginia Tech, June 2013.

1.5 Dissertation Outline

This dissertation starts with a discussion in Chapter 2 of various background topics that are relevant to the overall work presented in this work. We first describe the concept of software defined radio and examine and compare different open-source frameworks for developing SDR waveforms and applications. Also, we briefly introduce emerging technologies in wireless systems based on SDRs that will become the backbone for future generations of wireless networks and mobile devices. This chapter also focuses on different computer and network security concepts and principles that are important for developing secure systems. Finally, we conclude this chapter with an introduction and comparison of different types of virtualization that can be used for isolating applications and managing resources.

Chapter 3 presents existing research on software defined radio security and summarizes the different threats that are inherent to wireless systems and specifically software defined radios. We also cover examples of recent exploits of the firmware in different consumer wireless devices where attackers used exploits to successfully compromise a mobile device. Finally, we present previous models and approaches for developing secure software radios and discuss how the approaches are limited and fail to protect against exploits of vulnerabilities in the radio implementation itself.

Chapter 4 provides the main motivation for our research and demonstrates why the security of SDRs is critical and needs to be addressed. We first, present several examples of vulnerabilities that can exist in a SDR and demonstrate exploits against them from a wireless attack vector that result in either a denial-of-service or compromise of the system. This includes: control flow manipulation, control parameter corruption, heap-based buffer overflows, and stack-based buffer overflows.

Chapter 5 presents our defense-in-depth architecture for secure software radio systems. The architecture provides security by creating multiple layers of isolation within the radio itself that separate different components in the system. In this chapter, we define the different planes of the model (security, control, and data) and describe the different components that would exist within each plane. This chapter wraps up with some of the challenges and tradeoffs associated with our security architecture.

In Chapter 6, we characterize the performance overhead for executing SDR applications within different isolation environments. Specifically, we present and analyze the results of example waveforms using GNU Radio and LiquidDSP that are tested in two common isolation environments: containers (Docker) and virtual machines (VirtualBox). Tests also included various utilities that stressed specific components of the system, and tests with split flowgraphs to better understand the impact of inter-process communication and isolation within waveforms. We also describe the setup of the test framework we developed to automate all of the tests and briefly provide additional insights based on the testing results.

In Chapter 7, we present an example framework that integrates with the GNU Radio framework and allows waveforms to be developed and deployed using the defense-in-depth architecture. This chapter also discusses some of the limitations of the framework and presents some example use-cases of how the architecture can be applied to wireless systems.

Finally, in Chapter 8, we present a summary of our research and discuss future work based on this research.

Chapter 2

Background

The research presented in this dissertation addresses the security of SDR implementations and other software defined wireless communication systems. To that extent, this chapter addresses various background topics important for understanding the material presented later in this work. First, we briefly discuss the basics software radios and highlight several open source SDR frameworks and various use-cases for SDR. We also compare the differences of some frameworks such as the overall system model and data-flow architecture. Second, we introduce several different principles of cyber-security and various models for implementing security within systems. For example, exploits against systems are typically characterized by the overall effects that they have on the targeted, such as a loss of confidentiality, integrity, or availability (CIA) in the system. Other security concepts discussed include: the principle of least privilege, trusted computing base, security-by-design, and multiple layers of independent security (MILS). Next, we review various types of common vulnerabilities that can exist in software especially as software systems become more complex. We also briefly review existing defensive methods for these types of vulnerabilities, and how these defenses can be bypassed to still compromise the system. The final section of this chapter reviews virtualization and containerization techniques which are key components that are used in the defense-in-depth model presented later in this dissertation.

2.1 Software Defined Radio

The concept of software defined radio was first introduced in the 1990s [7, 16]. From a high-level perspective, it is a wireless system where the signal processing functionality of the radio is implemented in software rather than hardware. An example of a software defined receiver for broadcast FM (Frequency Modulation) radio signals is shown in Figure 2.1. Conventional hardware systems typically have the lower layers of the network stack (OSI model) implemented as hardware or Application Specific Integrated Circuitry (ASIC) which limits the flexibility of the system. With SDRs, the software implementation extends to the physical layer itself (shown in Figure 2.2), which is very flexible and allows a system's behavior to be modified with a simple software update rather than completely redesigning and rebuilding hardware. This is extremely useful from research, prototyping, and security perspectives. But, higher latencies and lower throughputs for SDR implementations have somewhat limited their real-world use cases. As the computing hardware and specifically embedded systems are becoming more powerful, software radios are more feasible solutions for real-world applications [17, 18, 19, 20, 21].

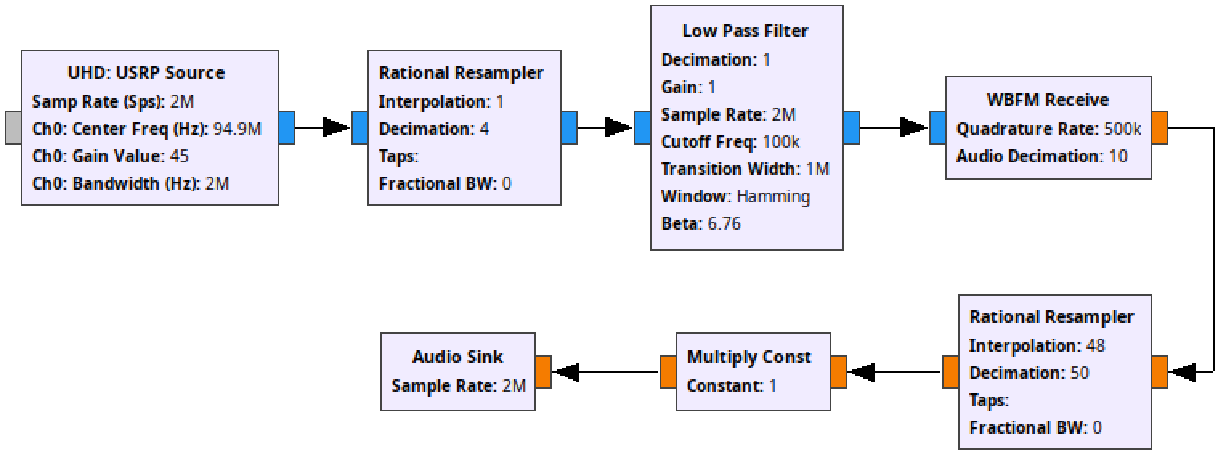


Figure 2.1: Example of a software defined FM receiver. This shows a GNU Radio flowgraph implementing a receiver for broadcast FM (Frequency Modulation) radio.

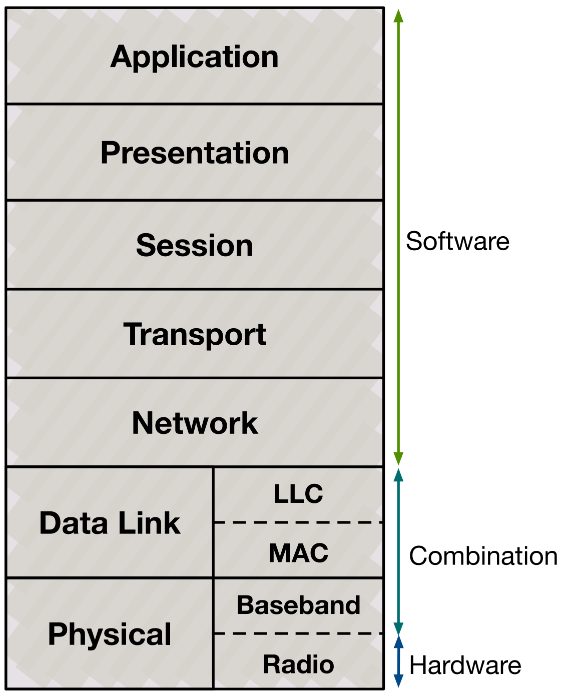


Figure 2.2: Example OSI network stack for software radios. Conventional communications systems implemented the lowest layers of this stack in hardware, whereas these lower layers are implemented as software in SDRs.

SDRs still consist of some hardware components; specifically, the radio hardware itself. SDRs can be divided into two main components: the hardware radio frequency (RF) frontend and the software waveform. SDR RF frontends typically consist of several components such as tuners and filters for initial signal conditioning, and the analog-to-digital and digital-to-analog converters. Software waveforms can be implemented in a variety of manners; some waveforms are completely implemented in reprogrammable logic devices, while others are implemented as traditional software on general purpose platforms. As the size, weight, and power (SWaP) of embedded and SDR hardware increases and the overall hardware cost decreases, software radios are becoming a more feasible solution for real-world applications. Until recently, SDRs have mostly existed in the research world due to high latency, low throughputs, and high cost in comparison to traditional hardware systems. As computing hardware has significantly increased in speed, efficiency, reduced power consumption, and physical footprints, software radios are becoming a more feasible solution for real-world applications that are traditionally filled by hardware radios.

Software defined radio allows for almost unlimited flexibility in developing wireless applications, which makes it useful for many different use cases. They are excellent prototyping tools for research and development and allow engineers to easily prototype and test new protocols, standards, and waveforms. Using SDRs in an industrial or commercial capacity allows for highly flexible, future-proof systems that can easily be upgraded once deployed to the field. One example of this is implementing low-cost cellular basestations using SDRs to support multiple different standards. Two examples of an SDR implementation for cellular standards are OpenBTS and srsLTE [22, 23, 24]. SDRs are also heavily used in the defense sector because of the ability to create agile systems that can be better adapted to battlefield conditions.

From a security perspective, it was difficult in the past to analyze, test, or attack wireless systems because the hardware required was either too expensive, too highly specialized or did not readily exist. SDRs and other low cost wireless hardware have significantly lowered this barrier for testing wireless systems and have become invaluable tools for wireless security professionals. This gives researchers an almost unlimited ability to develop waveforms for either analyzing, reverse engineering, or penetration testing wireless systems. Applications such as the Universal Radio Hacker [25] and scapy-radio [26] have made analyzing wireless protocols even easier.

2.1.1 Frameworks

While software radios can be implemented in many different ways, there are many existing frameworks that can simplify the process of building complete waveforms. Some of the more popular open source frameworks include GNU Radio, REDHAWK, and LiquidDSP [27, 28, 29, 30]. The GNU Radio and REDHAWK frameworks are somewhat similar in purpose; they both provide the underlying architecture for connecting multiple signal processing blocks together into a data pipeline as well as some basic signal functionality and visualization tools. On the other hand, other frameworks, like LiquidDSP, provide a much more comprehensive set of signal processing functions but do not provide the same data flow or scheduling functionality for building waveforms. The specific architecture used for an implementation is highly dependent on the processing requirements for the system. GNU Radio is primarily useful for applications running on a single system, whereas REDHAWK targets distributed environments. GQRX and ShinySDR are both examples of generic SDR applications that were developed on top of these frameworks; both utilize GNU Radio under the hood. [31, 32]

Both GNU Radio and REDHAWK share similarities in their purposes, but there are several major differences in the implementation and overall functionality of each framework. One of the major differences between the two frameworks is their fundamental computing model. GNU Radio is based on a thread-per-block model where an entire waveform executes as a single process on a single system. Waveforms can be interconnected using different networking blocks, but the primary computing model is the multi-threaded single process. At runtime, the scheduler creates shared memory buffers between blocks for streaming data through the application; a message-passing infrastructure is also provided for supporting packet and burst based applications.

Within the GNU Radio framework, there are two major ways of passing data or samples between processing blocks in a waveform: streams and messages. Streams provide blocks with a continuous stream of input samples or data, typically for the lifetime of the waveform. They are intended to be coupled to an input (source) or output (sink) block that is running at a fixed sample rate. Messages are part of GNU Radio's built-in message-passing framework and are objects (Polymorphic Types or PMTs) that can take any data type of any size and send it to any other block's message port. They are discrete packets of samples or data and can be sent to a block at any time during a waveform's lifetime. When working with a bursty waveform (such as Wi-Fi), this messaging framework provides an easy mechanism for handling received frames and passing packets throughout the waveform. A block in GNU Radio can assign specific callbacks to specific functions that act as handlers for incoming messages. When a message is sent to a block, the GNU Radio scheduler will call this handler and pass the message as an argument to the function.

On the other hand, REDHAWK was developed to comply with the Software Communications Architecture (SCA) specification and is an infrastructure designed for distributed real-time applications. Each processing component within a REDHAWK application executes as its own process on a system; communication between components is handled through the Common Architecture Request Broker Architecture (CORBA) middleware. Applications can be deployed to multiple types of targets including single Linux hosts or an entire network of servers. REDHAWK also provides the tools for developing and deploying applications onto a distributed architecture.

2.2 Emerging Technologies

Software defined radios are the key enabling technology for many different emerging technologies. The increasing number of wireless networks and connected devices is causing significant congestion and demand on available resources, so backend networks need to be adaptable, intelligent, and resilient [33]. Below, we briefly discuss some of these emerging technologies.

2.2.1 Cognitive Radios

Two of the major uses cases of software defined radios are cognitive radios and dynamic spectrum access [34, 35, 36]. A wireless system implemented mostly in software allows for systems that can learn about their surrounding environment and adapt their configurations to operate more efficiently. This concept is extremely broad and can be applied to many different use cases such as: avoiding interference, adapting error correction schemes, or changing the modulation scheme based on channel conditions. With a significant decrease in the amount of radio frequency (RF) spectrum available for use, additional research has focused on developing systems that can share already allocated frequency bands without affecting the primary users utilizing those bands. Software radios are the perfect tools for implementing these Dynamic Spectrum Access (DSA) systems due to their ability to easily adapt to their operational environment.

2.2.2 Cloud Radio Access Networks

In recent years, the number of mobile and connected devices has grown significantly. This has put huge demands on cellular networks. One of the recent concepts that has emerged in order to help handle the demand on network capacity is the Cloud Radio Access Network (C-RAN) [37]. Rather than using individual base-stations (BS) that can process data for users only within their geographical area, the C-RAN model virtualizes the base station and co-locates the systems in a centralized datacenter and uses an optical network to connect to remote radio heads (RRH). Since the base-stations are now virtualized, the data center is able to dynamically reconfigure and allocate resources based on the demand on the cellular network. This type of architecture also allows network providers to better utilize computing resources resulting in lower operating costs, which also sets the stage for a more cognitive cellular infrastructure that can more efficiently utilize available spectrum.

2.2.3 Satellite Communications

Another emerging trend making heavy use of software defined radio is the satellite communication industry [38]. With embedded and space-rated hardware becoming more powerful and cost effective, more satellite radio systems are being implemented this way.

In addition, SDRs are being heavily utilized as the main processing components for ground stations around the world. Rather than requiring a single ground station (or multiple) for each new satellite, SDRs allow a single ground station to communicate with many different satellites by simply changing the waveform in use. Software packages such as *gr-satellites* [39] are helping to build a common set of software functionality that can be tailored to fit the requirements for a specific mission. A prime example of this is the Virginia Tech Ground Station, which is a fully-software defined ground station designed to operate in multiple Amateur Radio bands [40].

2.2.4 Sensor Networks

Sensor networks are also a heavily growing area within wireless communications [41]. With the explosive growth of the Internet-of-Things, more and more sensors are constantly being connected to the Internet. SDRs may not directly be used to implement the radio in the sensor itself simply due to power constraints, but they are extremely useful for developing the centralized nodes receiving data. This allows master nodes to handle many different types of sensors and protocols without changing their hardware. This will help reduce overall costs for developing sensor nodes.

2.3 Principles of Security

In the security world, there are several different concepts and principles that are key to information security. Here, we briefly discuss some of those principles and models that are applicable to our research.

2.3.1 Confidentiality, Integrity, Availability

Within information security, the CIA triad (Confidentiality, Integrity, Availability) is the common method of characterizing the different security aspects of a system. The Common Vulnerability Scoring System (CVSS) [42] is an open standard that is used to score the impact of specific vulnerabilities based on their impact of each of the three CIA categories.

- **Confidentiality** - Information in the system can only be accessed by authorized personnel and cannot be accessed with unauthorized personnel.
- **Integrity** - Any information stored or handled by the system cannot be changed by unauthorized users.

- **Availability** - The system and any information stored should always be available for use by legitimate users.

In some cases, a fourth concept, *Non-Repudiation*, is added which deals with the authenticity of the data in the system. Essentially, this concept states that any actions taken by a user within a system were in-fact undertaken by that user.

2.3.2 Principle of Least Privilege

The principle of least privilege is a security concept for system design where each component within the system is given only the minimal privileges required in order to operate properly [43]. The goal is reducing the privileges for components in case a vulnerability exists within that component. If a vulnerability were exploited, then the attacker is limited in the amount of access to the system.

For a system implementing the principle of least privileges, the lowest level of the operating system (the kernel) is the only component to execute with full privileges in the system since it manages the system hardware. Userspace libraries and applications all execute with minimal privileges which can be enforced at the hardware layer.

Many modern operating systems implement the concept of privileges rings in hardware, where the innermost ring (ring 0) has the highest privileges. Outer successive rings have fewer privileges than the interior rings. The kernel is responsible for managing which processes execute in which rings. Typically, user applications are executed in the outermost ring with the least amount of privileges.

2.3.3 Multi-Layer Security/Defense-in-Depth

The concept of Multiple Independent Levels of Security/Safety (MILS) originates from a military mindset where there are several layers of defenses applied to a system to make compromising the system extremely difficult [44, 45]. The basic concept of this approach is developing systems in a distributed manner rather than as a monolithic system. Rather than executing each component on a physically separate system, a security kernel implements isolation between individual components so the entire system can execute on a single processor. Information flow is enforced in the system where components within the same security level or domain can communicate. One major goal of this isolation is to protect trusted data and applications from compromise if an exploit occurs. If a single layer of the system were compromised, the additional layers of defenses protect the core of the system and help mitigate the effects of an attack.

2.4 Vulnerabilities

There are many different types of vulnerabilities that can exist in software due to mistakes in either the overall design of a system or in the specific implementation [46]. Vulnerabilities typically arise because a developer improperly handled external user input to the system; improperly sanitized input can cause unexpected behavior in an application's execution. In this section, we briefly mention some common types of vulnerabilities that are found in software.

2.4.1 Buffer Overflows

Buffer overflows are historically one of the most common types of software vulnerabilities that have major implications for the security of the overall system [46]. In many low-level programming languages, correct memory management and access is a task left to the programmer, with very little or no protection against overflows being provided by the language. If a program is using a fixed size memory buffer and the programmer fails to correctly handle the memory operation, an overflow occurs when the program writes or reads memory outside of its bounds. This can result in unexpected execution behavior for the application or in information leakage. While there have been many defenses designed to protect against overflows, many embedded systems and software either do not have them properly configured or fail to implement them altogether. any SDR implementations are targeted at mobile and embedded devices, which could make them highly susceptible to this class of attack.

In low-level programming languages such as *C* and *C++*, memory management is the developer's responsibility and very little or no memory protection is provided by the language. Different types of vulnerabilities can occur if memory is improperly managed. Buffer overflows are historically one of the most common types of these software vulnerabilities and can have a major impact on the overall security of a system [46]. Overflows occur when a process attempts to access memory addresses that are outside the bounds of an allocated buffer. Most of the time this involves a process writing past the end of a buffer and causing corruption to adjacent memory which will likely result in unexpected behavior for the application. Overflows can also include reading memory past the end of a buffer; this can leak sensitive application information to an attacker. The Heartbleed bug (CVE-2014-0160) in the OpenSSL library is an example of a buffer over-read vulnerability that could be used to leak sensitive information such as passwords or keys to attackers [47].

A buffer overflow exploit can result in different behavior depending on the buffer's location in memory; this could include a crash, unauthorized system access, or remote code execution. A process's memory space is divided into several different segments such as the data, code, stack, and heap segments. The code segment contains an application's executable code and the data segment contains any statically or globally declared variables. Objects that have long lifetimes and are shared in multiple areas of the application are typically placed in the heap. The stack is mainly used to track the control flow of the application, while the heap segment is used for dynamically allocated objects. Each time a function is called, a new stack frame (Figure 2.3) is created that contains values such as a return memory address, any arguments passed to the function, and any variables declared within the function's scope. If a buffer with an overflow vulnerability exists on the stack, this can be used to modify the execution of the application by overwriting the return address with a malicious address. An attacker can overwrite important addresses within the stack frame and hijack the application's execution flow.

When a buffer is allocated, the programmer must ensure that all memory operations are within the bounds of the buffer. An overflow occurs when a process attempts to write to memory addresses that are outside of the allocated buffer. This corrupts memory and can cause unexpected behavior for the application, such as a crash (segmentation fault), control flow corruption, unauthorized access, or even arbitrary remote-code-execution (RCE).

Clever exploits of an overflow vulnerability allow an attacker to overwrite variables stored in memory adjacent to the original buffer and cause unexpected behavior of the application.

Stack

A process's memory space is divided into multiple different sections: heap, stack, code, and data sections. The code section contains all of the executable code for the application and the data section typically stores global variables. Both the stack and heap are used to store variables declared elsewhere within the application.

In addition, the stack handles all of the control flow information for the application and stores variables declared locally to a function. An example layout for the stack is shown in Figure 2.3. For each new call to a function, a new frame is created on the stack which includes the return address to the previous code and any arguments that were passed to the function. A stack overflow can occur when too many stack frames are created for the currently executing application, exhausting all the memory that was allocated for the stack. In this situation, the application will crash because there is no memory available to continue execution. This can sometimes be seen with improperly configured recursive functions that create an infinite calling loop. As each successive function is called, a new frame is created until all the stack memory is exhausted and the application is aborted.

2.4.2 Shellcode Injection

Shellcode injection is a method for exploiting vulnerabilities in order to control the execution flow of an application and possibly gain unauthorized access to a system. In this situation, an attacker generates a payload of binary, executable code that can be injected into an exploit. The attacker's goal is to use an exploit to trigger the targeted application to then

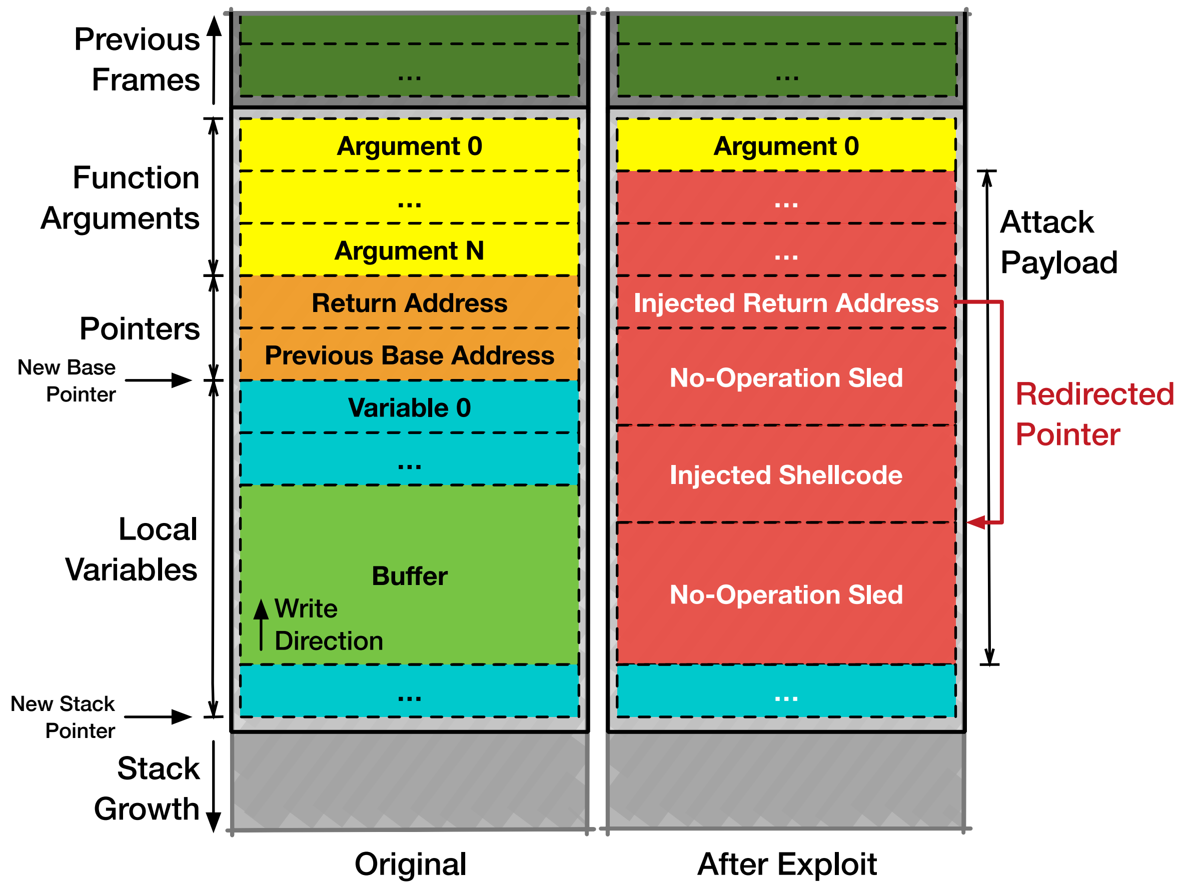


Figure 2.3: Diagram of a stack frame and overflow exploit. This shows an example of a stack frame (left) and the result of a stack buffer overflow exploit (right). For each function call, a new frame is created that contains a return address, any arguments passed to the function, and any local variables. A buffer overflow vulnerability can be used to overwrite the original return address in the stack frame to point to injected shellcode. If successful, the system will jump to the injected code rather than the original caller.

execute the injected code rather than it's original code. A shellcode injection vulnerability is key to allowing an attacker to compromise remote systems. Through this, an attacker can create a payload that opens backdoor access into the system and allows then to gain greater control over the targeted system.

2.4.3 Defenses

Since buffer overflows are a common type of vulnerability and shellcode injection attacks often use buffer overflow vulnerabilities under the hood, there have already been several major types of defenses developed to mitigate the risks of these vulnerabilities [48, 49, 50]. The main defenses include non-executable stacks [48], address space layout randomization [51], and stack canaries [50]. Modern operating systems already employ these defenses to prevent buffer overflow exploits which makes it very difficult to find vulnerabilities that allow remote code execution on these systems. Even with these defenses, methods do exist to bypass these defenses and still exploit a vulnerability. Also, many embedded systems do not implement or properly configure these protections and are vulnerable to attack. The GSM and Wi-Fi baseband exploits discussed in Section 3.2 are perfect examples of embedded implementations without these defenses.

Data Execution Prevention

This defense is designed to prevent shellcode from being executed by the system. Essentially, the stack memory segment is marked as non-executable by the system's memory protection unit which prevents an attacker from jumping to shellcode they injected on the stack. However, there have been several techniques that have been developed in order to bypass this defense: specifically, return oriented and jump oriented programming. With both of these methods, attackers construct payloads that use existing code chucks loaded into memory (like standard library functions) rather than providing their own shellcode.

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a method of randomizing the location of where different memory segments are loaded within a process's virtual memory space. By randomizing the location of segments each time an application is executed, it makes it extremely difficult for an attacker to create exploits that rely on shellcode injection or returning to specific locations in memory. However, ASLR can be bypassed by leaking memory locations through other vulnerabilities.

Stack Canaries

Stack canaries are a defense against buffer overflows in variables located on the application's stack segment. Essentially, a compiler adds a known value to memory adjacent to a buffer located on the stack which is checked once the current function has completed. If the value has been modified from the known value the compiler saved, then the application assumes the stack has been corrupted and aborts execution.

2.4.4 Integer Overflows

Data types within an application, like unsigned integers, can hold a value within a specific range depending on the total number of bits used to define that variable. An integer overflow occurs when a program increments a variable past its maximum value. In this situation, the hardware attempts to increment the value, the resulting value is outside of the range for that data type and the most significant bits of the result are dropped.

If a variable (such as a counter) is constantly being incremented, the result of an overflow depends specifically on the data type in use. For example: if the variable is an unsigned integer, the overflow causes that variable to be reset to 0. If the variable is a signed integer, exceeding the maximum positive value of the variable causes it to become negative and begin incrementing in the negative direction. Many such variables are used for accessing offsets within a buffer, which can lead to unexpected behavior if the integer overflows.

2.4.5 Off-By-One

An off-by-one vulnerability is typically a special case of the buffer overflow. In this situation, the application is incorrectly handling the length of a buffer due to an incorrect evaluation statement. Many programming languages use a 0-based indexing scheme which means that the first element of an array is element 0 and not element 1. Given an array of length N, if a programmer attempts to access element N, they would actually be accessing the next few bytes of memory (depending on an item's size) after the buffer rather than the last element of the buffer. This can allow attackers to modify data outside of the buffer if such a vulnerability exists.

2.4.6 Privilege Escalation

A privilege escalation vulnerability is simply one that allows an attacker to use the vulnerability to gain additional privileges for a specific application or process that it would normally not have. Since many systems are built on the principle of least privilege, this type of vulnerability allows an attacker to have greater access into the system. Typically, attackers target these types of vulnerabilities in the kernel of an operating system. The goal is exploiting the vulnerability in order to gain administrators access to the rest of the system.

When exploiting remote systems, attackers many times will use different combinations of these types of attacks. For example, once they have gained access to a system through a shellcode injection attack, they can then use a privilege escalation attack against a kernel vulnerability which would give them complete access to the targeted system.

2.5 Virtualization

Over the past two decades, virtualization has become an enabling technology for many of the systems we use every day. Virtualization was first introduced mainly as a methodology for improved resource management within server infrastructures. Rather than use a single server to execute a single application and waste valuable computing resources, system administrators could virtualize their available hardware and allocate only the required resources for an application on a specific system. This in turn allowed for multiple applications to execute independently on a virtualized host and gave administrators the ability to allocate extra resources or start more instances of an application based on user traffic. Here, we briefly describe two common virtualization techniques: hardware (or platform) virtualization and operating system virtualization (or containers). Figure 2.4 shows a comparison between these two types of virtualization technologies.

2.5.1 Hardware Virtualization

Hardware virtualization refers to the concept of creating guest virtual hardware in software that can be used to emulate a hardware system [52]. This allows for an entirely separate operating system and environment to be installed on the guest. Resources from the host system can be allocated as needed to the guest system.

A hypervisor is software that provides the ability to create this guest machine and manages how resources are allocated to the system. Two different types of hypervisors exist (Type 1 and Type 2) depending on how the system is virtualized.Type 1 hypervisors are mainly targeted toward data-center infrastructures and implement a small system that provides basic functionality such as guest management, resource management, and scheduling. This type of hypervisor acts similar to a kernel for a normal operating system, and is intended to run in a headless environment and be managed over the network.

On the other hand, Type 2 hypervisors execute as a child process within a host operating system and provide the functionality for guest virtual machines to be created on the running host. This type of hypervisor is typically useful for development scenarios. Developers can create guest machines and install a guest operating system and software independently of the host.

2.5.2 Operating System Virtualization

Another virtualization technique that has recently become more popular is operating system virtualization [53]. Rather than completely create a new guest virtual machine to execute a small application or micro-service, operating system virtualization relies on the ability to create containers or jails within the kernel. These containers provide isolated environments

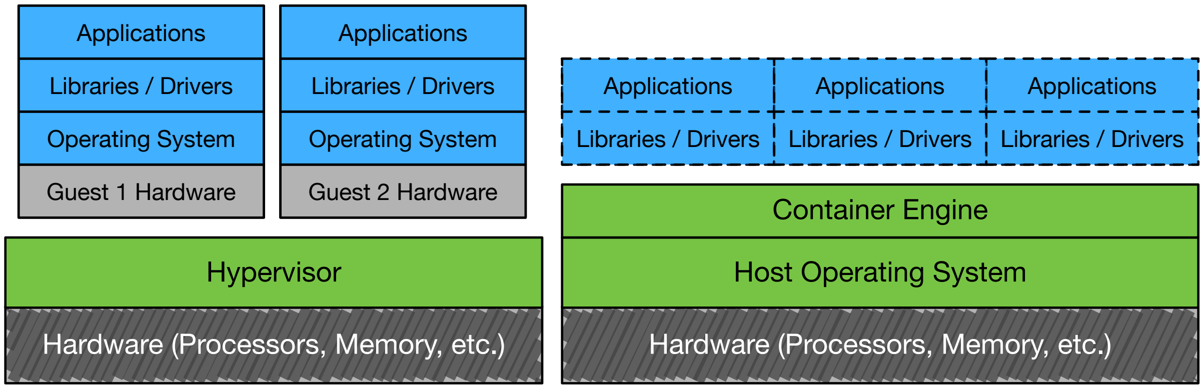


Figure 2.4: Comparison of virtual machines and containers. An example software stack for virtual machines is shown on the left and a container software stack is shown on the right. Virtual machine monitors (hypervisors) create entire software interfaces that mimic physical hardware and run an entire guest operating system. Containers share a common host kernel, but use namespaces to create different execution environments for processes. Overall, containers provide a more lightweight virtualization solution compared to virtual machines, but virtual machines provide more isolation from the host system.

for applications very similar to guest virtual machines but with significantly less performance overhead. Each container can have a completely separate set of resources which allows administrators to specifically tailor that environment for the intended application without affecting any other container. This technique makes it extremely easy to deploy applications to cloud computing environments.

Chapter 3

Related Work

As previously mentioned, wireless systems are rapidly evolving and becoming more complex and numerous. However, the security of these systems is still lagging behind due to this rapid evolution and it can be completely overlooked or is simply not a very high priority. The open nature of wireless makes security especially challenging because there are no physical connections with these systems that can be secured. Traditional security threats inherent to wireless systems can be generally categorized based on an attacker's capabilities. This includes 1) passive attacks where adversaries are simply observing transmitted signals (eavesdropping), 2) active attacks where adversaries attempt to disrupt communications by transmitting energy (jamming), and 3) active attacks against higher layer protocols to compromise the confidentiality and integrity of the signal [54].

Software defined implementations add a new fourth class of threat: active attacks against the implementation where adversaries use traditional cyber-security techniques to exploit vulnerabilities in a system and disrupt communications or gain unauthorized access to the system. Interestingly, SDRs can be both susceptible to this threat, as well as, part of the threat itself. They can be programmed to transmit almost any imaginable signal, and this flexibility makes them perfect research and penetration testing tools for evaluating the security of other systems. Since SDRs are software, they can also be exploited by the same type of attack, which is the focus of the work presented in this dissertation.

The chapter presents related work concerning the security of wireless systems. Most of the research in this area is focused specifically on the security of different wireless protocols that are supported by systems rather than the security of the implementations themselves. We briefly review existing work that provides a taxonomy of attacks against all wireless systems and generally categorizes threats against SDRs in Section 3.1. Next, Section 3.2 presents recent examples of exploits against firmware in consumer wireless devices where researchers targeted and exploited vulnerabilities in the software implementations of the protocol stack of different standards. In Section 3.3, we also examine existing models and methods for developing secure SDR applications; however, most of this work again only considers securing the protocol or update processes of software radio systems. The Secure Radio Middleware (SRM) model reviewed in this chapter is one of the few approaches that considers security from the perspective of defending from attacks against the vulnerabilities in implementation. Finally, Section 3.4 compares the different proposed models for SDR security and discusses the limitations of these methods and what attacks they are unable to prevent.

3.1 Security Issues in Software Radio

Software defined radio has also become an invaluable tool in the security world because its flexibility gives researchers the ability to analyze, test, or reverse engineer almost any wireless system or protocol. Applications such as the Universal Radio Hacker [25] and scapy-radio [26] have made this even easier. However, research into the security of the software radio implementations themselves has been less widespread. Past research has been primarily focused on protocols and features such as authentication and encryption and not necessarily the radio implementation itself. There are many different security threats and issues that arise with software radios. Some are inherent to any wireless communication systems, and others are only applicable to SDRs because of their software nature and design philosophy. SDRs are designed to be very flexible, so ensuring that the software and configuration of these radios are secure is of major importance.

In [55], Baldini et al. provide a comprehensive survey of the different security threats that can exist against software radios and cognitive radio networks. They identify fifteen different threats to software radios, including denial-of-service attacks, compromise of user and configuration data, and vulnerabilities in the software framework of the radio. A summary of these threats and their overall effect on the system when exploited is shown in Table 3.1. These threats can be generically classified using the traditional CIA model of security objectives: confidentiality, integrity, and availability. A report on telecommunications network security published by the ITU [56] slightly expands on this model by making a distinction between data integrity and system integrity:

- Confidentiality - Access to data transmitted or stored by the system must be controlled
- Data integrity - Data transmitted or stored by the system must be protected from unauthorized modification
- System Integrity - The system's underlying operating system, services, and libraries must be protected
- Availability - Users and providers should not experience interruption of service

Baldini et al. further note there are specific functions and assets of software radios that can be affected by security threats. Assets within a SDR consist of user data, configuration data, and the software waveforms executing on the system. From a functionality perspective, SDRs provide the ability to execute different applications and waveforms on generic computing hardware. This requires that a secure SDR architecture provide secure methods for application and resource management; this involves ensuring the security and integrity of downloaded waveforms as well as preventing computing resources from being maliciously consumed or mis-managed. Another major concern is ensuring secure management of user data as it is stored and/or transmitted by the SDR. Finally, SDRs are inherently data-flow applications, so the transport mechanism between SDR components should be secure.

Based on these generic security objectives and SDR assets and functionality, the authors have identified fifteen different threats to a software radio. For the scope of this report, we are mainly focused on the threats against the software implementation of the radio itself. Threats that are primarily against the integrity of the system include: inserting malicious software or waveforms and altering the SDR's configuration, waveform, operating system, and/or software frameworks. Availability threats that could result in denial-of-service include unauthorized consumption of resources and software/hardware failure. A complicating factor is that some of these threats can impact more than one of the security objectives in the CIA model.

Table 3.1: Threats to SDR systems. This table gives a brief list of threats inherent to software defined radio systems [55]. The final column shows the primary effect (in terms of confidentiality, integrity, or availability) an attack would have on a system.

SDR Threat	**Effect C/I/A**
Malicious software/waveforms	All
Altered configuration data or user data	Integrity/Availability
Altered waveforms, frameworks, operating system	Integrity/Availability
Software/Hardware failure	Availability
Extraction of configuration/waveform/user data	Confidentiality
Excessive resource consumption	Availability
Masquerading as authorized software waveform	All
Unauthorized use of SDR services	Confidentiality

In [57], the authors conducted a threat analysis of the GNU Radio framework and presented different threats against the platform and requirements for building secure radios. They listed several threats to ensuring a secure download and execution environment for new waveforms. Many of the threats they present have been addressed elsewhere, but one interesting threat they address is the fact that an entire GNU Radio waveform executes in the same address space. This would allow any compromised or malicious radio module to access memory from another module.

They also present the possibility that buffer overflows could occur on the memory buffers shared between blocks and be exploited to modify the behavior of a waveform. A buffer overflow within a specific block in GNU Radio could allow an attacker to remotely modify the behavior of a waveform through an exploit, which we presented an example of in the previous section. A couple of solutions were proposed to help defend against these attacks, such as moving away from the thread-per-block model of GNU Radio and instead using multiple processes like REDHAWK. The authors developed a new shared buffer using a Linux privileged memory system call that provides better and more secure management of connections within a flowgraph; specific blocks in the flowgraph could only read or write from the buffer if they were properly authorized.

3.2 Wireless Firmware Exploits

In many different wireless systems on the market today, a significant portion of the overall functionality is implemented in firmware running on separate co-processors rather than hardware. This firmware typically executes on a separate co-processor in the system to reduce the load on the main application processor and improve power consumption.This allows the host's main processor to offload most of the network functionality, usually consisting of the higher-level network functionality, to the wireless component itself. Offloading work from the main processor frees resources, as well as, providing a more power efficient implementation, since these co-processors typically include a low-power micro-controller core that handles the functionality. For example, in Linux terminology, Wi-Fi modules can be categorized as either SoftMAC or HardMAC devices depending on whether the management and data layer functionality is handled by the main host or offloaded to the wireless module. In either case, the management and data layers of the network stack are all implemented in software (or firmware on the module) rather than actual hardware.

In addition, the co-processor hardware and firmware do not necessarily implement the same defense mechanisms against exploits that commonly exist in modern desktop, server, and mobile operating systems. This could be a result of limitations with the co-processor's hardware or an improperly configured system that fails to take advantage of available security features like memory protection units. Any vulnerabilities within the firmware could be exploited by an attacker to compromise the firmware and possibly even compromise the host device as well.

In this section, we discuss several examples of attacks against the wireless firmware of a system. While these wireless chips are not full software radios, the same security principles issues can apply because of their software implementation. Since much of the network functionality is implemented as firmware that may have exploitable vulnerabilities creates new attack vectors against the firmware. There are several examples of this type of attack against the firmware of a co-processor that, once compromised, can be used to exploit the host systems and completely take over control of the device.

3.2.1 Cellular (GSM) Baseband Exploit

One of the first examples of this type of attack was shown by Weinmann [58] and demonstrated a wireless exploit of a GSM baseband implementation. The authors targeted the different management layers of GSM (Layer 3), which is divided into several different sublayers including: Radio Resource Management, Mobility Management, and Connection Management. After completing a vulnerability analysis of the firmware, they discovered several different types of bugs in these layers, mainly including buffer overflows, integer overflows, and memory leaks.

Most of these exploitable vulnerabilities existed due to insufficient length checks for fields within different GSM management frames. For example, during registration a GSM base-station should assign a 32 bit long Temporary Mobile Subscriber Identity (TMSI) to an unknown device. However, this length was not checked in the baseband firmware, which resulted in software failures if a malicious base station transmitted a longer TMSI. This management frame used a variable length field that allowed an attacker to send a much larger value to a device, which results in an overflow that crashes the firmware. The authors constructed exploits using the discovered vulnerabilities and were able to remotely execute arbitrary code on a device. They demonstrated exploits against several different targets including an HTC Dream and an Apple iPhone 4.

3.2.2 Broadcom Wi-Fi Exploit

In 2017, researchers from both Google's Project Zero [59] and Exodus Intelligence [60] released similar exploits of the firmware on Broadcom Wi-Fi modules. The ultimate goal of their research was wirelessly exploiting the Wi-Fi firmware on one of these modules and using the compromised module as an attack vector to exploit the kernel of a mobile device. One issue they discovered was the co-processor's Memory Protection Unit (MPU) was incorrectly configured and allowed all memory to be executable. This greatly simplified the ability to attack the firmware.

Both exploits used buffer overflows and different memory management techniques to target important objects in the firmware's memory. By transmitting specially crafted Wi-Fi frames, the researchers were able to successfully use heap construction techniques and the stack overflows to modify important sections of the firmware's memory.

Since the Wi-Fi standard requires certain events to be periodically triggered, several timer objects were stored in memory. If a timer could be overwritten and its function pointer given a new memory address, an attacker could inject shellcode (binary, executable code) and modify the timer to execute a malicious handler function rather than the original when the timer elapsed. Once the timer event was triggered, the firmware would jump to the injected shellcode which could be used to further compromise the Wi-Fi module and eventually the host processor on the mobile device. By using heap construction techniques and overflows the researchers demonstrated this ability to overwrite a timer object with an address to injected shellcode. The researchers were able to demonstrate this exploit and eventually use the compromised Wi-Fi firmware as a vector to exploit and compromise the mobile device.

3.2.3 BlueBourne Exploit

Bluebourne is another interesting example of a similar attack. In 2017, researchers from Armis disclosed multiple overflow vulnerabilities in different implementations of the Bluetooth network stack in both the Linux (BlueZ) and Android (Bluedroid/Fluoride) operating systems [61]. A major difference between BlueBourne and the GSM and Wi-Fi exploits is that the Bluetooth network stack executed on the main system processor rather than a co-processor. This meant that the exploitation of an overflow was not quite as straightforward as the other examples. Defenses against overflows, such as Address Space Layout Randomization (ASLR), required the researchers to use additional techniques and vulnerabilities to leak important values in memory before they could successfully exploit the firmware and bypass ASLR.

3.3 SDR Security Models

There have been several past architectures proposed in existing research that address approaches for security SDRs against these threats. Software radios are designed to be upgradeable and execute different waveforms depending on the desired configuration, so the majority of these architectures have focused on securing the download process, radio configuration, and enforcing proper operation based on a device's specifications and its associated, regulated security policies. SDRs can be reprogrammed to operate in almost any wireless band (that is supported by the front-end), so they must be designed to block threats that cause them to operate outside their regulatory specifications. Different frameworks for securing the download process of new waveforms and configurations to a SDR platform are described in [62, 63, 64]. Since SDRs can simply be reprogrammed to operate in different configurations and radio bands (assuming the physical hardware itself is capable of operation in the selected band), ensuring the radio is operating with an authorized configuration is critical [65]. A denial-of-service for the system could occur if a radio's physical layer parameters (modulation, power, bandwidth, and/or frequency) were corrupted or maliciously modified [57]. A radio transmitting at an incorrect frequency, bandwidth, or power could interfere and jam other users or complete communication systems if it is operating outside of its authorized frequency band. Also, building intelligent, cognitive radios that can more efficiently utilize available resources through dynamic spectrum access is a major application for SDR, so there has been a significant amount of research into security threats against cognitive radio networks [66, 67, 68, 69].

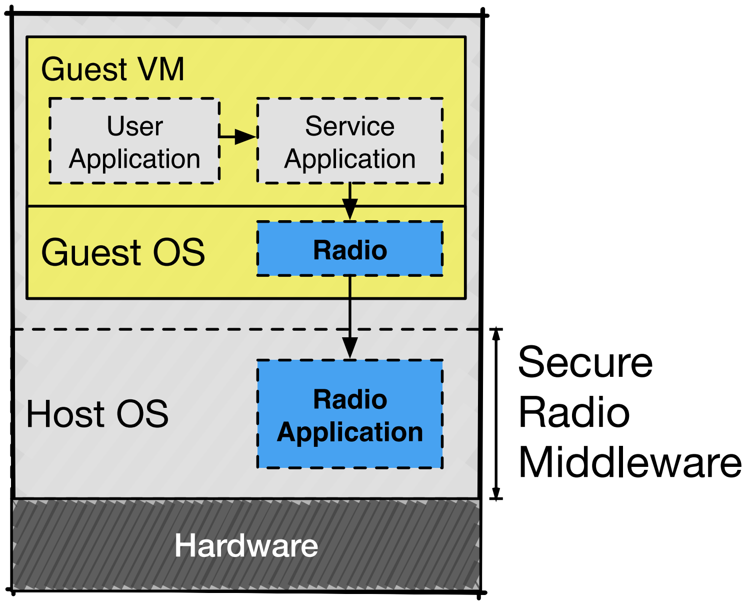


Figure 3.1: Secure Radio Middleware security model - Here, the user application layer is contained within a virtual machine and the radio layer exists as part of the underlying hypervisor.

The concept of using isolation mechanisms within software radios is not particularly new. Some have briefly mentioned using this technique, but do not provide good models or examples of how such an architecture would be implemented [70].

In [71], the authors present a new architecture for SDRs designed to protect the radio from exploitation and unauthorized reconfiguration. They show multiple examples of how an attacker could compromise an un-secure application in a user environment and use this to compromise the underlying operating system. To protect an SDR from modification, they propose a new model for secure software radios that uses a virtualization based layer that isolates the user environment from the radio application and the host system. This 'Secure Radio Middleware' creates an isolated environment where the user applications execute; any exploitation in this environment would be contained and unable to maliciously reconfigure the underlying hardware. An example is shown in Figure 3.1. The middleware layer also includes a security monitor that checks every outgoing message against an allowable policy and blocks any attempts to incorrectly reconfigure the radio.

In [72], the authors present the *High Assurance Wireless Computing System*, which is a specific implementation of a wireless system designed to protect against driver exploits in a system. This implementation is based on a multi-layer system architecture [45] and uses a similar approach to isolating the user application and the radio layer as Li et al. A separation microkernel implements different memory domains that are used to isolate components the user application from the device drivers and radio layer and the SCA framework is used for communication between the different domains. A firewall monitors traffic flow both to and from the isolated user environment and can block malicious traffic flowing either direction. Since the user environment sits in its own domain, an attack against that layer cannot escape and infect the radio layer. However, an exploit against the radio module can still be used to disable the firewall and infect another layer.

3.4 Limitations of Existing Approaches

The major limitation to the existing models described above is they do not take into account the possibility of the radio layer itself being compromised. While some of these models successfully protect radios from exploits from the user application layer, they do not protect the radio runtime itself from being exploited. As we previously showed, vulnerabilities can exist in the radio firmware; any exploits in the radio layer itself can be used to fully compromise the rest of the system. Because of this, secure SDR architectures must take into consideration the possibility that the radio layer itself could be compromised and provide mitigations against this attack vector.

In Table 3.2, we present a brief overview of the different levels of protection provided by each of the existing approaches. Our proposed Defense-in-Depth architecture presented in Chapter 5 provides the additional benefit of securing the radio itself against exploits. Other approaches either focus primarily on secure downloading of waveforms and policy enforcement, rather than the radio implementation itself. Our model for a secure, defense-in-depth architecture discussed in the next section takes this into account and builds upon these previous architectures to provide an isolation layer for multiple components in the system. Our approach can both secure the radio layer from exploit as well as provide monitoring and policy enforcement provided by other approaches.

3.5 Summary

Wireless systems are constantly evolving and growing more complex, and the open nature of wireless posses unique security threats to wireless systems that are not found in traditional IT systems. Systems are increasingly being implemented in software rather than application specific hardware in order to handle complex, modern wireless standards. But these software based systems are introducing a new class of threats where attackers can exploit vulnerabilities in the implementations.

In this chapter, we examined some of the existing work on SDR security which includes surveys of the different threats against SDRs, examples of recent exploits of vulnerabilities in commercial consumer device firmware, and related models for building secure software

Approach	Download	User App	Radio	Policy	Auditing
Michael et al.	P	N	N	N	N
Brawerman et al.	P	N	N	N	N
Uchikawa et al.	P	N	N	N	N
Sakaguchi et al.	P	N	N	P	P
Secure Radio Middleware	P	P	N	P	P
HAWCS	N	P	N	N	N
Defense-in-Depth (Proposed)	O	P	P	P	O

Table 3.2: Comparison of SDR security architectures. This compares the protection provided by different secure radio approaches: P - Protected, N - Not Protected, O - Optional. Most of these approaches focus on securing the software update process or protecting from exploits in the application layer.

defined radios. However, much of this existing work is focused either on the security of the SDR upgrade processes, the security of different SDR based protocols, or using SDRs as a tool for testing the security of other protocols. Very few related works have focused on protecting SDR implementations from exploits of vulnerabilities in the SDR themselves. The closest examples are the *Secure Radio Middleware* and the *HAWCS* systems, which both use isolation techniques to protect a software radio from exploit. The SRM system focuses on isolating the Application layer of the radio from the underlying radio firmware, so an exploited application cannot maliciously reconfigure the radio system. The HAWCS system uses isolation in network device drivers on SDR systems to protect the SDR from network side attacks. However, neither of these approaches consider attacks against vulnerabilities in the SDR itself, which is the major limitation of related models. The main focus of the work presented in this dissertation is this new class of attack where vulnerabilities can be exploited within an SDR implementation itself. Chapter 4 presented examples of these types of vulnerabilities, and Chapter 5 presents our Defense-in-Depth security model for mitigating these types of exploits in SDRs.

Chapter 4

Exploiting Software Defined Radios

Much of the existing work in wireless security has been focused primarily on vulnerabilities in wireless standards. Specifically with respect to SDRs, there is some additional work focused on models for securely updating systems and for ensuring proper reconfiguration of the underlying radio hardware. Very limited work has considered how vulnerabilities existing within implementations could be exploited; some recent research (presented previously) has demonstrated exploits in consumer wireless hardware. Like any complex software application, SDRs are not immune to these types of vulnerabilities, and this threat must be considered when developing future SDR applications and frameworks.

In this chapter, we examine and demonstrate this threat, which serves as the main motivation for our Defense-in-Depth SDR architecture proposed in the following chapter. We analyze several different examples of vulnerabilities in an implementation and demonstrate over-the-air exploits that target those vulnerabilities from a wireless attack vector. The example vulnerabilities presented here are common types of weaknesses that are found in other complex software systems. Our example demonstrations show how an attacker can exploit these weaknesses to cause significant performance degradation for the target, like a persistent denial of service against the system. These examples demonstrate the need for secure SDR architectures.

It is important to note that there are many existing defenses for the common vulnerabilities we demonstrate in this chapter. However, many of these defenses are intended for desktop and server operating systems. As we showed in the previous chapters, embedded SDR hardware does not necessarily support these defenses or the defenses are not properly configured or implemented in software. The examples we demonstrate in this chapter are implemented on desktop systems for easy of demonstration but are intended to mimic how exploits against embedded SDRs might occur. To that end, defenses like ASLR, NX bit, and stack canaries have intentionally been disabled to mimic vulnerable embedded systems.

Section 4.1 demonstrates how an attacker can manipulate the control flow of an SDR by transmitting malicious headers which trigger unexpected transitions within the system's state machine. In Section 4.2, we demonstrate how failure to properly sanitize control inputs within a control message can lead to inadvertently misconfiguring the SDR itself. Sections 4.3 and 4.4 present exploits of example buffer overflow vulnerabilities in heap memory and stack memory respectively [73, 74].

4.1 Control Flow Manipulation

The first example we present is a control flow manipulation attack that allows an attacker to modify the behavior of the system and degrade the overall system performance. In this example, a state machine controlling the system's behavior does not correctly account for all possible situations that may occur. Specifically, it is missing state transitions that would allow the system to recover from an unexpected situation. This allows an attacker to trigger a series of events that will cause a receiver to unintentionally drop legitimately transmitted frames.

The specific vulnerability shown in this example exists in the LiquidDSP framework and one of the included framing modules: the FlexFrame module. This module includes a frame synchronizer object which is responsible for detecting and decoding transmitted frames. The synchronizer operation is defined by a state machine consisting of four main states: frame detection, preamble synchronization, header demodulation, and payload demodulation. The implementation of the state machine in the *FlexFrame* synchronizer is shown in Figure 4.1 and is a very commonly used method for implementing bursty communication systems.

```
...
switch (_q->state) {
case FLEXFRAMESYNC_STATE_DETECTFRAME:
    // detect frame (look for p/n sequence)
    flexframesync_execute_seekpn(_q, _x[i]);
    break;
case FLEXFRAMESYNC_STATE_RXPREAMBLE:
    // receive p/n sequence symbols
    flexframesync_execute_rxpreamble(_q, _x[i]);
    break;
case FLEXFRAMESYNC_STATE_RXHEADER:
    // receive header symbols
    flexframesync_execute_rxheader(_q, _x[i]);
    break;
case FLEXFRAMESYNC_STATE_RXPAYLOAD:
    // receive payload symbols
    flexframesync_execute_rxpayload(_q, _x[i]);
  break;
default:
    fprintf(stderr,"error: flexframesync_exeucte(), unknown/unsupported state\n");
    exit(1);
}
```

Figure 4.1: FlexFrame state machine implementation. This shows an abridged version of the implemented state machine in the *flexframesync* object in LiquidDSP [30].

The frame detection and preamble synchronization stages allow the receiver to synchronize to the transmitted frame and correct for any timing and frequency offsets. Once synchronized, the receiver can demodulate the header and the included control information and configures the next stage to demodulate the payload using the selected modulation and error-correction schemes for that frame. The *FlexFrame* module allows for different modulation and error-correction schemes for each frame. Finally, the payload is demodulated and verified using the transmitted checksum. If the checksum is correct, the resulting payload can be delivered to the higher layers of the waveform; otherwise the payload is dropped.

As each state is completed, an internal variable is updated which triggers the next state once the loop restarts. If the receiver cannot synchronize with the preamble or the header is invalid or incorrectly demodulated, then the header demodulation state can abort processing the frame and reset the receiver to the detection state and wait for the next frame. Other than this, each state simply transitions to the next state during normal operation; assuming the header was correctly demodulated, then the receiver begins to demodulate the payload. The state transition diagram for the synchronizer is shown in Figure 4.2.

However, this is not all of the possible state transitions that should exist for this implementation. The issue is that the payload demodulation state can only transition back to the frame detection state once an entire frame has been received. It is unable to detect and handle a situation where only the header was transmitted with no corresponding payload. Once a header is correctly demodulated, the synchronizer will continue to demodulate the frame until the entire payload is received, even if it does not actually exist. Only then will the synchronizer continue to search for the next frame.

Since there is no check in place to confirm that a payload is actually being transmitted, attackers can take advantage of this to significantly degrade the performance of the waveform. Essentially, by transmitting a header with no payload, the attacker can trick the receiver to switch to the payload demodulation stage and miss any legitimate frames that were transmitted during that time.

This issue is compounded because the *FlexFrame* receiver can be configured differently for individual frames. Each frame can use different modulation schemes, inner and outer forward-error-correction (FEC) methods, and different lengths for the payload. Attackers can configure malicious headers to use the maximum length and the modulation and FEC settings that result in the longest possible payload. Assuming that the malicious headers are correctly formatted and demodulated by the receiver, this will block the receiver for the maximum possible time.

Once an attacker transmits a malicious header, the receiver would process any legitimate frame transmitted during the payload demodulation stage as part of the non-existent payload of the malicious header. The receiver is basically demodulating noise at that point, so the checksum it would compute for that payload would ultimately fail, causing the legitimate frame to be lost when the “payload” is dropped. If an attacker can correctly time the malicious header and periodically transmits them, they could easily cause significant performance degradation for the system. This attack is shown in Figure 4.3.

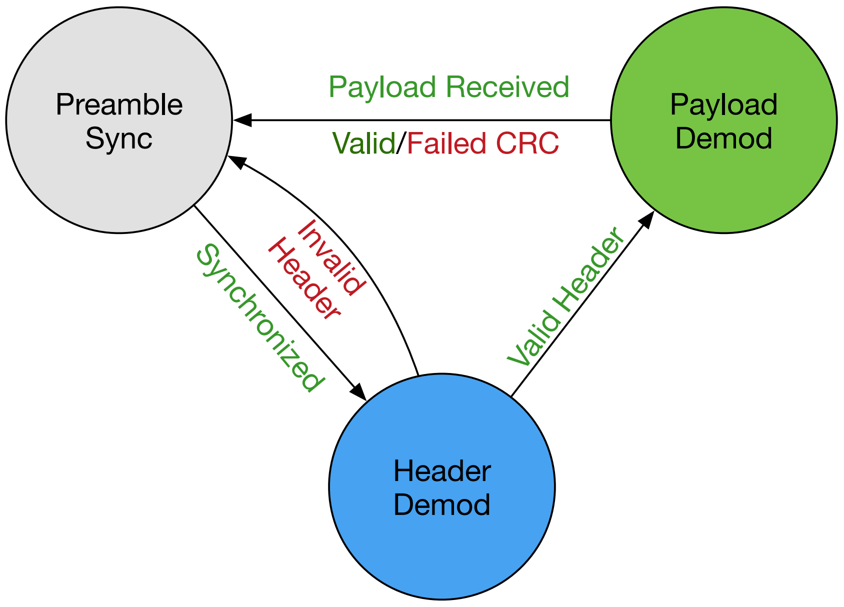


Figure 4.2: FlexFrame state machine diagram. This shows the state space of a vulnerable receiver implementation.

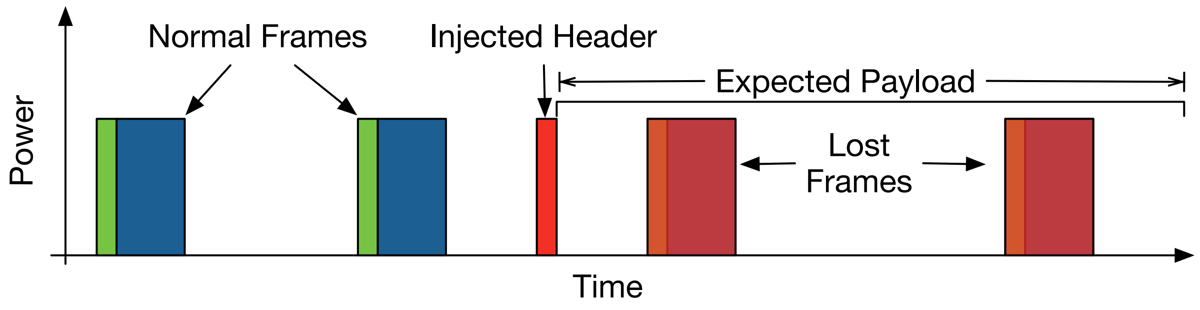


Figure 4.3: Example timeline of a DoS attack against a control flow vulnerability. If an attacker can successfully inject malicious headers specifying very large payloads at the proper time, the receiver could inadvertently drop legitimate frames when processing a non-existent payload.

Attack steps:

1. The attacker transmits a valid preamble and header but no payload. The malicious header specifies longest possible payload length, as well as, using a configuration of

modulation and FEC that results in the longest possible payload.

2. The receiver demodulates the malicious header and attempts to receive the non-existent payload. Legitimate frames transmitted at this point are treated as part of this payload.

3. The receiver attempts to calculate the checksum and validate this against the transmitted checksum, which is likely just noise.

4. The checksum ultimately fails and the receiver drops the failed payload along with any legitimate frames, resulting in a denial-of-service.

5. Based on the receiver duty cycle, the attacker can periodically retransmit the malicious headers.

The effectiveness of this attack depends on the specific configuration used for the waveform and the duty cycle for the targeted communication system. If the maximum payload size is much greater than the typical frames being transmitted, an attacker can easily force the receiver to miss multiple legitimate frames with relatively little effort on the attacker's part. This allows for a low power, course synchronization denial-of-service scheme that can be more efficient than other wireless jamming techniques.

Traditionally, an attacker would need to use a high power transmitter to reduce the apparent signal-to-noise ratio (SNR) at the receiver. More sophisticated attacks attempt to target specific characteristics of the message in order to reduce the required duty-cycle for the jammer. However, this can be difficult to achieve because the attacker must first detect and synchronize to transmitted frames and quickly overpower it before the receiver can detect and demodulate the transmitted frame.

With this exploit, the attacker no longer needs to overpower or synchronize to any transmission; they simply need the receiver to process the malicious header before a legitimate header is transmitted. If the malicious header is timed correctly, the receiver will simply drop the legitimate frames as part of the fake payload when the checksum validation fails. If the attacker knows the exact time the receiver needs to process the non-existent payload, they could simply transmit the malicious headers at that frequency and effectively disable the communications system completely.

4.2 Un-sanitized Control Parameters

Many vulnerabilities arise in applications because the developers simply assumed that input coming from an external source would always match a certain format, contain values within some defined range, or contain specific content. In fact, the majority of vulnerabilities that exist today are probably due to this in some manner. Failing to properly check external input and verify that it meets expected properties can lead to major vulnerabilities.

The second vulnerability we present is an example of how this can occur in a software radio. In this case, the control parameters through a packet header were not properly sanitized which led to unexpected behavior in the application. This could either result in an improperly configured waveform or even a crash, both of which would cause a denial of service.

Implementation

The vulnerable implementation is a dynamic spectrum access waveform that utilizes LiquidDSP's OFDM FlexFrame under the hood. This code was designed for a cognitive radio competition and implemented a basic coordinated, frequency-hoping algorithm that used a small number of channels within the frequency band allowed for the competition. The system was made up of two nodes using Ettus Research Universal Software Radio Peripherals (USRPs) [75] implementing a duplex communications link with one or more primary users in the same frequency bands.

Upon initialization, the nodes would begin probing the available channels attempting to synchronize with each other. If they were able to successfully detect each other, they would attempt to find an unused channel. Each node was responsible for determining the quality of the channel it was receiving and whether to hop to a new channel. If the current channel was not desirable due to a low signal-to-noise ratio or the radio detected a primary user, the nodes would coordinate with each other before hopping to a new channel.

The coordination was achieved using the 6 bytes available to the user in the OFDM FlexFrame header. Rather than transmit the actual frequencies for the transmit and receive channels for each radio, a single integer was used to describe the current channel in use. Due to the hopping algorithm, each radio's control information contained four bytes: the current transmit and receive channels and the requested transmit and receive channels. The specific channel configurations were stored using a fixed length float array, and the input parameter was used as the array's indexing when determining the correct channel frequency.

Vulnerability

The vulnerability in this system was the fact that the input fields for these control values were not sanitized before being used to determine the correct destination channel. Because each parameter was a single byte, this could allow an attacker to input any value up to 256 (shown in Figure 4.1). However, the number of actual channels configured for hopping was far lower than the maximum value that could be specified for any of the control values. This resulted in a buffer over-read where the software would read from a memory location outside of the actual channel configuration array if the input control value was greater than the number of configured channels. The management component would read from unknown memory and would retrieve an invalid center frequency for the new channel.

Field	Description	Expected Range	Actual Range
0	Transmit Channel	0-8	0-255
1	Transmit Bandwidth	0-6	0-255
2	Requested Channel	0-8	0-255
3	Requested Bandwidth	0-6	0-255
4	Receive Channel	0-8	0-255
5	Receive Bandwidth	0-6	0-255

Table 4.1: Control fields for a cognitive OFDM waveform using LiquidDSP. These control parameters were not properly sanitized in received frames which could result in unexpected behavior of the system if malicious control parameters were transmitted by an attacker. Each parameter only expected a small range of values, but since full bytes were used for each field, a much larger range could be transmitted.

Because of this, there is no guarantee of what value would be passed to the USRP hardware driver when attempting to tune the USRP to a new channel. Most likely, whatever value was read from the garbage memory location would not be a valid float value that was within the correct tuning range of the hardware.

If this unknown value is passed to the driver to change the USRPs center frequency, a couple of different scenarios could occur. The most probable outcome is that the invalid channel would return a float that does not represent a supported frequency. When UHD detects such a tuning request, it automatically tunes the USRP to a default frequency for that specific model. At this point, the targeted node would be attempting to receive a signal outside of the valid channel frequencies. The other possible scenario is that the invalid channel returns a float representing a frequency that is within the USRP's tuning range. In either, case the USRP would no longer be receiving on one of the valid channels which would result in a denial-of-service.

Depending on which value an attacker attempted to corrupt in the forged frame header, this control parameter may also be propagated to the other node. In such a situation, both nodes would be attempting to change their transmit/receive frequencies to the same invalid channel. The best case scenario is that both radios attempt to tune with an invalid center frequency and they both jump to the default frequency. At that point, the radios would be able to re-synchronize, but would be outside of the allowed bands of operation; eventually, they may hop on their own back to a valid channel. The worst case scenario is the nodes tune to different frequencies and completely lose synchronization.

This vulnerability could possibly induce a crash in the system in multiple different components; for example, the driver hardware for a different type of software radio may be buggy and unable to handle invalid frequency requests.

In a newer version of the radio software, additional parameters were added to control the transmit and receive channel bandwidth as well as the forward error correction being used. With more parameters, the number of bits used for each parameter was reduced, effectively patching this vulnerability.

4.3 Buffer Overflows

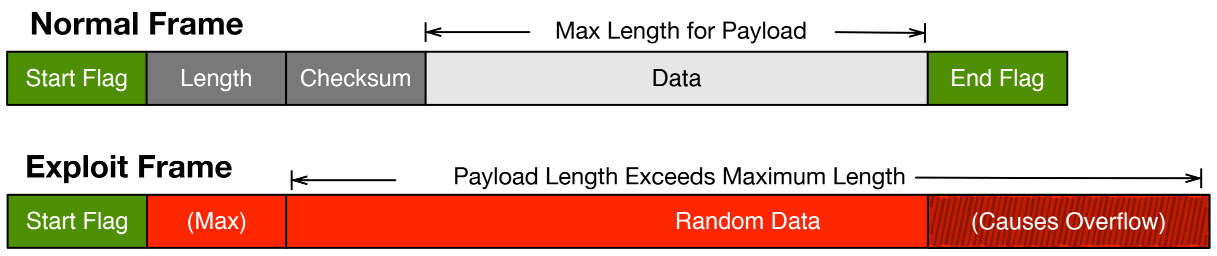


Figure 4.4: Example of normal frame structure versus an attack frame. The normal frame structure is shown above and a continuous exploit frame is shown below. Here the attacker transmitted a start flag but never transmits the end flag resulting in a heap overflow in the waveform.

To demonstrate the feasibility of attacks against SDR vulnerabilities, we developed an example SDR waveform that is vulnerable to a simple buffer overflow in one of the processing blocks of the system, and demonstrate a remote, over-the-air exploit of the vulnerability. It is caused by a message size mismatch between two different blocks in the waveform and can allow an attacker to execute arbitrary code on the targeted system.

For the purposes of this paper, overflow mitigations previously mentioned (ASLR, DEP, and stack canaries) have been configured on the test system in order to better mimic the configuration of an embedded system.

In this section, we briefly discuss the basics of buffer overflow vulnerabilities and also present two different examples of signal processing blocks in a GNU Radio waveform that are vulnerable to overflows. We then demonstrate exploiting these vulnerabilities with a remote, over-the-air attack that allows an attacker to modify the behavior of a software radio through an external attack vector.

4.3.1 Heap Overflow Exploit

The first example of a vulnerability we explore is a heap overflow that results in a denial-of-service attack against the target radio when exploited. We designed a simple Orthogonal Frequency-Division Multiplexing (OFDM) modem using both GNU Radio and LiquidDSP that creates a bursty, duplex wireless link between two different nodes. The waveform includes a virtual network interface that enables the nodes to communicate with each other using normal network traffic.

The waveform uses a framing protocol above the OFDM physical layer that is similar to many existing data link layer frames and is shown at the top of Figure 4.4. The frame structure itself consists of three sections: the length, a checksum, and the payload. Pre-defined, 16-bit flags mark the start and end of a transmitted frame. Both the length and checksum are 16-bits (unsigned shorts) and immediately follow the frame header. The payload is variable length but is limited to a maximum length or maximum transmission unit (MTU), defined at design time.

The waveform includes a frame synchronizer block that is vulnerable to a buffer overflow attack. This block is responsible for searching the incoming bitstream for new frames, validating the payload, and finally passing the validated payload to the higher layers in the network stack. The waveform contains an overflow vulnerability due to a poorly designed implementation and an assumption that a transmitted frame will never exceed the maximum length defined by the protocol itself. Because of this assumption, the internal buffer used for saving incoming frames is a fixed length, pre-allocated buffer whose size is equal to the maximum payload length. This fails to account for a situation where an attacker purposefully transmits malicious frames that exceed this length.

The synchronizer uses a sliding window to search for the start flag (indicating a new frame is being received). Once this flag is detected, the length and checksums for the frame are saved and the block continues to push received bits into its memory buffer until the end flag is detected. After a full frame is received, the synchronizer validates the payload and delivers it to the network stack. If the checksum validation fails, the frame is dropped.

An overflow vulnerability exists because the synchronizer fails to immediately recognize a frame has exceeded the maximum length. Rather than immediately dropping the offending frame, the synchronizer continues to search for the ending flag and continues to save incoming bits into its internal buffer. This buffer used for storing the incoming payload is declared as a *C++* class member and is allocated in the class data structure in the heap segment. Overflowing this buffer can corrupt the *C++* class object itself and any other data structures stored nearby in the heap.

The simplest exploit of this vulnerability is transmitting a start flag and never transmitting an end flag (shown in Figure 4.4). The payload itself, along with the other fields in the frame, can be random data. Since the synchronizer is searching for the end flag, it would not exit this mode and would continue to save incoming data to its buffer. If enough memory is corrupted by the overflow or the synchronizer attempts a write operation to an invalid memory address, the waveform will eventually crash. This causes a persistent denial-of-service attack until the waveform is restarted on the targeted system. We can demonstrate this result by transmitting a continuous frame until the receiver waveform crashes. In our specific example, the receiver quickly crashes because an index is corrupted the synchronizer tries to write to an invalid or unauthorized memory address in the system and is killed by the operating system.

In addition, an attacker could exploit the overflow using specially constructed frames that overwrite internal class variables with malicious values. This more sophisticated attack would require some knowledge of the specific implementation being targeted. Attackers could use this technique to modify variables and maliciously modify the behavior of the waveform itself.

4.4 Stack Overflow Exploit

Waveform

The waveform demonstrating a stack overflow vulnerability was designed to simulate an IoT sensor node that is part of a larger mesh network. Nodes within the network communicate with each other using an OFDM physical layer link, with an optional backhaul network connection for internet connectivity. The main component implementing this connectivity within the flow-graph is the *router* block, which has several message passing inputs and outputs for linking the different blocks in the waveform. The router implements some basic link layer functionality, where each message has a source and destination address, a 2 byte header, the payload, and a frame checksum. A diagram of the waveform is shown in Figure 4.5; a fully implemented node would include a duplex physical layer plus the additional socket for the backhaul connection.

The second example of an overflow vulnerability in a software radio we present is a stack based buffer overflow. This vulnerability is susceptible to a more sophisticated attack and results in arbitrary code execution on the targeted system. To demonstrate the feasibility of this type of vulnerability, we designed an example GNU Radio waveform with a vulnerability in a message passing block that can be exploited with this more advanced attack. As we mentioned previously, defenses against stack buffer overflows do already exist in traditional desktop or server based operating systems, but embedded and real-time systems may not have these same defenses and can still be susceptible to these attacks. So, for our demonstrations, these defenses were disabled on a Linux desktop system to mimic an embedded, real-time system.

Our example waveform simulates a small sensor node that is part of a larger mesh network. Nodes within the network communicate using an OFDM link, with an optional backhaul network connection for internet connectivity. A *router* block implements some basic link layer functionality, where each message has a 6 byte source and destination address, a 2 byte header, the payload (with a maximum size of 256 bytes), and 4 bytes for the frame checksum. Frames not addressed to the receiving node are dropped; correctly addressed frames are forwarded to the correct block in the waveform based on the received packet's type. A flowgraph of the basic node is shown in Figure 4.5.

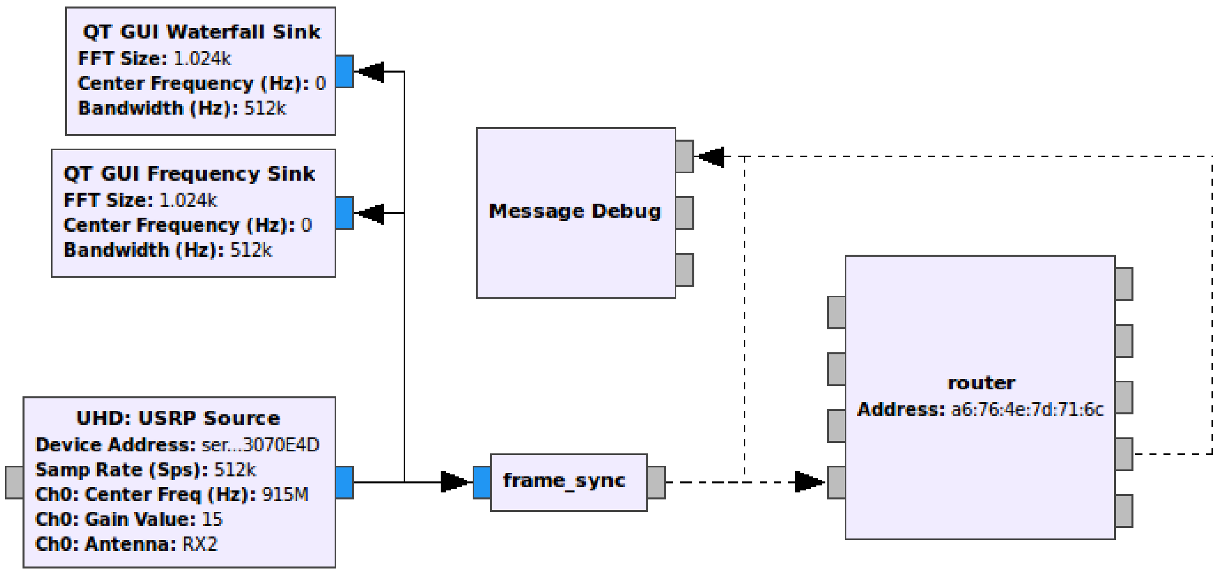


Figure 4.5: GNU Radio receiver for a sensor node using a vulnerable router block. The router block improperly handles incoming messages that are too long, which results in a stack buffer overflow.

The vulnerability in our example waveform mainly exists due to how the router block processes incoming messages. Like our previous example, this example uses the same assumption that the length of a message will not exceed the maximum length of a transmitted frame for the application (in this case it is 256 bytes). In the router's handler function (*handle_link*), the block copies an incoming message's payload into a buffer before processing it and without performing a length check on the payload of the received frame. A simplified version of the vulnerable function is shown in Figure 4.6. If an incoming message is too long, an overflow will occur during the memory operation (Line 9) that copies the received payload to a fixed length buffer declared in the function. The assumption is that the length of a frame received by the router would not exceed the maximum length of the payload defined in the link layer protocol.

However, there is no guarantee that a received message will not exceed this maximum length and the length is never checked prior to moving it to the new buffer. So, whenever a frame is transmitted that exceeds the maximum payload length, an overflow will occur on the stack that corrupts local memory. This situation can occur because other blocks within our waveform can handle much larger frame sizes than the router block. Our OFDM physical layer is implemented using LiquidDSP's FlexFrame module that can have a much longer maximum payload than the rest of the system (up to 65535 bytes). The OFDM block simply receives an incoming frame and creates a new message in GNU Radio that is passed to the router block; there is no size limitation on the length of this message other than the length of the received OFDM frame. Whenever a frame is transmitted that exceeds the maximum message length for the router, the OFDM block will generate a message that causes a stack overflow when it is processed by the router block. This is a feasible vulnerability because waveforms can consist of various blocks from existing modules and developers may use these implementations and fail to account for possible size mismatches like the one demonstrated in our example.

```
void handle_link(pmt::pmt_t msg) {
  if (pmt::is_pair(msg)) {
    unsigned char payload[256];
    ...
    pmt::pmt_t packet = pmt::cdr(msg);
    uint8_t * data;
    data = (uint8_t*)pmt::blob_data(packet);
    ...
    uint16_t len = pmt::blob_length(packet)
    memcpy(payload, data+14, len-18);
    ...
  }
}
```

Figure 4.6: Vulnerable message handling function in the router block. The *router* block's message handler that is vulnerable to a stack based buffer overflow. The overflow will occur if the incoming message length is greater than 256 bytes. An attacker can exploit this vulnerability to possibly inject code and change the behavior of the application.

This vulnerability was exploited in two major ways: a stack smashing attack and shellcode injection attack. A stack smashing attack is the more simple of the two and is essentially the same as the generic overflow exploit described previously.

Stack Smashing

Stack smashing occurs when an overflow is used to corrupt important memory locations within the current stack frame. The goal is simple: use an overflow to corrupt as much of the stack as possible until the application crashes. When the currently executing function completed, the application would attempt to use this corrupted control flow information and

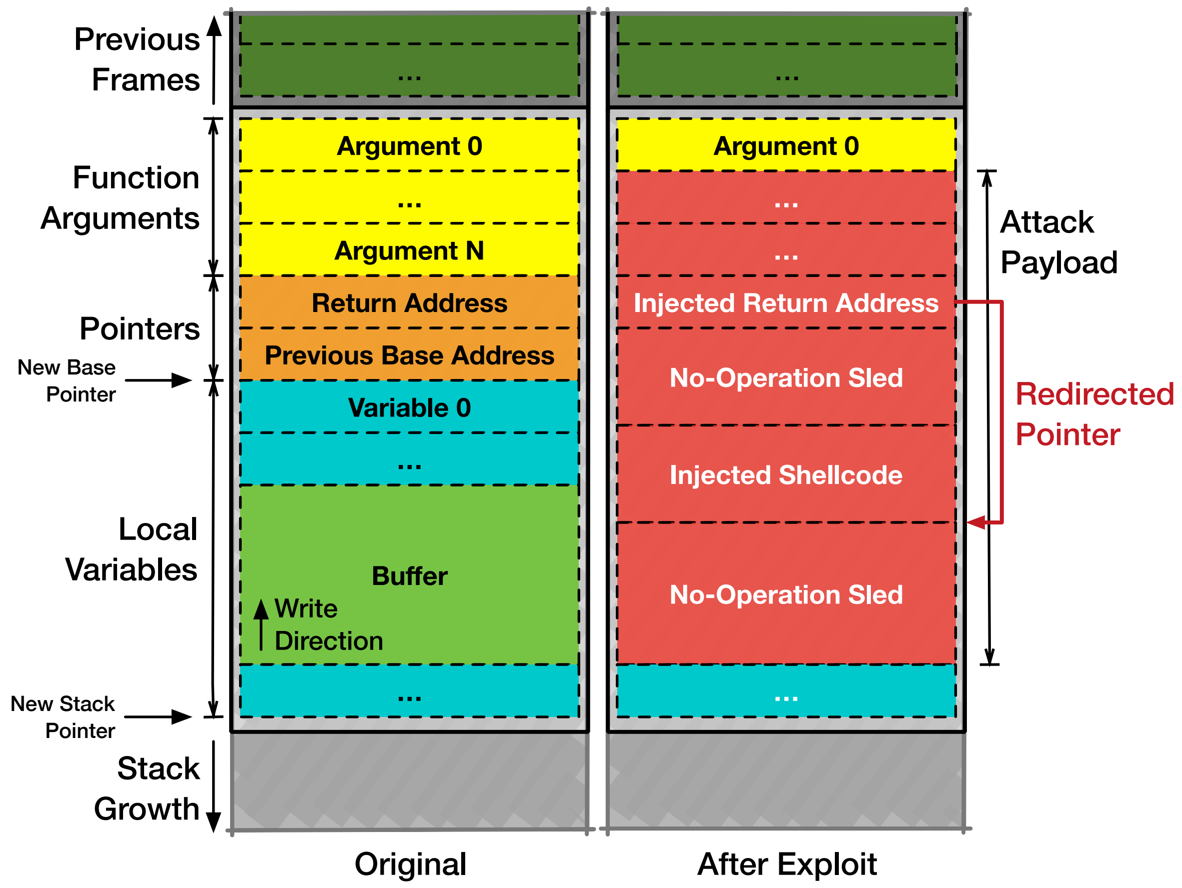


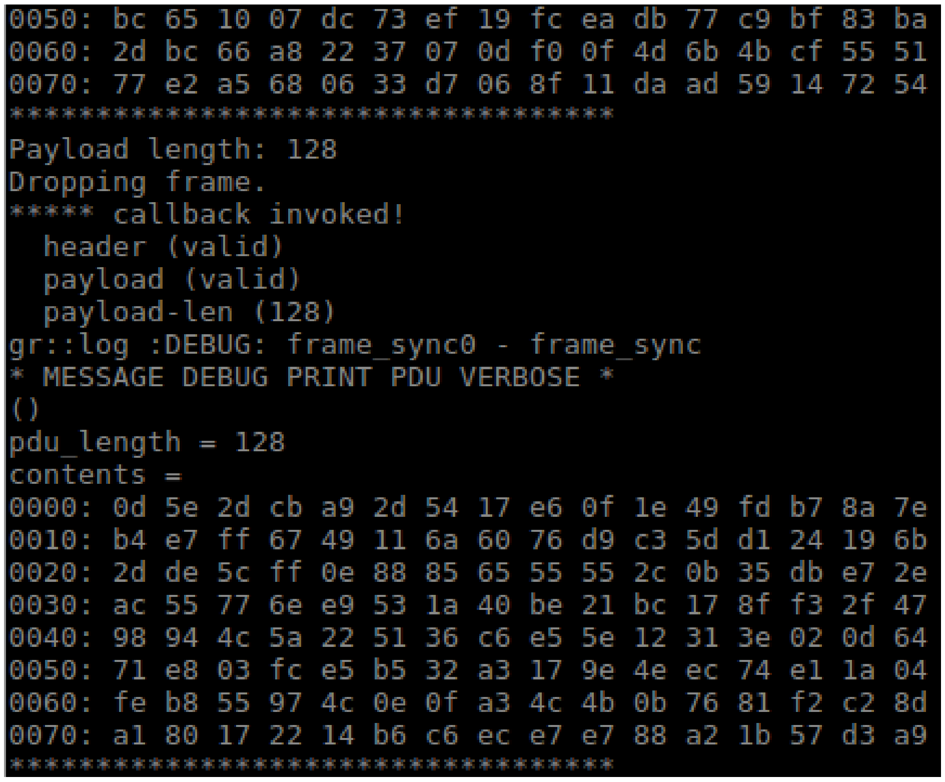
Figure 4.7: Example of a stack layout after a successful exploit. The shellcode has been written to the original buffer, and the overflow vulnerability has allowed the attacker to overwrite the original return address to now point to the injected shellcode.

would result in a memory fault and crash. We were able to demonstrate a stack smashing attack by transmitting a very large frame with randomly generated data. The message exceeded the length of the router's allocated buffer and corrupted the stack, causing the waveform to crash once it was received. This resulted in a denial-of-service attack until the waveform was restarted.

Shellcode Injection

The shellcode injection attack is a much more sophisticated attack that requires an in-depth understanding of the stack structure of the application in order to successfully exploit it.

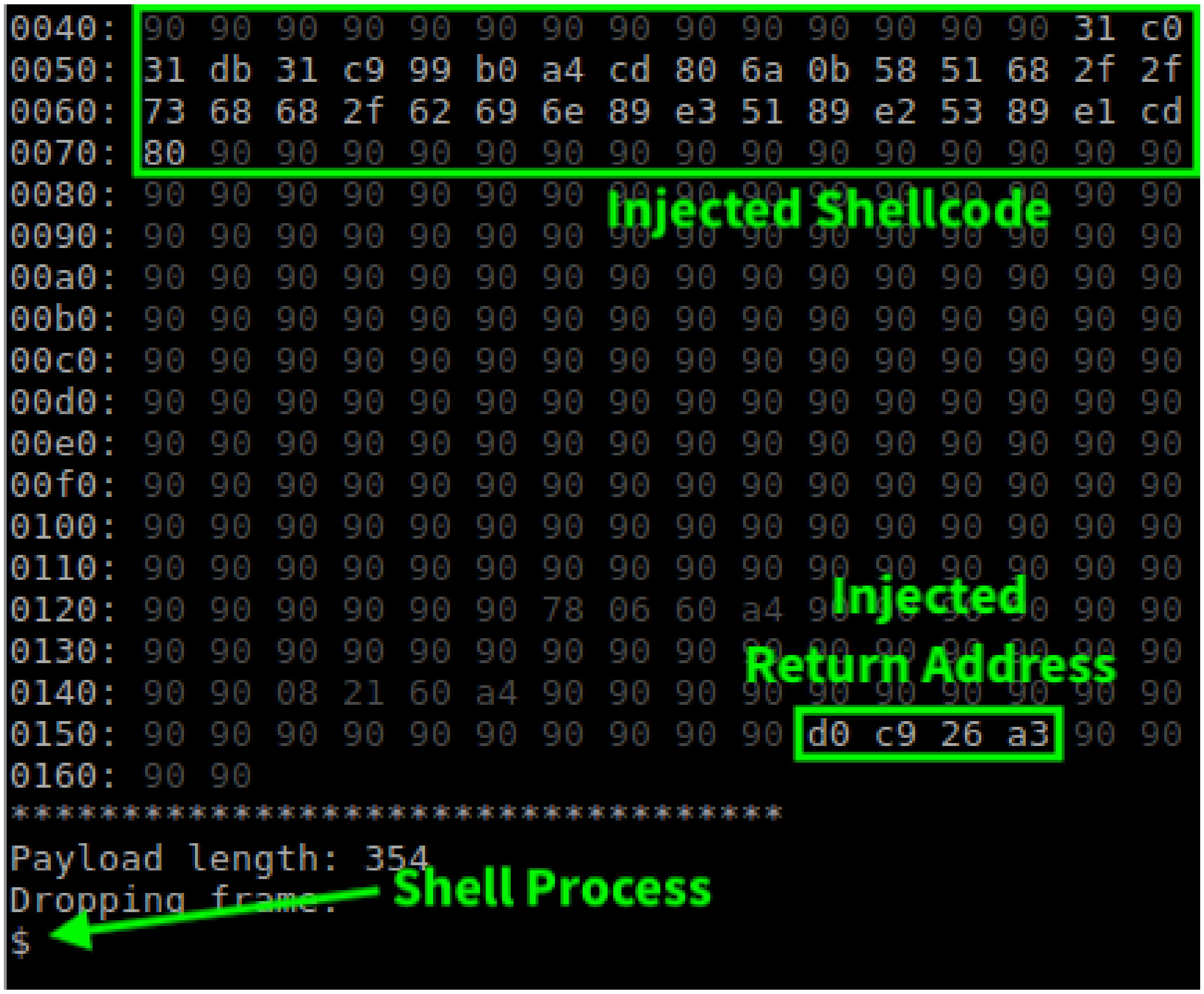
An attacker could determine the stack structure through multiple methods; two examples include reverse engineering the binary or executing the application with a debugging utility. The objective is not to simply corrupt the current stack but instead modify the stack frame to control the execution flow of the application. The goal is exploiting the overflow vulnerability to overwrite the return address in the current stack frame with a malicious address that points to shellcode previously injected by the attacker. The injected shellcode could also be contained in the malicious frame's payload, but could also be injected through other means. If the exploit is successfully constructed and injected into the system, the system will jump to the injected shellcode rather than the original return address when the currently executing function has finished executing. An example of a stack frame after a shellcode injection exploit has occurred is shown in Figure 4.7.



```
0050: bc 65 10 07 dc 73 ef 19 fc ea db 77 c9 bf 83 ba
0060: 2d bc 66 a8 22 37 07 0d f0 0f 4d 6b 4b cf 55 51
0070: 77 e2 a5 68 06 33 d7 06 8f 11 da ad 59 14 72 54
***********************************
Payload length: 128
Dropping frame.
***** callback invoked!
  header (valid)
  payload (valid)
  payload-len (128)
gr::log :DEBUG: frame_sync0 - frame_sync
* MESSAGE DEBUG PRINT PDU VERBOSE *
()
pdu_length = 128
contents =
0000: 0d 5e 2d cb a9 2d 54 17 e6 0f 1e 49 fd b7 8a 7e
0010: b4 e7 ff 67 49 11 6a 60 76 d9 c3 5d d1 24 19 6b
0020: 2d de 5c ff 0e 88 85 65 55 55 2c 0b 35 db e7 2e
0030: ac 55 77 6e e9 53 1a 40 be 21 bc 17 8f f3 2f 47
0040: 98 94 4c 5a 22 51 36 c6 e5 5e 12 31 3e 02 0d 64
0050: 71 e8 03 fc e5 b5 32 a3 17 9e 4e ec 74 e1 1a 04
0060: fe b8 55 97 4c 0e 0f a3 4c 4b 0b 76 81 f2 c2 8d
0070: a1 80 17 22 14 b6 c6 ec e7 e7 88 a2 1b 57 d3 a9
***********************************
```

Figure 4.8: Console output showing normal receiver behavior. In this instance, a 128 byte message has been correctly received and the contents of its payload displayed on the console.

Using a debugger application (the GNU Project Debugger), we were able to breakpoint and analyze the running waveform and examine the memory layout of the stack during the execution of the *handle_link* function. The important values that needed to be determined from this type of in-depth analysis include the location of the return address relative to the start of the buffer and the start address of the buffer itself. Once the location of the return



```
0040: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0
0050: 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 2f 2f
0060: 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 e1 cd
0070: 80 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0080: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0090: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00a0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00b0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00c0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00d0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00e0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00f0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0100: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0110: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0120: 90 90 90 90 90 90 78 06 60 a4 90 90 90 90 90 90
0130: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0140: 90 90 08 21 60 a4 90 90 90 90 90 90 90 90 90 90
0150: 90 90 90 90 90 90 90 90 90 90 d0 c9 26 a3 90 90
0160: 90 90
************************************

Payload length: 354
Dropping frame.
$
```

Figure 4.9: Console output showing a successful exploit. The output shown includes the injected shellcode, corrupted return address, and a new shell prompt triggered by the shellcode execution.

address relative to the buffer and the start address of the buffer was known, it was possible to build an exploit payload to accomplish the injection. The exploit payload itself consists of several components: the NOP sled, the shellcode, and the new return address [46]. An example of this payload is shown in Figure 4.7. This payload contained the shellcode and a malicious return address pointing to the injected shellcode in the buffer. This shellcode is binary, executable code that is written for the target's specific hardware architecture, which in our demonstration is the common x86 architecture. The shellcode is the actual payload that will be executed on the target system once the application jumps to the malicious return address. For this exploit, the shellcode simply calls the *exec()* system call that launches the *sh* shell process which takes over the execution context of the original waveform. The attack has two results: a persistent denial-of-service since the original waveform is replaced by *sh* and arbitrary code execution on the host. The rest of the payload was filled with the no operation opcode (NOP) for the targeted processor architecture. For our specific demonstration, the NOP sled was not really needed, but its main purpose is to allow the malicious return address to point anywhere within the stack buffer in situations where the exact location of the buffer starting address is not known.

To demonstrate this exploit over the air, we also implemented an attack waveform that periodically transmitted a random 128 byte message simulating normal sensor network traffic. The attack waveform would also periodically transmit the injection attack and attempt to exploit the receiver. We were able to successfully cause the receiver to jump to the injected shellcode, allowing us to execute arbitrary code on the targeted system.

The exploit payload itself consists of several components: the NOP sled, the shellcode, and the new return address [46]. An example of this payload is shown in Figure 4.7. The shellcode is the actual payload that will be executed on the target system once the application jumps to the malicious return address. This shellcode is binary, executable code that is written for the target's specific hardware architecture, which in our demonstration is the common x86 architecture.

For our example exploit, the shellcode simply calls the *exec()* system call to launch a shell process that takes over the execution context of the original waveform. We were able to successfully cause the receiver to jump to the injected shellcode triggering the waveform to execute the *sh* process which takes over the receiver waveform's execution. Figures 4.8 and 4.9 show screenshots of the normal behavior of the receiver and a successful shellcode injection respectively. Figure 4.9 shows the output of the receiver when the *sh* was triggered and shows the received malicious payload, the injected shellcode, the malicious return address, and the resulting shell prompt. In the second figure, the exploit payload is shown from the debug output and the NOP sled, shellcode, and malicious return address is clearly visible. Also, the shell prompt at the end of the output indicates the waveform executed the shellcode and launched the shell process.

Simply launching a shell process as part of the shellcode exploit is not a very effective attack for this example, but it does prove that the exploit is possible. An attacker could use more complicated shellcode to better exploit the system like exploit the kernel. Much more sophisticated shellcode could be used to open up backdoor access for the attackers allowing them to remotely control the targeted system. By changing the shellcode injected in the payload, an attacker is able to execute almost anything on the host. One example is opening a reverse network connection back to the attacker, giving them a backdoor into the sensor itself and also the sensor network.

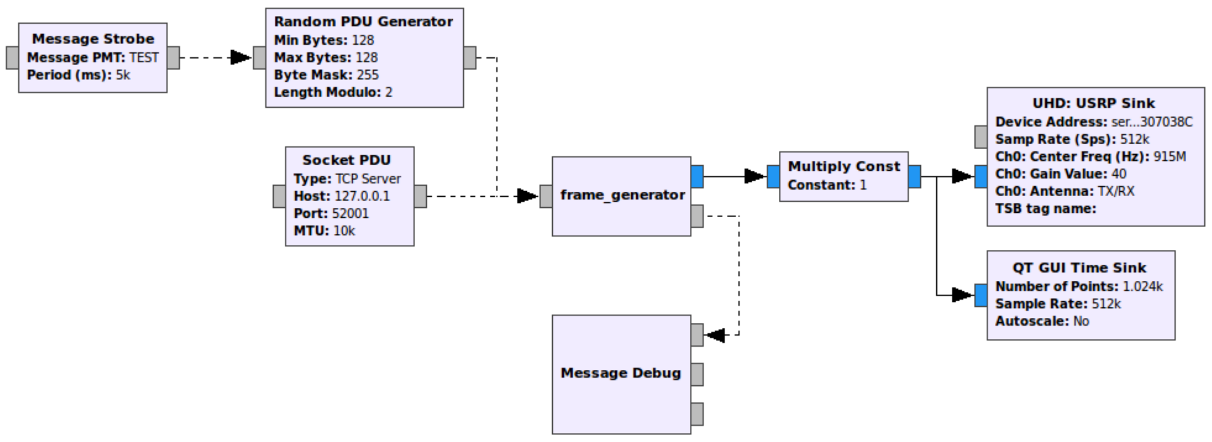


Figure 4.10: Example GNU Radio exploit flowgraph. This flowgraph transmits randomly generated 128 byte payloads that are normally expected by the sensor node flowgraph. It also allows larger payloads to be injected, which can be used to exploit the receiver flowgraph.

4.5 Summary

Software defined communication systems have introduced a new class of threats and a new attack surface where adversaries target vulnerabilities in the implementations themselves. Because modern wireless standards are complex and are rapidly evolving and systems are implemented quickly, common software vulnerabilities can easily occur within SDR implementations. However, there is not a significant amount of previous work that has focused on these types of attacks. In this chapter, we specifically focus on these weaknesses within SDRs and present several example vulnerabilities and demonstrate exploiting them.

While there are defenses that exist for these types of common vulnerabilities, they are typically implemented in desktop and server operating systems and not embedded systems. Many SDRs are implemented on embedded systems and can be vulnerable to these types of attacks. The best defense is following secure coding practices, but it can be difficult to catch every single possible vulnerability within complicated software systems.

In this chapter, we presented several examples of vulnerabilities in SDR implementations and demonstrated exploits of these weaknesses from a wireless attack vector. With the control flow attack, we showed that injected malicious headers could prematurely trigger state changes in the waveform that could cause legitimate messages to be dropped unintentionally. If properly timed, the receiver can be effectively jammed resulting in a denial-of-service against the system. Our next example demonstrated a vulnerability where the receiver did not properly sanitize input control parameters. In this case, the attacker could inject invalid control information which triggered buffer over-reads and could result in the radio being misconfigured resulting in unexpected behavior and a persistent denial-of-service.

Next, we demonstrated the ability to exploit both heap and stack buffer overflow vulnerabilities in the SDR. These exploits resulted in persistent denial-of-services against the system, as well as, the ability to exert control of the waveform remotely by injecting and triggering arbitrary shellcode on the target. An example heap-based buffer overflow in a frame synchronizer block could be exploited by transmitting a frame that exceeds the protocol's maximum length and results in corrupted memory and class variables triggering an eventual crash of the system. The stack-based overflow allowed both arbitrary, remote code execution on the targeted system and a denial-of-service attack through a simple stack smashing similar to the heap overflow attack.

We have shown that wireless systems implemented in pure software can be susceptible to traditional cyber-security attacks. Since SDRs are increasingly used as production systems, this threat will continue to increase. Protecting the system against these types of exploits and preventing further exploits by an attacker is the main motivation for our isolation based, defense-in-depth architecture presented in the next chapter.

Chapter 5

Defense-in-Depth Architecture for Software Radios

With this shift towards software in communications systems, any framework for developing secure wireless communications systems must account for vulnerabilities that exist within the radio implementation itself rather than just isolating the application layer or securing protocols. The related security models we discussed in previous chapters proposed using isolation to protect the radio from misconfiguration due to exploits in an application using a SDR waveform rather than the waveform itself. Isolating the application layer to protect from exploits there is important, however vulnerabilities within the lower layers of the signal processing stack can lead to the entire system (both waveform and application) being compromised. The application layer would never need to be compromised if the lower layers can be exploited, so only isolating the application layer is insufficient.

In the previous chapter, we demonstrated how these vulnerabilities in a SDR could be exploited to cause unexpected behavior and even allow arbitrary code execution. As applications grow more complex, there is a greater chance of this type of vulnerability existing that can be easily overlooked. While secure coding practices are the best defense, many times unknown vulnerabilities still exist within applications (known as zero-day vulnerabilities) and can be extremely difficult and time intensive to identify.

This chapter presents a generic defense-in-depth architecture for securing SDRs across all software layers by isolating components of the system (especially high-risk ones) into different domains. Adding multiple layers of isolation into the radio using techniques such as virtualization helps construct a system that provides better security than the monolithic SDR architecture. Attacks against vulnerabilities in the waveform itself can be contained within the specific domain and prevented from affecting the rest of the radio. This allows for isolated domains that can be both secured against outside attackers, but also secured against inside attackers attempting to gain unauthorized access to data.

In this chapter, Section 5.1 presents a general overview of the entire proposed defense-in-depth SDR architecture. Section 5.2 then details the Security Plane which is the key component of the architecture that provides the isolation mechanisms, devices drivers, and inter-process communication mechanisms need to implement an entire waveform. Sections 5.3 and 5.4 then describe the Control and Data/Application Planes of the architecture. Section 5.5 briefly details how policy management would be handled within our architecture.

In Section 5.7 we describe some challenges that are associated with our security architecture.

This chapter also gives a brief description of how policy management would work in the proposed security architecture, but an in-depth discussion of how policies are defined and example policies and their enforcement is outside the scope of this dissertation. However, it is important to understand how hardware policies would be enforced in the architecture which is briefly discussed.

5.1 Overview

The primary goals of the defense-in-depth architecture are providing mechanisms for 1) limiting the effectiveness of exploits against vulnerabilities by stopping further compromise of the SDR and 2) detecting the malicious behavior and mitigating the threat. This architecture employs a similar design philosophy to the micro-kernel architecture used in various operating systems, where the intent is reducing the core of the system to the smallest possible Trusted Computing Base (TCB) and isolating every other component within the system. This is accomplished by moving the components of the waveform, especially high risk components, into separate isolated domains which limits the effectiveness of an exploit against any individual components by preventing the entire waveform from being compromised. While making any complex system completely secure is practically impossible, this architecture is designed to start with a high-assurance core and layer defenses around the SDR components to make it difficult for attackers to completely compromise the system through exploiting a single component. This approach is also comparable to traditional operating system security models such as the hierarchical privilege model where components are placed at different privilege levels in the system. The strength of this model is isolating each component or group of components of the system into their own domains and keeping the core of the system as minimal as possible.

By isolating each component in the system, the CIA security model can be better enforced. If one isolated component is compromised, the confidentiality of the system is protected through isolation; the attack would be contained within the isolated environment. Integrity is protected through monitoring each of the virtual environments. If an attack is detected, the environment can be reset back to a nominal state without affecting the rest of the waveform. Availability is also protected through monitoring and isolation. Resource limits are enforced by the isolation environment, and if a component crashes, the monitoring system can detect the crash and immediately restart the component.

The overall architecture (shown in Figure 5.1) can be divided into three functional planes:

- The *Security Plane* is the core of the system and is the most critical layer for ensuring overall system security. It constitutes the TCB and provides the isolation and separation mechanisms used to isolate all of the components in the system, device drivers for

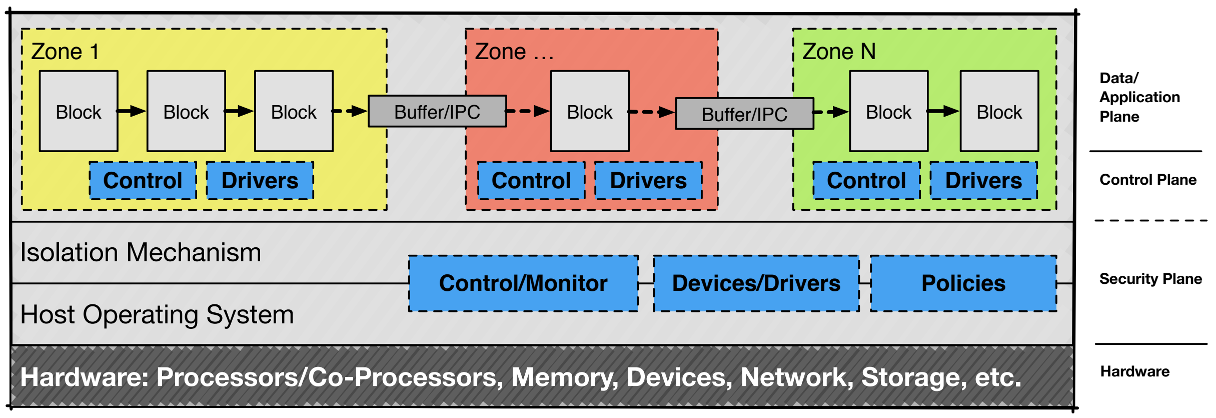


Figure 5.1: Basic defense-in-depth SDR architecture - This security architecture consists of three major layers: 1) the Security Plane which comprises the trusted computing base and includes the isolation techniques, and hardware device drivers, monitoring utilities, and policy enforcement utilities; 2) the Control Plane which includes command and control functionality for the waveform; and 3) the Data/Application Plane which includes the various applications, services, and waveforms that comprise the communications system. The dashed line between the Security and Control Plane indicates some of the control capabilities must also reside in the host layer and not simply the isolated environments.

hardware, and inter-process/inter-domain communication for connecting components. Monitoring for the isolated environments and policy enforcement also exist within this plane and are useful for detecting malicious behavior in the system and protecting against mis-configuration.

- The *Control Plane* contains all components necessary for controlling, updating, and managing the system. It handles properly configuring the Security Plane based on the system's security requirements and manages the overall execution of the waveform and other applications.

- The *Data/Application Plane* consists of generic services, applications, waveforms, or third-party applications required to implement the communications system. These components implement the actual communication system, so they are exposed to attack from the radio's interfaces. Since they can contain exploitable vulnerabilities, each of these components or groups of components must be isolated to protect against exploits affecting other parts of the system.

5.2 Security Plane

The Security Plane is the core of the system; it is a minimal, hardened system that provides the functionality necessary for the fundamental operation of the system such as the isolation environments. Since this is the lowest and most trusted layer of the architecture and enforces the security for the application, it must be kept as minimal and secure as possible because any vulnerabilities in this layer could be exploited to compromise the entire system. It also handles the device drivers required to utilize different RF front-ends, co-processors, and other hardware, and also the inter-process-communication (IPC) mechanisms for components in the different layers to communicate with each other. These mechanisms themselves are implemented in the Security Plane, but are managed through the Control Plane discussed later. The Security Plane uses the Principle of Least Privilege where all isolated environments execute with as few permissions as possible. In the traditional hierarchical privilege model, the Security Plane is the most privileged ring of the system; all other layers must execute with fewer privileges.

5.2.1 Isolation

The core of the architecture is the isolation environments responsible for ensuring that components only have access to authorized resources such as available processors, devices, or memory. Memory isolation and protection is vital so that the individual environments are unable to access data within other isolation environments. Two examples of common isolation mechanisms are virtualization (hardware virtualization) and containerization (operating system virtualization). We briefly discuss a few isolation techniques below:

- *Virtualization* provides the highest level of isolation for components in the system, but also has the highest overhead, especially in systems lacking hardware virtualization support. A hypervisor creates a software interface that abstracts the physical system hardware away and provides a virtual machine (VM) that appears like a dedicated physical system to the software. A guest operating system can be installed on the VM and executed like a physical system; however, the guest environment is isolated from the host environment through the hypervisor interface and hardware support. One downside of this method is systems without virtualization support in hardware would incur a high performance overhead due to additional software emulation. So, constrained resource systems like embedded systems may not be able to use virtual machines as an effective solution. Examples of different virtualization solutions include KVM and VirtualBox [14, 76].

- *Containerization* provides a lightweight isolation mechanism for components in the system. Rather than abstracting hardware, containers abstract interfaces within the operating system where the kernel provides methods for isolating groups of processing into logical containers. This is implemented through *CGroups (Control Groups)* and *namespaces* in Linux; the Linux kernel creates different internal control structures for processes in each namespace. Since containers are not abstracting and virtualizing physical hardware but only virtualizing kernel structures, they provide lower overhead but also lower isolation since there is a shared kernel. Exploits against the shared kernel could lead to compromising the entire system. Example implementations of containers are Docker and Kubernetes [15, 77].

- *MicroVMs* provide a hybrid approach between virtualization and containerization; they provide a higher level of isolation than containers but much faster startup times, speed, and resource efficiency than virtual machines. ClearContainers are a solution that was introduced by Intel in 2015 (now Kata Containers) as a middle ground between containers and virtual machines. The goal was providing a system that is very similar to the workflow of containers but still provided the isolation of a full virtual machine. This is accomplished by using a virtual machine that runs a minimal kernel and a minimal amount of required software to implement the container interface. These have a much lower overhead than full virtual machines due to the reduced amount of software running in the VM. Examples of microVMs include kata-containers and Firecracker [78, 79].

- *Sandboxing* is the lowest level of isolation but provides the least amount of overhead. With this solution, the host operating system limits the functionality of the application so it cannot maliciously access critical system data. However, this solution can be the easiest to break. AppArmor and SELinux are examples of this approach. [80, 81]

- *Microkernels or separation kernels* reduce the amount of code running in kernel mode to the minimal amount required to build a functional system. This would include the scheduler, memory management, and inter-process communication for connecting components in the system. Device drivers and services in a microkernel run in user space, and applications access the devices through the message passing interface. Since this represents the minimal amount of code required to build a larger system, it can be far easier to harden. However, a downside to a microkernel is the increased development required to develop a system. One examples of a microkernel is DARPA's mathematically secured kernel that is formally proved to be secure [82, 83].

This architecture can employ various types of isolation for each component or group of components depending on the risk involved and any defined security requirements. Hardware support, component risk, security requirements, and performance requirements play key roles in determining what isolation mechanisms should be utilized for various application components. In some cases (like embedded systems), processors do not support full virtualization, so containerization is the only feasible option. Also, multiple isolation techniques could be utilized as a hybrid approach for an application to allow for greater control over the performance/security trade-offs. Higher risk components should have the highest degree of isolation from the rest of the system (*least privilege*), but these environments equate to higher overhead and lower overall performance.

In SDRs, the physical layer components are typically the lowest risk since they process digital samples rather than bytes of data but require high performance. Any vulnerability in these components would likely result in the system becoming corrupted or non-responsive rather than fully compromised as shown in the previous section. Once samples are demodulated to actual bytes and are passed to higher layers of the stack, the risk factor of these higher layer components increases since an attacker can directly control the data they process. For low level components, mechanisms like containers provide some isolation with less overall overhead, while more high-risk components are placed into more isolated environments like virtual machines. The trade-off of isolation versus performance is further discussed in the following chapter.

Extremely high-risk components can also be placed in nested isolation environments to enforce even more separation from the rest of the system. For example, an application may consist of components all isolated within different containers that are isolated within a single large virtual machine. This simply adds even more overhead to the system, but increases the isolation and adding more layers of defense around the components.

5.2.2 Device Drivers

The Security Plane is also primarily responsible for handling device drivers for physical hardware on the system. Since this layer implements the isolation environment for the different components in the system, it must also provide the appropriate interfaces to allow components to communicate with the underlying hardware from within the environment. A split device driver model is used where the main system driver exists in the Security Plane, and a minimal driver is exposed within the isolated environment itself (shown in Figure 5.2. In some cases, this interface can be a direct pass through from the virtual machine or container directly to the hardware. For example, RF data can be directly passed to the signal processing waveform in the isolated environment from the hardware. However, in order to enforce security and regulatory constraints, the driver would intercept control messages and block certain actions based on system policies.

Developing secure drivers and hardware interfaces for the isolation environments is vital since these are implemented in the core of the architecture, run with high privileges, and interface directly with the high risk components of the application. These drivers must have a minimal attack surface since vulnerabilities in the drivers could allow a compromised environment to escape, compromise the Security Plane and therefore compromise the entire system.

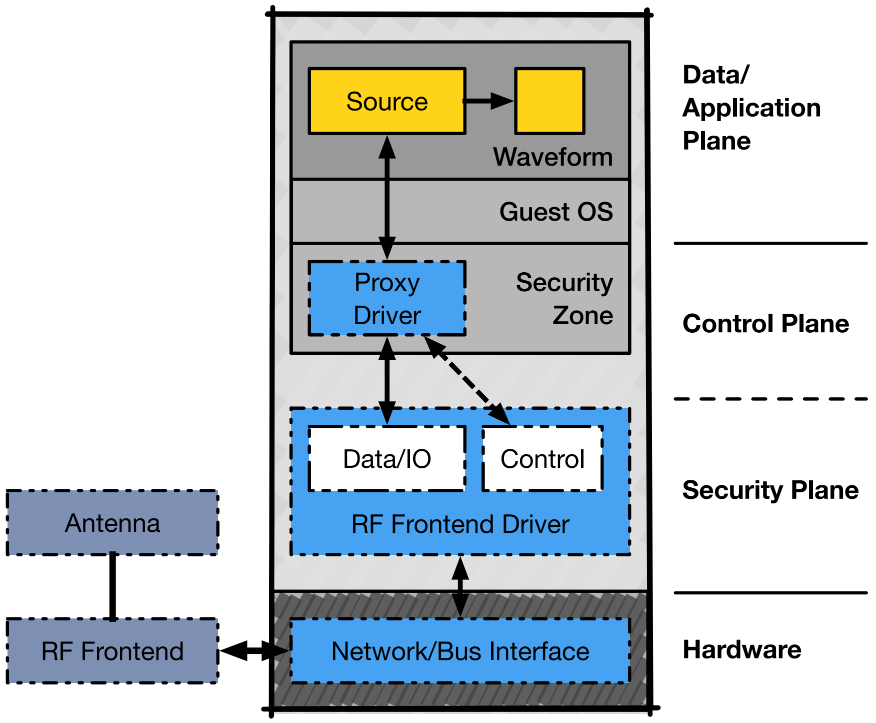


Figure 5.2: Split device driver model for the defense-in-depth architecture.

5.2.3 Inter-Process/Domain Communication

The final part of the Security Plane is inter-process/inter-domain communication between isolated environments which handles data flowing between isolated components in those environments. The IPC mechanisms can be implemented in several different ways depending on the isolated environments utilized. The most straightforward method is a network based IPC which uses the network stack built into each isolation environment. However, this method introduces significant overhead and traffic on the host's internal network interfaces which can result in longer latencies and reduced throughput. Other solutions include allocating shared memory buffers on the host and sharing these with the appropriate environments, which would provide a lower overhead implementation since it does not rely on the environment's network stack. For example, this could include the use of Linux Pipes to connect blocks executing within different container environments on the host.

5.2.4 Monitoring

Monitoring plays a critical role by detecting malicious behavior occurring within isolation environments and applying the appropriate techniques to mitigate the malicious activity. This involves inspecting the execution environment within the isolation environment from the outside and attempting to detect malicious execution. Monitored components are oblivious to this monitoring process and cannot easily influence it. If malicious activity is detected within a component or environment, an alarm could be triggered and actions taken to mitigate the behavior such as resetting the environment to a known safe state. There are several existing techniques for monitoring execution within isolation environments such as containers and virtual machines that can be implemented [84, 85, 86, 87]. Additional monitoring and intrusion detection systems (IDS) can be applied to the IPC mechanisms in the architecture. In this case, an IDS could monitor incoming data to different components in the waveforms or applications scanning for known malicious signatures or shell-code and could immediately block the attack before it could be processed by the targeted component.

5.2.5 Policy Enforcement

Another role of the Security Plane is policy enforcement which is an integral part of any secure SDR architecture. SDRs are inherently configurable, but production systems must operate under strict regulatory policies. While the specifics of how these policies are enforced is outside the scope of this dissertation, it is important to understand where policy enforcement exists within the defense-in-depth architecture.

Like the *Secure Radio Middleware* model discussed in Section 3.3, the policy enforcement components in our architecture exist outside of the isolated zones where the waveform and application components execute. This ensures that the components themselves cannot bypass the policy enforcement in the system. However, a major difference between this architecture and the SRM is the policy enforcement is not a singular component like in the SRM architecture. In that architecture, every message sent from the user application to the radio middleware was checked by the policy manager and enforced there before messages were passed to the radio middleware. The distributed nature of this architecture, where multiple components exist in separate environments, policy enforcement must also be distributed throughout the system. The policies themselves can still be stored by a singular component in the Control Plane which will configure the Security Plane with the acceptable policies, but the enforcement is handled on a per-component basis in the Security Plane.

Policies can be very broadly defined in SDRs applications that can span multiple aspects of the systems such as hardware configurations, resource management, Application Programming Interface (API) permissions, and application settings. The Security Plane is responsible for enforcing some of these types of policies, such as the legal hardware configurations for operation (this includes hardware front-end settings such as power, bandwidth, frequencies, etc), resource management, and authorized application interfaces. Policies for allowable application configurations and user actions should be enforced at the Data Plane.

First, the architecture should always enforce regulatory policies from attempts to misconfigure the system due to an exploit or user error. These policies define what configuration limits should be applied to the front-end hardware and therefore must be enforced at the driver levels. Each device driver exposed to the isolated environment must enforce policies through intercepting and blocking components from configuring the hardware outside of allowable configurations. This could include enforcing policies that are only valid during certain times or at specific locations; this type of approach to detecting and blocking invalid changes to the radio configuration was presented in [65].

Second, the Security Plane can also enforce policies regarding resource usage and authorized application interface usage based on the application security requirements. One threat specific to software radios is a denial-of-service due to exhausting system resources. Since this plane enforces separation between components in isolation environments, it can also enforce limits on resources consumed by each environment. In addition, this plane can enforce which components within the higher level planes are allowed access to different APIs or software interfaces implemented in the system.

5.3 Control Plane

The Control Plane of the architecture includes all components responsible for the overall control and operation of the system. This includes the functionality for launching different waveforms and configuring the Security Plane to implement the appropriate isolation environments required for the system. It also manages the interaction between components within the Data/Application Plane by establishing the proper IPC channels and configuring each isolation environment to have proper access to connected hardware. The Control Plane also includes any interfaces for management (both local and remote), intrusion detection systems, firewalls, and monitoring and auditing utilities. The Control Plane would also be responsible for secure software updates and installing and validating any new components on the system. Once new waveforms and applications are properly installed, the Control Plane would handle validating the waveform's integrity and securely booting and initializing the waveform. In a static system, the Control Plane may be very minimal and include only monitoring and auditing tools.

Since the Control Plane essentially manages the Security Plane, it is important that this layer also remains secure and implements only the minimum requirements so the available attack surface that could be exploited is small. Some of the components within the Control Plane (like remote management interfaces) should also execute within their own isolated environment to mitigate against possible exploits and use a minimal interface to communicate with the rest of the Control Plane. These components have exposed interfaces and handle data from untrusted sources, which could be used as an attack vector to compromise the system. Isolating these components limits the effectiveness of using these components as attack vectors into the rest of the system.

One of the main security objectives listed in the ITU-T report on Telecommunications Network Security [56] was ensuring a system provided proper accountability through requirements such as authentication, authorization, logging, alarm reporting, and auditing. In the defense-in-depth architecture, the Control Plane is responsible for handling all of these security requirements. Some requirements like identity verification, activity logging, and auditing can handled through the different exposed management interfaces (like Secure Shell or SSH) which provides user authentication, management, and logging. However, this only applies to the management interfaces and Control Plane of the system; the implemented application itself would also need to provide these controls. In addition, the Control Plane handles monitoring and policy enforcement for the system.

5.4 Data/Application Plane

The final plane of the architecture, the Data/Application Plane, includes all the SDR waveforms, system services, and end-user or third-party applications that are required to actually implement a communications system. Components at this layer are exposed through the wireless interfaces and may be vulnerable to attack and therefore should be treated as untrusted (with varying levels of risk). Each component executes within an isolated environment which is critical for limiting the effectiveness of an exploit against a vulnerability and preventing exploits from affecting the rest of the communications system.

The specific implementation of a waveform and how its components are grouped and isolated is highly dependent on each individual system and the requirements for security and performance are ultimately determined by the engineers developing the system. They can be organized in a variety of ways: For example, waveforms could be monolithic applications within a single isolated environment where the user applications execute in separate environments. Or a waveform's high-risk components and layers of the stack could be split into separate environments, implementing a micro-services type of architecture. The monolithic approach is similar to the current GNU Radio model and can be more vulnerable to an attack, but can also have slightly higher overall performance since there is no need for IPC. The distributed approach is more robust against attack because each component or layer is individually isolated, but this results in higher overheads and lower overall performance due to increased IPC between environments. Finally, waveforms could implement a nested approach where lightweight isolation techniques are embedded within heavier isolation environments to provide additional layers of security around the high-risk components. For example, each layer in the waveform could execute in an individual container with the entire waveform executing in a single virtual machine. However, this would add additional performance overhead due to the nested environments.

In addition to the waveforms, the Data Plane also consists of services and applications required for a fully functioning communications system. Again, the exact implementation and required applications are highly depending on the specific communications system. The important key is that each of these services or applications also executes within an isolated environment and require connections to other system components that are established by the Control Plane. Systems could allow for the installation of third-party applications that would be untrusted and are another reason for using high-isolation environments. Overall, this uses a design philosophy similar to that of micro-services that are commonly implemented for cloud computing systems. Examples of services and applications existing at this layer could include networking services like Virtual Private Networks, messaging clients, user interfaces, or even full platforms or operating systems.

5.5 Policy Management

Many communications systems have policies that define what system configurations are authorized for use by different users. These policies vary based on the system itself but can include settings such as the modulation scheme, error correction, encryption keys, framing protocol, and the channel. However, unlike the policies described earlier, these policies are specifically related to the application components themselves and not hardware supported in the Security Plane. Therefore, any security policies directly related to the application itself should be enforced at the appropriate component in the application. For example, the Security Plane enforces allowable frequencies for radio hardware, but the application itself should enforce what channels are available to specific users. Since the waveform components are isolated in separate environments from any end-user applications in our architecture, a compromised or malicious user application cannot bypass the policy enforcement of another component in the system. However, if a waveform component were compromised, an attacker could use it to bypass policies specific to that component, but not policies enforced elsewhere in the system.

5.6 Layered Defenses

While we did not mention it in the Security Plane section, one of the keys to this architecture is the defense-in-depth or layered approach. The goal of the architecture is to split a waveform into distinct components or groups of components so that exploits cannot affect the entire system. Some components within a system can be higher risk than others; for example, the higher layer protocol processing components are much higher risk than the physical layer since they process bytes rather than samples. In this case, it might be beneficial to use the layered approach to isolating these components and nesting one isolation mechanism within another. This essentially serves as another isolation that an attacker must escape an overcome to compromise more of the system. Figure 5.3 shows an example of this layered isolation.

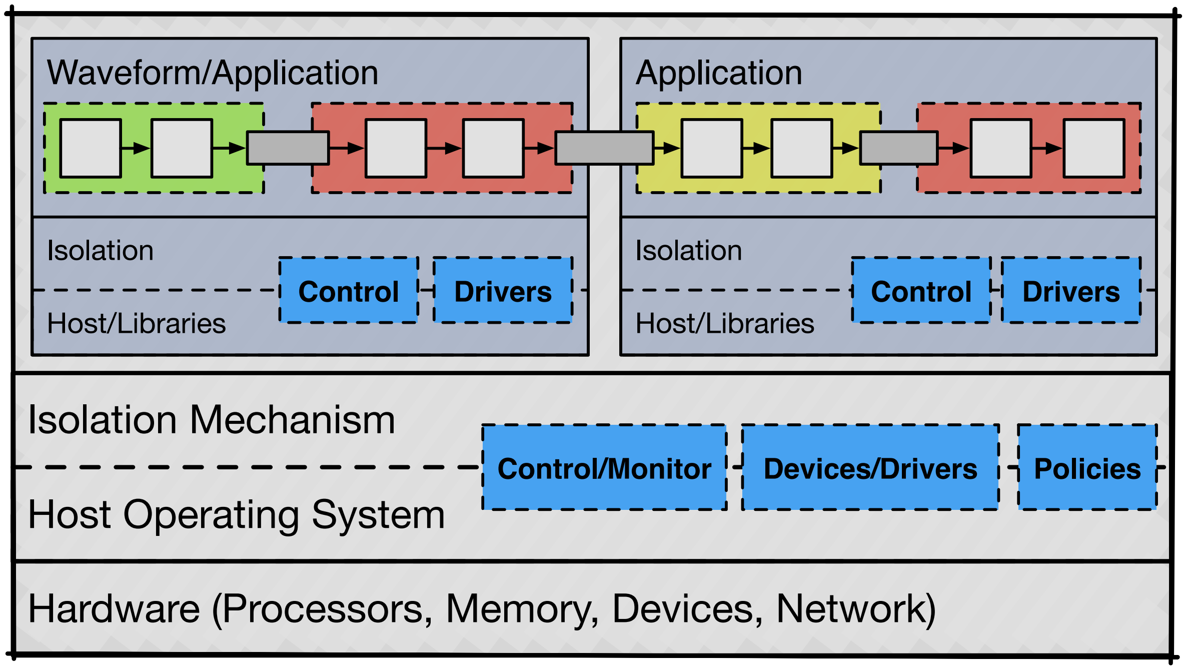


Figure 5.3: Defense-in-depth SDR with nested isolation. This is an example of the multi-layered defense-in-depth SDR architecture. Here, there are nested layers of isolation added into the main isolation environments of the system. For example, containers could be nested within virtual machine environments to implement the defense-in-depth approach.

5.7 Challenges

Distributed

Some of the main challenges that exist with implementing radios based on the defense-in-depth methodology is dealing with the now distributed nature of the waveform. This complicates how the system is interconnected and controlled versus a more monolithic approach to system design. Resource management is also a major component since the isolation mechanisms will add additional overhead and reduce the performance of the overall system.

Security Holes

Another main challenge to consider with our mechanism is the security of the isolation mechanisms themselves. We are relying on these mechanisms to isolate the zones from the secure platform. So if there is a vulnerability in the mechanism itself, this can be used to compromise the platform and gain access to the rest of the system.

For example, there have been many example exploits that show the ability of malicious code in a guest virtual machine to exploit its host hypervisor and install a root-kit in the host. Unfortunately, it can be very difficult to completely secure complex systems like a hypervisor, so there is a possibility of an unknown vulnerability existing. The goal of our approach is to reduce the size of the platform as much as possible in order to reduce the likelihood of an existing vulnerability.

In addition, access control implementations like SELinux or other techniques like Linux's *seccomp* feature which limits a process's ability to execute system calls. Techniques such as this could be used in the platform layer in order to better secure the zones and prevent an attacker from using an exploit to escape from the isolation.

5.8 Summary

In this chapter, we presented a new generic defense-in-depth architecture that provides a foundation for developing secure software radio systems. Because wireless systems are shifting towards using SDRs which could contain exploitable vulnerabilities, frameworks must be designed with this threat in mind and account for vulnerabilities that can exist in the implementation itself. As applications grow more complex, there is a greater chance of this type of vulnerability existing. Unlike previous security models for SDRs, our architecture specifically focuses on protecting the implementation itself from compromise.

Our architecture employs isolation mechanisms to separate components in the waveform into different domains. This ensures that exploits against a specific component in a waveform can be contained without the entire system being compromised. The key component of the architecture is the security plane, which provides the minimal set of software required to build the application including the isolation mechanism, device drivers, and the inter-process communication mechanism for connecting blocks within different isolated zones of the waveform.

We also briefly detail how policy management and enforcement would be handled within our architecture. Unlike previous models, like the SRM, where the user application has a single interface to the virtual radio, the radio is now divided into a distributed architecture. This requires that policy enforcement is also implemented in a distributed manner, where the isolation mechanisms properly enforce resource limitations and the device drivers implement policy enforcement for each specific device. Using the split driver model, any control commands sent to the proxy driver within the isolated environment would first be checked against the available policies in the security plane driver before the command would actually be sent to the device itself.

Chapter 6

Performance Analysis

One of the challenges in developing any type of software architecture is understanding what trade-offs need to be considered in the system design and determining the best route forward based on defined application requirements. Our proposed security architecture relies heavily on isolation environments (such as virtualization) to provide additional security, but this increases the overhead of the entire software stack and can affect overall system performance. Performance is critical to software radio applications, so this security versus performance trade-off must be taken into consideration when developing a full system architecture. Some performance loss may be acceptable for high-risk components in order to gain the additional security provided by the isolation, but not all software radio applications can tolerate the loss in overall performance. The overhead from these isolation environments needs to be well understood when developing a SDR application using the defense-in-depth architecture.

This chapter focuses on characterizing the expected performance overhead added to waveforms using our defense-in-depth approach presented in the previous chapter. We introduce a testing framework that is designed to simplify the process of measuring overhead from isolation for different waveforms over a range of system configurations and isolation environments. Executing multiple example SDR applications waveforms using different frameworks can help determine what an expected overhead might be for other SDR waveforms. However, it is important to note that each waveform and implementation will have unique overheads due to unique implementations and configurations.

In this chapter, we first present our motivation for using a waveform's maximum throughput as the primary metric for characterizing isolation overhead (Section 6.1). Next, Section 6.2 introduces a testing framework that was specifically developed to automate testing waveforms in different environments and system configurations. Section 6.3 gives an overview of the different tests and waveforms that were executed to compare the overhead between environments. In Section 6.4, we present examples of several challenges that arose that can affect the overall throughput of a flowgraph and must be accounted for in our testing framework. Section 6.5 presents the testing results and shows the overheads that occur in different isolation environments. Finally, Sections 6.5.4 and 6.5.5 provide some overall analysis and conclusions drawn from the performance testing results.

6.1 Overhead Characterization

Characterizing the overall impact of isolation environments on software radio performance can be a complicated task. Different systems have unique processing and performance requirements due to different overall implementations, utilized software frameworks, supported wireless protocols, and physical waveform specifications (like bandwidth). The additional overhead from isolating radio components into different environments will affect each system uniquely, so it is impossible to define a single test to measure the exact the overhead for *all* SDR waveforms.

Maximum Throughput

An application's maximum performance is typically bound by the speed or capacity of some hardware component within the system. It can either become limited by the overall speed of the processors (CPU bound), the amount of available memory (memory bound), or limited by the speed of the system bus or peripherals (I/O bound). Understanding how an application is bound is important for optimizing it and achieving the maximum possible performance for the application.

In a production setting, software radios are I/O bound applications which operate at specific sampling frequencies that are driven by the settings of the RF frontend hardware. The hardware enforces a set sampling rate and either provides samples (during receive) or consumes samples (during transmit) at that fixed rate (which is defined by whatever wireless standard is implemented). The system must have sufficient computational resources available for the software to meet this sampling rate and operate properly. Resource requirements vary significantly between different systems and standards; features like the bandwidth of the signal, error correction settings, and the implemented modulations are all determining factors. More complicated and higher sample rate waveforms will require more system resources. If there are insufficient resources, the SDR waveform will not operate properly and the received or transmitted signal can become corrupted.

Software defined radios implement processing pipelines, so the overall throughput of the waveform can be heavily impacted by a single component or set of components becoming performance bound. In this situation, this block or set of blocks becomes rate limiting and the bottleneck within the pipeline; ultimately, this is the determining factor for the maximum throughput the radio can achieve. Sometimes, a single block can result in significant performance degradation for the system. There can be many reasons why an SDR application becomes CPU or I/O bound, some of which include:

1. Limited computing resources exist on a system so all components in the waveform are forced to operate at lower a throughput and cause the application to not meet the required performance. This may occur when using a non-deterministic scheduler which

implements a fair scheduling algorithm and attempts to split the limited processing time equally to all components. With a limited amount of resources, the scheduler may not properly prioritize blocks that require more resources or there are simply not enough resources to distribute.

2. One or more components consume a significant majority of the available resources on the system which reduces the resources available for remaining blocks that can no longer execute at the required rate. Unlike the first example, this occurs when there normally are sufficient resources available on the system, but one or more blocks have entered an unexpected runaway condition and are over-consuming resources.

3. One or more components are not properly designed to maximize the utilization of available system resources. For example, this can occur where a component is not multi-threaded and therefore only uses a single core on a multi-core system and leaves the overall system under-utilized. If such a block becomes constrained on the single core, it can become the bottleneck limiting all other components. A key to note is the components in this situation could be optimized to better utilize resources.

4. One or more components cannot be optimized to utilize available resources. This is somewhat similar to the previous example above with the one exception that a block may not be able to be optimized. For example, this can occur if the algorithm implemented in a waveform or block cannot be parallelized and it becomes constrained by a single core.

Normally, this software bottleneck does not occur in a production SDR because the RF front-end is the rate limiting component and the system hardware has been chosen so sufficient resources exist for the software to operate properly. Some components may not be fully optimized and still become a bottleneck in an application, but if the software can still process the required sampling rate then this is not an issue in production systems. When a waveform is resource constrained and software becomes the bottleneck for whatever reason, this marks the upper performance limit of the application and defines the maximum rate that the software is capable of processing on the given hardware. For waveforms on non-deterministic operating systems, this maximum rate could fluctuate due to other executing applications on the system. This bound can be characterized as the application's maximum throughput and defines an upper limit for sampling rates set in hardware; attempting to process faster sampling rates will result in corrupted signals as the software cannot achieve the desired rate.

While hardware selection, waveform properties, and optimization are important factors to consider when developing a SDR system, the key for understanding the performance overhead of the isolation mechanisms in our architecture is this maximum throughput for a waveform. In our tests we execute waveforms without a hardware RF front-end in order to force this performance bounding to occur. By forcing the waveform to be CPU or I/O bound, we can compare the maximum throughput of the same waveform within different environments in order to characterize the overhead that is imposed by that environment.

Trade-Offs

Understanding the maximum throughput of a waveform is key to determining what trade-offs are required when implementing a system with the defense-in-depth architecture. Each system will have a set of defined security and performance requirements which designers must consider when choosing the proper system configurations and isolation environments to use. For example, if the overall bandwidth and throughput is a primary design requirement for a system, using a highly isolated environment may not be a feasible solution if the maximum throughput of the isolated waveform drops below the required throughput due to overhead. This forces the designers to make trade-offs in terms of either security or performance when implementing a system.

For many systems, performance is the primary design requirement, so any implemented security features must allow the application to still meet defined performance requirements. If a SDR stack with isolation is still capable of meeting these minimum requirements, then implementing this isolation for the additional security benefits is an easy design decision. Measuring the maximum throughput of the application within an isolation environment without a hardware front-end determines the upper limit of software performance for that specific waveform. For production systems, any RF front-end would simply need to operate at a lower sampling rate than this determined maximum rate for the software application to operate properly. If security is the primary design requirement of the system (like high-assurance systems), the performance overhead (no matter how high) is likely acceptable in order to implement the highest isolation environments possible within the waveform.

However, if both security and performance are primary design goals and the added isolation causes the maximum throughput to drop below the required rate, then other design choices need to be considered. Options include: 1) selecting different waveform configurations to reduce the overall resource consumption, 2) choosing a lighter-weight isolation mechanism that has lower overhead, 3) increasing available resources by changing hardware platforms, or 4) attempting to optimize the radio implementation or isolation environment to best use available resources.

6.2 Testing Framework

As we mentioned before, each implemented SDR system is unique, so determining a single value that represents the overhead of isolation for all SDR applications is impossible. Unique implementation can have different overheads for each isolation environment which makes characterizing the overhead of those isolation environments difficult. However, measuring

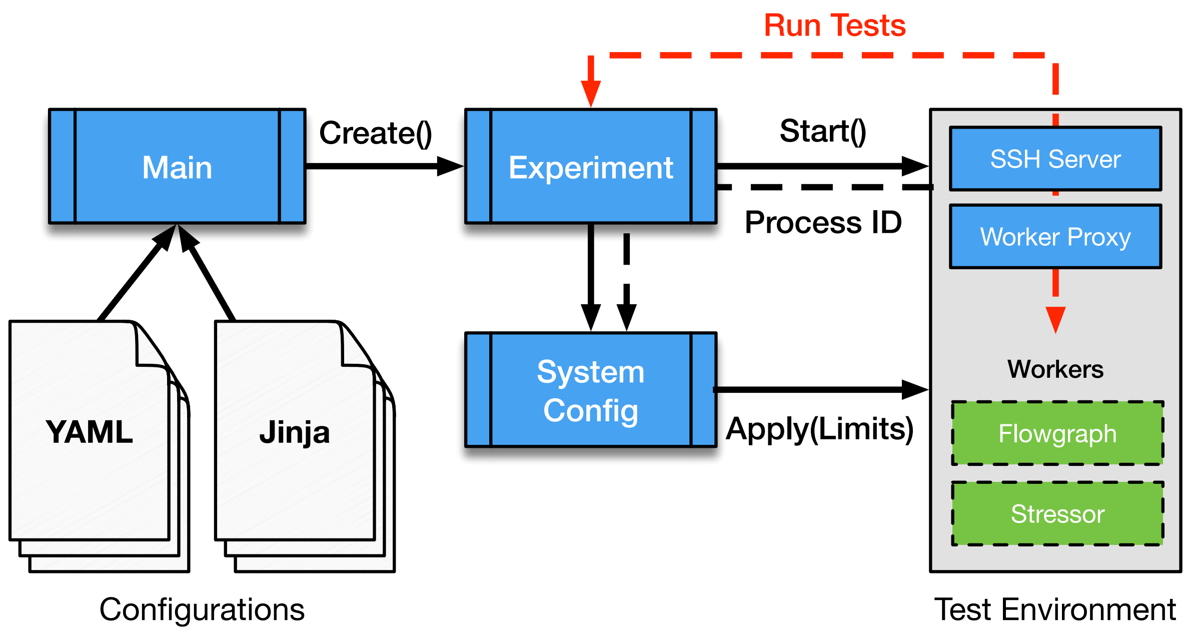


Figure 6.1: Diagram of the performance testing framework. Our testing framework was developed to execute different tests in isolated environments and measure the performance overhead of those waveforms executing in the isolation environments. The main controller is responsible for reading the input configuration and launching the appropriate testing environment for each experiment. Once the environment is loaded, the controller then applies resource limitations defined in the loaded configuration (such as processor pinning) by modifying the active host configuration. The controller then executes the worker proxy over an SSH connection and uses the running proxy to launch the individual tests that were defined in the loaded configuration.

the maximum throughput of different example waveforms and comparing the performance in different isolation environments does allow us to make some generalized predictions on the overhead expected for a particular environment.

Toward this goal, we developed a flexible testing framework designed to measure the performance of various software radio applications and other utilities over a wide range of possible system configurations. One of our main goals was creating a framework that could be easily extended to support new environments, tests, and configurations and could dynamically configure the system in order to test thousands of different configurations. For example, the framework automatically detects implemented components, so future extension of the framework is rather simple. A diagram of the testing framework is shown in Figure 6.1.

6.2.1 Components

The framework utilizes a modular architecture that consists of four major types of components: images, experiments, workers, and configurations:

Images are the pre-configured isolation environment used for testing different waveforms and utilities (for example virtual machines or containers). These are created independently of the framework and must include an SSH server, the client/worker components of the test framework, test waveforms or utilities, and any required software dependencies. For each test, the framework will launch and dynamically configure the image settings (available memory, processors, etc) based on that test's configuration.

Experiments consist of all the required code to start and configure the images for running tests. Each experiment class manages a single type of isolation environment and uses the appropriate Application Programming Interfaces (API) to start an environment and configure it with the proper network or device access for testing. The experiment classes inherit from a base class that implements all of the functionality for iterating over the set of configuration operations, connecting to the executing environment over SSH, and executing any tests or commands.

Worker classes contain the actual functions, tests, and utilities that can be executed within the running environments. The workers themselves can either directly implement the tests in Python or use the *subprocess* module to launch utilities in another process and parse the resulting output. Once an experiment class launches an environment, a worker proxy is started and is responsible for executing each worker test defined in the configuration and returning the results back to the caller.

Configurations define the how the overall tests are executed and include: 1) the different experiments and images that need to be started, 2) the system configurations and limits that should be applied for each environment, and 3) the set of worker functions to execute for each variation of the environment and system configurations. Any environment or worker that properly inherits from the framework's base classes can be specified in a configuration and launched by the framework. Multiple options can be specified for each test and environment such as the available resources (enabled processors, memory, etc), connection settings (usernames, passwords, etc), and environment settings (virtual machine name, container name, etc). Changing the overall behavior of the testing framework is as simple as providing a new configuration file that implements a different set of experiments, images, and workers to execute.

The test framework is intended to be simple to extend and add new environments, experiments, and workers. Both the experiment and worker classes are dynamically detected by the main framework controller and integrating new components into the framework is as simple as creating a new class that inherits from the appropriate parent class. Components like the experiment and worker classes are implemented in Python but are not limited to using Python for executing tests. For example, workers can execute native test binaries through the *subprocess* module.

Configuration files for the framework are written in the YAML markup language [88], but can optionally be written in the Jinja template language (with some slight modifications) [89]. Any Jinja templates are dynamically rendered at runtime to YAML and imported, which allows for more concise configuration files than pure YAML. Additional Python functionality was enabled in the Jinja parsing engine that allows these functions to be directly used within the configuration itself. One example is the *itertools.product* function, which creates a list of all possible permutations from a given set of input lists. This dynamic configuration feature is extremely useful because it simplifies the process of creating and editing configurations that can be relatively small but can generate several thousand unique configurations to test. The Jinja based configurations can also be exported as YAML configurations from the framework for easily saving and repeating the same tests and configurations in the future. An example of a Jinja configuration is shown in Figure 6.2 with a portion of the YAML config generated from this Jinja configuration shown in Figure 6.3.

6.2.2 Workflow

One critical aspect of performance testing is ensuring that all tests execute in a similar manner between different environments so any overhead due to the framework or test methods themselves are removed from the results. So, the framework executes all tests using the same method for every environment. First, the framework loads the proper experiment class and passes the list of tests to execute. For each of the tests in this loaded set, the experiment class will: 1) load and start the proper environment, 2) determine the process id (PID) of the executing environment, and 3) use this PID to apply any resource limits defined in the configuration to the executing environment. Next, the controller connects over SSH to the running environment and executes the worker proxy process for managing the actual tests.

This worker proxy is a Python script implementing remote procedure calls (RPC) using the Python remote objects (*Pyro4*) library to expose the workers themselves [90]. This proxy is used instead of SSH to simply the process of launching tests in the isolation environment. The RPC mechanism also simplifies handling results because each test returns a dictionary object containing the test results. Test results are all collated by the controller and saved to a single YAML output file. This workflow is used for every environment, so the process is consistent in order to remove any variability due to the framework itself.

6.2.3 Test Configurations

There are three experiments currently implemented in the framework; these include two isolation environments with virtual machines and containers, and the native host environment which acts as the baseline control for test performance. Several workers are also currently

```
<# Experiment Configs #>
<% set experiments = ['native', 'docker', 'virtualbox'] %>
<% set flowgraph = 'gfsk_loopback' %>
<% set samples = int(10|mega) %>
<% set repeat = 10 %>
<# CPU Configs #>
<% set cpus = list(Range.count(1, 4)) %>
<% set cpu_frequency = int(2400|kilo) %>
<% set modes = ['pinned', 'disabled'] %>

<% for experiment in experiments %>
---
experiment: <<experiment>>
source:
  - '/home/shared/env/setup_env.sh'
<# Worker settings #>
worker.dir: '/home/shared/ferret'
worker.endpoint: 'tcp://0.0.0.0:8080'
<# Forwarding rules #>
forward.ssh: 18022
<# Native settings #>
native.ssh.port: 22
<# Images #>
docker.container: ubuntu-test
docker.image: ubuntu
virtualbox.vm: ubuntu
<# Tests #>
tests:
  <% for ii, (cc, md) in enumerate(itertools.product(cpus, modes)) %>
  test<<ii>>:
    config: {cpu: {count: <<cc>>, limit: 100, frequency: <<cpu_frequency>>},
      hyperthreading: False, mode: <<md>>}
    workers:
      throughput:
        - run: {flowgraph: <<flowgraph>>, samples: <<samples>>}
  <% endfor %>
repeat: <<repeat>>
results: results/flowgraphs/gfsk-default/<<experiment>>
...
<% endfor %>
```

Figure 6.2: Example Jinja test configuration. This configuration would be dynamically rendered at runtime to produce the YAML configuration specifying all tests to execute.

implemented, including: two SDR frameworks and a stress testing utility, as well as other workers for testing framework functionality. This includes a GNU Radio worker capable of dynamically detecting, loading, and executing different flowgraphs, and a LiquidDSP worker

```
experiment: native
source: [/home/shared/env/setup_env.sh]
worker.dir: /home/shared/ferret
worker.endpoint: tcp://0.0.0.0:8080
forward.ssh: 18022
native.ssh.port: 22
docker.container: ubuntu-test
docker.image: ubuntu
virtualbox.vm: ubuntu
tests:
 test0:
   config:
     cpu: {count: 1, limit: 100, frequency: 2400000, profile: performance}
     hyperthreading: true, mode: pinned
   workers:
     throughput:
       - run: {flowgraph: gfsk_loopback, samples: 10000000}
 test1:
   config:
     cpu: {count: 1, limit: 100, frequency: 2400000, profile: powersave}
    hyperthreading: true, mode: pinned
   workers:
     throughput:
       - run: {flowgraph: gfsk_loopback, samples: 10000000}
...
 test31:
   config:
     cpu: {count: 4, limit: 100, frequency: 2400000, profile: powersave}
     hyperthreading: false, mode: disabled
   workers:
     throughput:
       - run: {flowgraph: gfsk_loopback, samples: 10000000}
repeat: 10
results: results/flowgraphs/gfsk-default/virtualbox
```

Figure 6.3: Example of an auto-generated YAML test configuration. This shows a portion of the YAML configuration that was auto-generated from the Jinja configuration shown in Figure 6.2

for launching a few example applications. The *stress-ng* worker can launch multiple stress tests targeting different components of the system such as the processor and memory.

For our testing, we mainly focused on two common types of isolation techniques: virtual machines (Oracle VirtualBox [14]) and containers (Docker [15]). One major goal of testing was keeping the environment configuration consistent between different environments, so the same software stacks and versions (Ubuntu 16.04, GNU Radio 3.7.13.4, stress-ng 0.05.23, etc.) as the host were used through each environment. When the framework launches individual tests, it modifies the environment's path variables to ensure the correct software stack within the environment is executed for each test.

Our framework can also dynamically configure the host configuration and enable/disable processor cores, set the processor frequency, and set an environment's core affinity before executing any performance tests. The rationales for these features are discussed in Section 6.4.

Each environment was tested over a range of 1 to 4 enabled processor cores, where the enabled cores were either hyperthreaded or non-hyperthreaded cores. For the results presented in this work, only independent physical cores (no sibling or hyper-threaded cores) were enabled for these tests. Also, a fixed host processor frequency of 2.40 GHz was used (based on an Intel Core i7-4770K CPU with a base frequency of 3.50GHz and max frequency of 3.90GHz) so results were not skewed by frequency scaling due to load or temperature. Other than modifying the active cores and configuration of the host, each environment itself was tested with the default settings.

6.3 Test Waveforms

For our tests, we implemented several example flowgraphs and applications that are designed to stress different parts of the system to understand the overhead of added isolation. Our main goal was stressing the system and force bottlenecks to occur to measure the maximum throughput of the application. Specifically, we focused on testing two main aspects of the system: the memory and CPU performance. While these flowgraphs are only examples and are not particularly useful for any production system, they were designed to have similar functionality to real-world systems. In this section, we briefly describe the different waveforms and tests implemented for characterizing the overhead of isolation.

6.3.1 GNU Radio Flowgraphs

The tests flowgraphs implemented in the GNU Radio framework were specifically designed to either stress the memory performance of the framework (*Null_Test* and *Bytes_Loopback*) or stressing the CPU performance through modulating and demodulating a signal (*GFSK_Loopback* and *GMSK_Loopback*) .

The *Null_Test* flowgraph (shown in Figure 6.4) does not include any signal processing functionality and simply connects the *null* source and sink blocks together to test the memory bandwidths of buffers within the flowgraph. A *head* block exists between the source and sink which will stop the flowgraph's execution once a set number of samples is processed. Since there is no signal processing in this flowgraph, the only operations performed are memory copies between two buffers and updates to the buffer's pointers (*null* source to *head* and

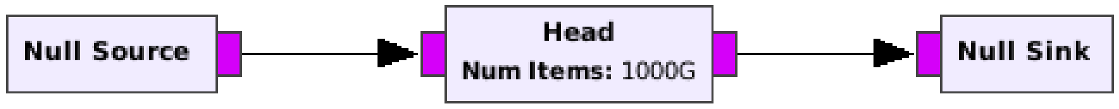


Figure 6.4: Null_Test flowgraph - This flowgraph was used for characterizing the memory performance of the GNU Radio framework in different isolation environments. A variable number of copy blocks could be added in the flowgraph prior to the head block to simulate longer flowgraphs.

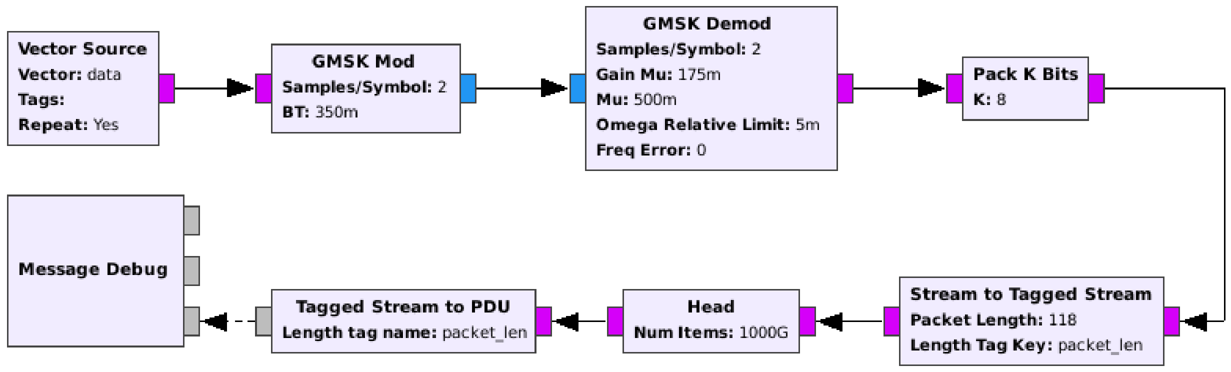


Figure 6.5: GMSK_Loopback test flowgraph - This flowgraph was used for characterizing the generic signal processing performance of a GNU Radio framework in different isolation environments. This flowgraph is structurally similar to both the GFSK_Loopback and Bytes_Loopback flowgraphs that were used for many tests.

head to *null* sink). By default, there are three blocks in this flowgraph, but it can be configured to have any number of additional *copy* blocks that exist between the *null* source and *head* blocks to simulate longer flowgraphs. This flowgraph quickly becomes I/O bound and demonstrates the upper bound of buffer performance within a GNU Radio flowgraph.

The *Bytes_Loopback*, *GMSK_Loopback*, and *GFSK_Loopback* flowgraphs are very similar in structure and implement the same basic data flow. Random or known bytes are generated, passed through a modulation and demodulation stage, and then processed into GNU Radio messages. An example of the *GMSK_Loopback* flowgraph is shown in Figure 6.5. The only difference between these flowgraphs is the modulation blocks. The *Bytes_Loopback* does not implement a modulation/demodulation stage and instead implements a memory copy operations in place of these blocks; all of the other byte handling operations remain the same. The *GFSK_Loopback* uses a slightly different modulation scheme but overall remains the same.

These flowgraphs are intended to mimic a very simple and generic waveform that includes both transmit (modulation) and receive (demodulation) components within the waveform. The modulation and demodulation blocks in the *GFSK_Loopback* and *GMSK_Loopback* flowgraphs cause this flowgraph to be predominately CPU bound. Since the *Bytes_Loopback* flowgraph lacks this functionality it is predominately I/O bound and demonstrates the upper performance bound for similar flowgraphs. Because the other loopback flowgraphs are mainly CPU bound, they typically have significantly lower throughput than the *Bytes_Loopback* flowgraph.

Like the *Null_Test* flowgraph, each of the loopback flowgraphs contains a *head* block which stops flowgraph execution once a certain number of samples has been processed. This block is required for the framework to properly measure the flowgraph's overall throughput. When executing a test, the framework loads the proper flowgraph, sets the sample length for the *head* block, and measures the overall execution time of the flowgraph. Once the flowgraph completes, the elapsed time is used to determine the average throughput of the flowgraph (measured in bytes/second for the loopback flowgraphs and samples/second for the *Null_Test* flowgraph).

6.3.2 LiquidDSP Waveforms

In addition to the GNU Radio flowgraphs, we also developed several other example applications that utilized the LiquidDSP framework. These flowgraphs were based on example benchmark applications already included in the framework, and specifically tested two common types of components in SDR implementations: a filter component and the *FlexFrame* components.

The filter benchmark tested the Finite Impulse Response (FIR) filter component in LiquidDSP which is commonly implemented in waveforms to remove unwanted components in received signals. During the initialization, a buffer is populated with random samples which are then processed by the filter and saved to a secondary buffer. These processed samples then become the input for the next filtering operation. This loop continues until a timeout occurs, after which the total number of processed samples is calculated to determine the final throughput of the test.

The *FlexFrame* component provides a basic framing structure useful for implementing the data link layer of a wireless protocol. It provides a flexible protocol and allows users to change many settings such as the payload modulation schemes and the forward error-correction (FEC) schemes. This includes both the generator objects responsible for creating the frames and the synchronizer object that is implemented at the receiver. For these benchmarks, the generator object first creates the frame and the resulting samples are saved to a buffer which is then passed to the synchronizer object to processes the frame. After a set time has elapsed, the total number of bytes processed is calculated to generate the overall throughput (or bitrate) of the test application. This benchmark can optionally be set to only process the synchronization stage of the application. In this case, a single frame is generated during startup and it is repeatedly processed by the synchronizer.

All of the benchmark applications were multi-threaded, but not all of the tests included synchronization between the different threads. This is different than the GNU Radio flowgraphs which do include a significant amount of synchronization between the different threads executing in the flowgraph. For the tests without synchronization, each thread operated independently and attempted to process as many samples or frames as possible.

Two of the *FlexFrame* tests did include synchronization. A single thread would be responsible for generating frames and producing samples to a buffer. A pointer to this memory buffer would be passed to one of multiple threads handling the receive processing. By adding synchronization between threads, these tests could better simulate how an actual waveform implementation would process incoming samples. In one case, each receiver thread owned a lock that needed to be acquired by the transmitter thread before a buffer pointer could be passed to the receiver thread. The transmitter thread would first generate samples and then loop through the receiver threads looking for a waiting receiver. For the other test, a linked list tracking the generated frames was shared between all threads. The transmitter thread generated samples and added the corresponding buffer pointers to the list (unless it was already full); receiver threads would then pull the buffer pointers from the list and process the samples in the buffer.

6.3.3 Stress-ng Tests

In addition to the GNU Radio and LiquidDSP waveform tests, we included multiple tests using the *stress-ng* utility that stressed specific system components [91, 92]. By stressing individual components such as the processor, kernel, or memory, these tests can help identify what components may become a bottleneck in the application while running in different isolation environments. Also, they help define a baseline for the minimum overhead that could be achieved for an application. The *stress-ng* utility launches multiple workers per test that all execute in independent threads without coordination, so this provides a benchmark of the best case scenario in terms of environment overhead. This differs from the GNU Radio flowgraphs where all threads are dependent on the execution of other threads in the system. The stressors demonstrate the minimum overhead that could be expected when using containers or virtual machines for isolating a software radio waveform. All SDR waveforms would require some type of coordinate between threads, so the percent overhead for a waveform is typically greater than the overhead for these stressors due to synchronization between the threads.

Out of the numerous stressors that exist in *stress-ng*, we specifically focused on stressing the processor, memory, and kernel:

- **matrix** - Performs a mix of floating point, cache, and memory operations which mainly stress the system CPU.
- **bsearch** - Performs a binary search of an array of integers to stress both the processor, cache, and memory access.
- **hsearch** - Similar to bsearch but performs a search on a hash table rather than a binary search.
- **stream** - Stresses memory by allocating memory buffers 4+ times larger than the CPU cache and performing multiple floating point operations (copy, scale, add, triad) between the buffers.
- **vm** - Stresses the memory bandwidth of the system by continuously allocating and deallocating memory buffers and performing read/write memory operations on the buffers.
- **switch** - Uses pipes to send messages between processes and forces context switches to occur to stress kernel operation.
- **context** - Forces rapid context switching to occur between different threads to stress kernel operation.

6.3.4 Split Flowgraphs

For all of the tests described above, the test waveform or application was implemented as a single monolithic process. While these tests are useful for characterizing the general overhead for entire applications in isolated environments, they are not characteristic of waveforms using a defense-in-depth architecture. The main goal of our defense-in-depth architecture is segmenting an application into multiple isolated zones rather than a monolithic implementation.

The *Null_Test* and *GFSK_Loopback* flowgraphs were split into two separate flowgraphs (a transmitter and receiver flowgraph) for measuring the overall throughput and overhead for the split configuration. The two sub-flowgraphs were connected using various Inter-Process Communication (IPC) mechanisms including: networking (using TCP), Unix domain sockets, and named pipes. Three different configurations were tested: 1) each flowgraph was created and started as a completely different process, 2) a single Python process launched the flowgraphs through the multiprocessing library module, and 3) each flowgraph was separately started in independent containers. TCP sockets provide a familiar network-stack based approach to connecting components on a local host. Domain sockets use a similar API as standard network sockets in POSIX kernels, however, all operations are directly handled in the kernel rather than the network stack which provides a lower overhead alternative to full TCP sockets [93] Domain sockets provide a better overall solution for passing data between processes in the same host in a network like fashion. Named pipes are another IPC method that uses a filesystem interface rather than the network stack; special files are created in the filesystem which multiple processes can open for reading or writing. When data is written to the named pipe, the kernel directly passes this data between the communicating processes without writing to the actual filesystem [94].

6.4 Testing Challenges

Throughout the development and testing process, we identified several system features and configurations which can cause variances in the performance results of the SDR applications. The system's hardware configuration, software components, and flowgraph configurations can all be contributing factors. This emphasized the importance of accounting for these factors and using a consistent workflow for executing tests within our framework. Addressing these issues when testing different environments and configurations and maintaining a consistent workflow throughout testing helps significantly reduce the overall variance due to these factors. A few examples of how different configurations can affect test results are presented below:

6.4.1 Frequency Scaling

Modern processor architectures implement features such as turbo boost and frequency scaling [95, 96] which allow the processor's operating frequency to be dynamically changed based on the current system workload and temperature. During periods where the processor is idle or there are minimal operations, the overall frequency is reduced to conserve power (and improve battery life for mobile systems). As the workload on the system increases, the processor frequency can be dynamically increased in order to handle the additional work. However, the processor frequency can also be reduced if the overall system temperature becomes too great in order to prevent damage to the processor.

While these are useful features for many cases, this scaling can significantly affect the test results when performance profiling applications. Over multiple consecutive tests, the CPU's frequency can be significantly reduced as the temperature of the system increases, resulting in significant performance loss for test iterations occurring later during overall testing. This can be especially true for tests that are specifically designed to stress the CPU performance. Figure 6.6 shows the effect of CPU scaling on the throughput performance of multiple consecutive iterations of the *GFSK_Loopback* flowgraph. Table 6.1 compares the average

Table 6.1: Flowgraph performance with CPU frequency scaling enabled and disabled. This table shows the performance comparison of the *GFSK_Loopback* flowgraph when the CPU's frequency scaling features are both enabled and disabled. With frequency scaling enabled, later iterations show a 20+% loss in performance, while the tests with the CPU frequency fixed showed no loss in performance relative to earlier tests. A plot of these results is shown in Figure 6.6.

Mode	**Index**	**Average (MBps)**	**Deviation**	**Variance**	**Difference**
Scaled	0-4	2.90	0.10	0.01	-
	55-59	2.29	0.13	0.02	21.03%
	All	2.37	0.34	0.11	18.28%
Fixed	0-4	1.83	0.01	0.00	-
	55-59	1.83	0.01	0.00	0.00%
	All	1.83	0.01	0.00	0.00%

throughput of the *GFSK_Loopback* flowgraph at different points of the overall experiment.

These results show the fixed frequency tests have consistent throughput for the entire experiment, while the scaled results show a 21% change in performance for later iterations of the test. This is a significant performance loss and shows how SDR performance results can be heavily distorted over time by frequency scaling due to heavier workloads and system temperatures. Because of this possible variance in the testing results because frequency scaling, our framework accounts for this by using fixed CPU frequencies for all tests.

6.4.2 Hyperthreading

Another feature implemented in modern desktop and server processors is Hyperthreading which helps improve the performance of the overall system [97, 98]. Effectively, a single physical core in the processor is treated as two logical cores with independent hardware threads. Each hardware thread has an independent execution pipeline, but they share some of the main execution resources of the core. If one of the hardware threads becomes stalled waiting for an operation or some external processing to complete, the processor can switch to the second hardware thread and remain active as long as the resources needed for the second core are still available to use. Without hyperthreading, the entire processor core would remain stalled waiting for the operation to complete. Hyperthreading helps boost the overall system performance by keeping the processor active even when one hardware thread is stalled waiting for an operation. Operating systems supporting hyperthreading will create two logical cores for each of the available physical cores and can dynamically

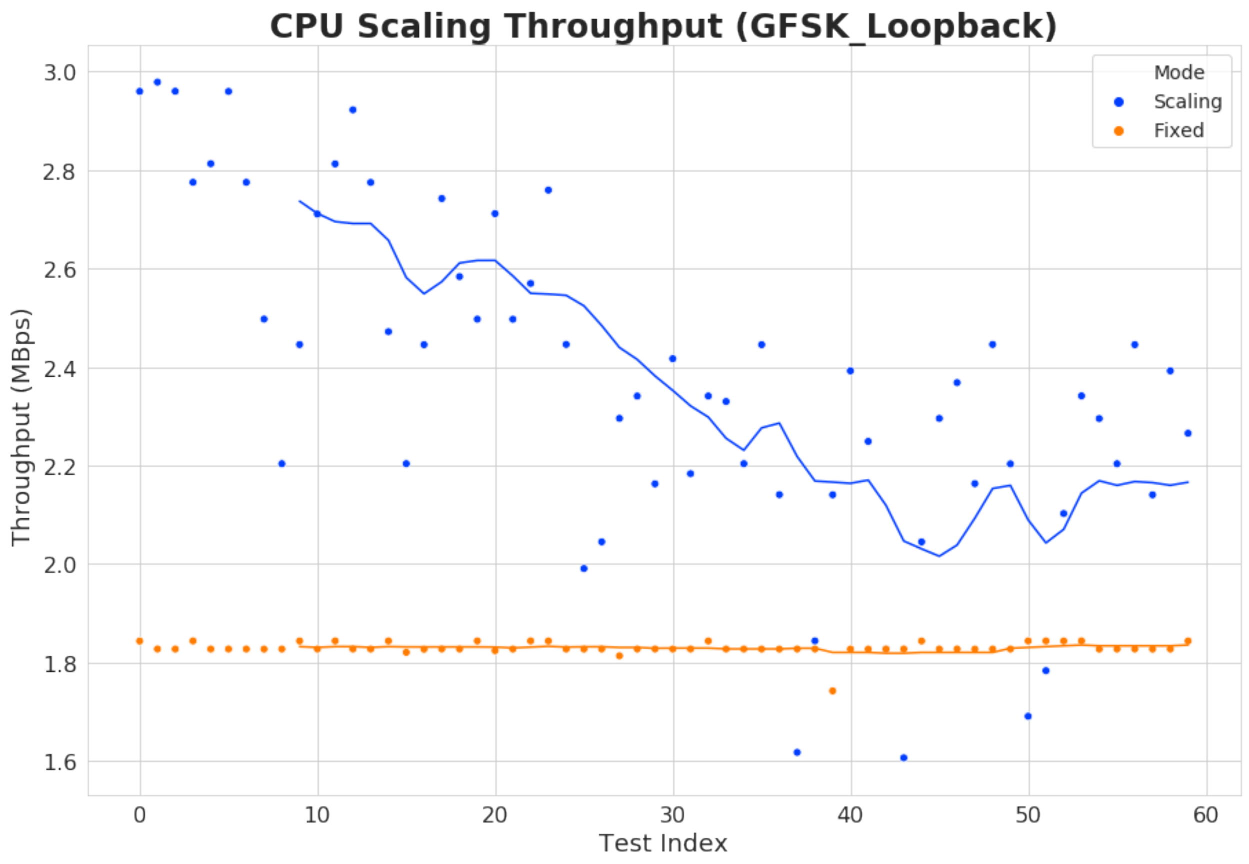


Figure 6.6: Flowgraph performance with CPU frequency scaling enabled and disabled. This plot shows the difference of executing the *GFSK_Loopback* flowgraph with the CPU frequency scaling enabled and disabled. It shows the drop in overall throughput due to frequency scaling as tests are repeated for the entire experiment. However, with scaling disabled and the CPU frequency fixed, the tests executed consistently over the entire experiment. The plotted lines show the moving average for throughput (for a window of 10 tests). Some average throughputs for these tests are shown in Table 6.1.

schedule processes to run on any of the logical cores.

However, while this does improve overall system performance, the independent hardware threads do not provide the same boost as a true, independent, physical core since they still share some resources. If software threads are scheduled to logical cores that are shared on a single physical core and require the same execution resources, then the performance gain of hyperthreading will be minimal since the threads would still compete for resources. An operating system kernel will typically attempt to evenly distribute processes and software threads to all of the logical cores available. If the kernel schedules threads to the shared logical

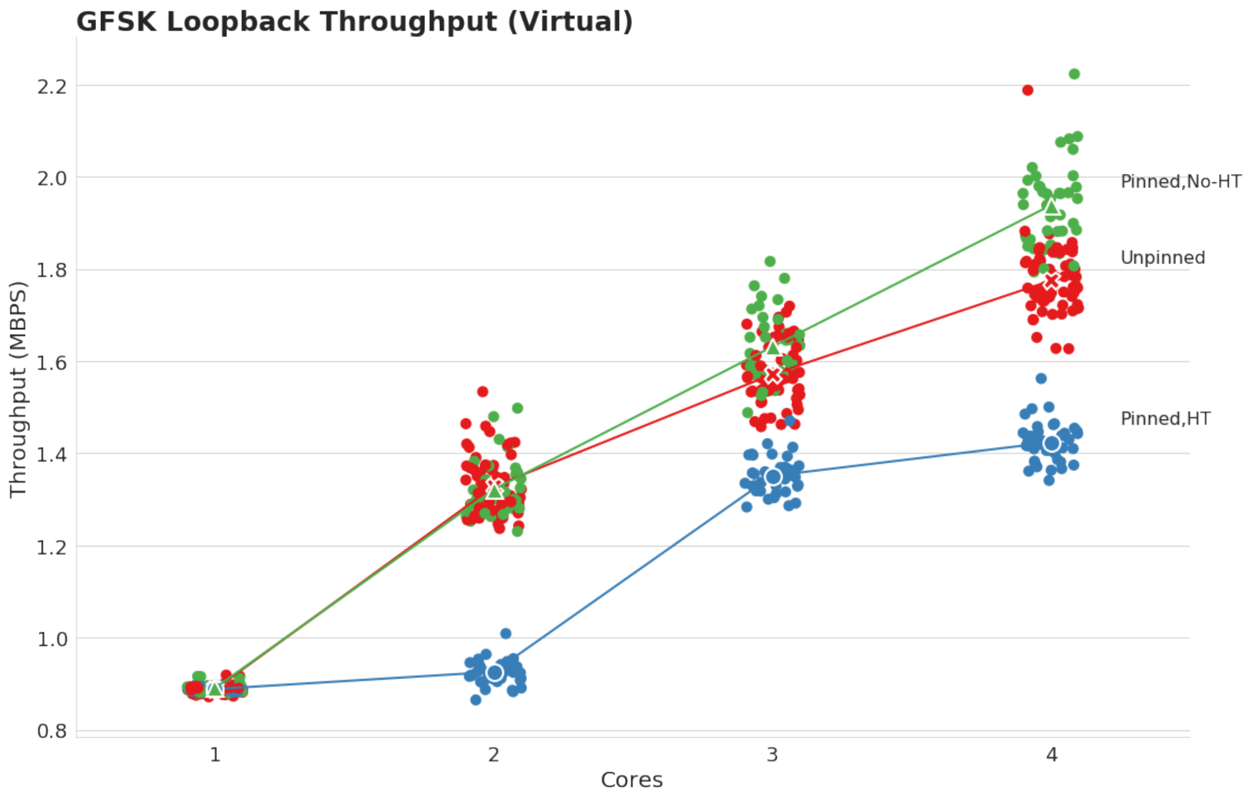


Figure 6.7: Flowgraph performance of pinned versus unpinned processes. This plot compares the throughput of the *GFSK_Loopback* flowgraph executing in a virtual machine environment with different settings for pinned cores and the number of active cores. Without pinning cores, the operating system is free to schedule the flowgraph process on any available core. The variance in the results can heavily depend on the mix of non-hyperthreaded and hyperthreaded cores selected by the kernel.

cores, the performance gain may be minimal compared to executing threads on different physical cores.

For our testing, this can add variance to the test results if the non-deterministic scheduler happens to utilize the shared logical cores where resource contention becomes an issue. Figure 6.7 shows the performance comparison of the *GFSK_Loopback* flowgraph when 1) the process is pinned to non-shared, physical cores, 2) the process is pinned to logical cores shared on physical cores, and 3) the process is unpinned. In this case, the tests were executed within a virtual machine environment which enforced the CPU core limits and process pinning to the proper hyperthreaded and non-hyperthreaded logical cores.

This test shows that the flowgraph did perform well when the process was not pinned to any cores, but explicitly pinning the flowgraph process to the non-shared logical cores results in the best overall performance. As expected, there is a slight performance increase when the flowgraph is pinned to hyperthreaded logical cores, but it is very minimal since the logical cores still share execution resources. The large jump in performance from 2 to 3 cores for the hyperthreaded test is due to adding another physical core to the configuration. Effectively, this shows the performance comparison of a flowgraph executing on a quad-core system without hyperthreading versus a dual-core system with hyperthreading.

Overall, the best flowgraph performance is seen when the flowgraph process is pinned to logical cores not shared on a physical core. Without specifically pinning processes to certain cores, there could some additional variance in the performance results when the underlying operating system uses a non-deterministic scheduler. If the scheduler happens to use non-shared logical cores for most of the execution, then the resulting performance would be similar to the non-hyperthreaded pinned results. If the shared logical cores are utilized instead, then the overall performance would mirror that of the hyperthreaded pinned results. To account for this, all tests executed within our test framework are pinned to either the hyperthreaded or non-hyperthreaded logical cores (which is selectable at runtime).

6.4.3 Sample Count

Table 6.2: Flowgraph performance comparison based on sample count. This table shows the performance comparison of the *Null_Test* flowgraph when using different sample counts for each test. The results show that tests processing less than 10^9 samples have a high standard deviation from the mean and test processing a larger number of samples have a relatively small standard deviation. A plot of these results is shown in Figure 6.8.

Environment	**Samples**	**Mean (Msps)**	**Deviation**	**Variance**
Native	$< 10^9$	737.42	393.10	154526.54
	$>= 10^9$	**1254.48**	**14.33**	**205.30**
	All	872.89	407.20	165810.64
Docker	$< 10^9$	686.72	352.15	124007.90
	$>= 10^9$	**1134.54**	**7.91**	**62.53**
	All	804.05	361.00	130323.44
VirtualBox	$< 10^9$	211.08	79.33	6293.07
	$>= 10^9$	**292.86**	**6.25**	**39.05**
	All	235.29	76.38	5834.51

Because all of our tests were executed on a non-deterministic operating system, the length of the overall test could also create variation in the throughput results. For some of the

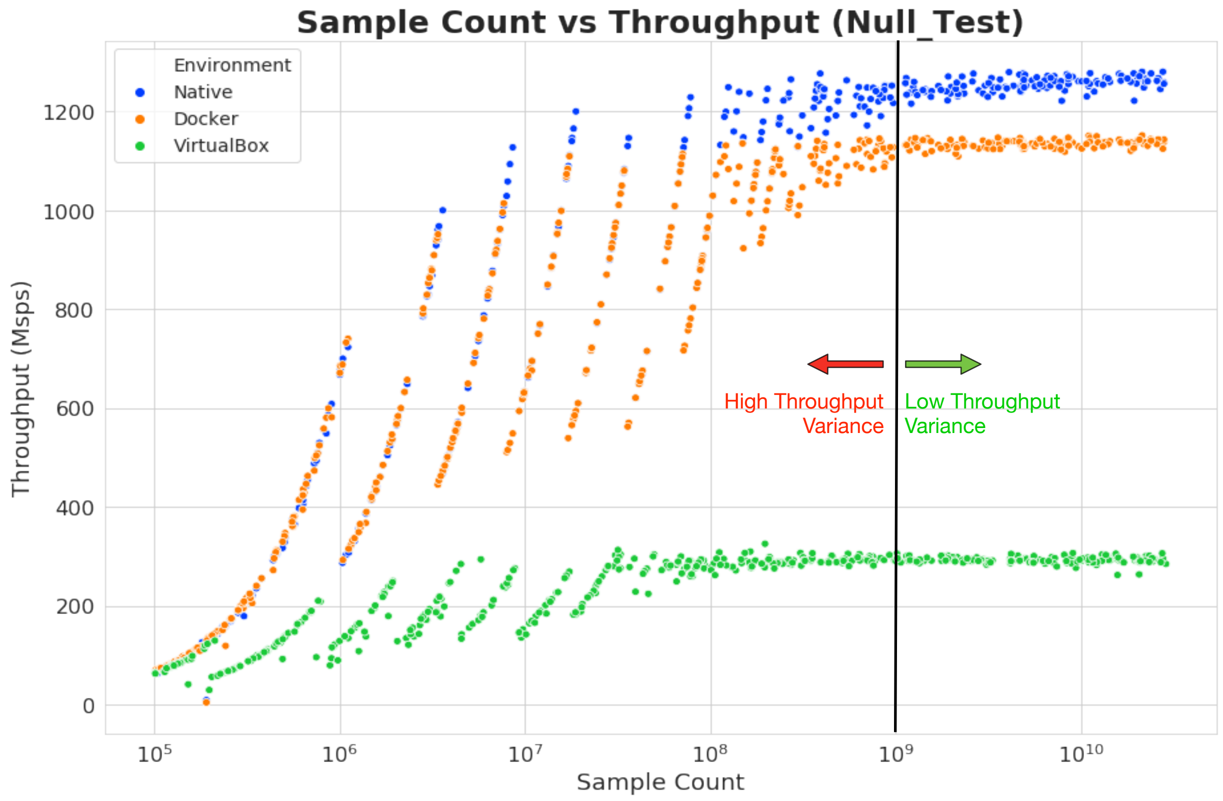


Figure 6.8: Sample count versus flowgraph performance. Our testing framework calculates the overall throughput by measuring the execution time for a flowgraph that is processing a fixed number of samples or bytes. This plot shows the resulting performance of a flowgraph using different sample counts for the *Null_Test* flowgraph. Shorter lengths result in higher variance in the measured throughputs, and longer lengths converge to the flowgraph's maximum throughput. Table 6.2 compares the results of using different sample counts.

tests, part of the application startup and shutdown processing is also included as part of the overall measured elapsed time. This presents an issue for shorter length tests, because the majority of the elapsed time may have been spent in the startup or shutdown routines rather than actually processing data and samples. The resulting throughput measurements may not be a representative value of the actual performance of the flowgraph. Additionally, the non-deterministic nature of the kernel scheduler can also add variance to the results. For shorter length tests, the measured throughput may be highly dependent on how and when the underlying operating system actually schedules and executes the flowgraph threads. The solution to remove this variance is to simply execute the flowgraph for a longer duration so that any overhead due to starting, stopping, or scheduling the flowgraph is minimal in comparison to the overall elapsed time.

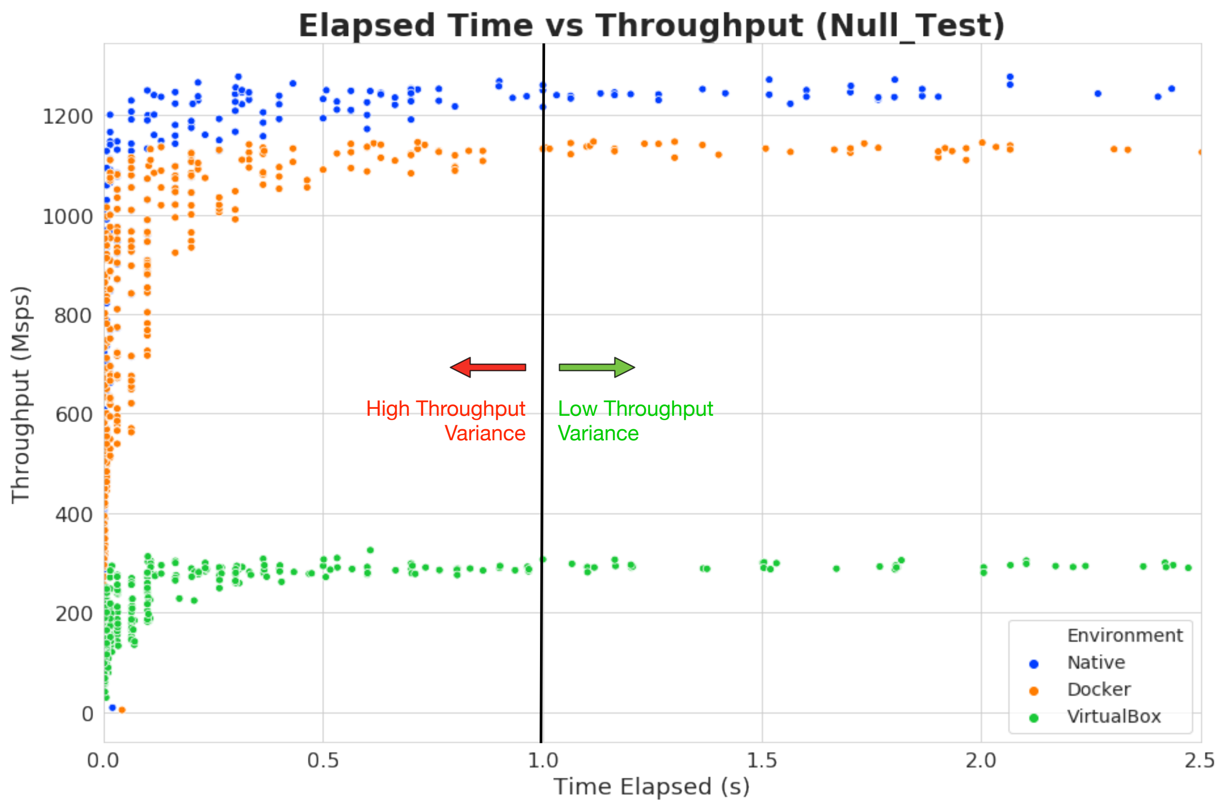


Figure 6.9: Time elapsed versus flowgraph performance. This plot is similar to Figure 6.8, but instead compares the elapsed execution time versus the overall throughput. This also shows that the very short duration tests show a significant variance in overall throughput. Executing tests longer than one second have a lower variance for the overall results. Table 6.3 shows the flowgraph performance compared to the overall elapsed execution time.

Figures 6.8, 6.9, and 6.10 all show the effect the length of a test has on the overall throughput of a flowgraph. Tables 6.2 and 6.3 also show the comparison of sample count and elapsed time on the overall performance of a flowgraph. Figure 6.8 shows the comparison of the number of processed samples versus the overall measured throughput for the *Null_Test* flowgraph. Due to the high throughput of this flowgraph, the flowgraph requires a very large (10^9) number of samples before the measured throughput begins to converge and show a small deviation in the measured values. Figure 6.9 shows a similar behavior when comparing the overall length of the test to the measured throughputs. In this case, the measured values begin to settle once the flowgraph has executed for longer than 0.5 seconds and are consistent after executing for at least 1.0 seconds. Finally, Figure 6.10, plots the relationship between the number of samples and the overall elapsed time of the flowgraph. Here, we show a reasonable range for sample counts (10^9 to 10^{10}) that result in stable results and are relatively quick

Table 6.3: Flowgraph performance comparison based on time elapsed. This table is similar to Table 6.2, but instead compares the flowgraph performance based on the overall elapsed time of a test. This shows that short duration tests (less than 1.0 second) show high variance, where the longer tests show consistent throughputs. A plot of these results is shown in Figure 6.9.

Environment	Elapsed Time (s)	Mean (Msps)	Deviation	Variance
Native	< 1.0	742.91	394.53	155655.86
	>= 1.0	**1254.64**	**14.31**	**204.83**
	All	872.89	407.20	165810.64
Docker	< 1.0	686.72	352.15	124007.90
	>= 1.0	**1134.54**	**7.91**	**62.53**
	All	804.05	361.00	130323.44
VirtualBox	< 1.0	198.37	77.75	6045.58
	>= 1.0	**293.04**	**6.23**	**38.77**
	All	235.29	76.38	5834.51

tests. When executing thousands of iterations, even a couple of additional seconds per test will add significant time to the overall experiment.

It is important to note that selecting the number of samples to execute for each test is highly dependent on the overall throughput for the flowgraph. For example, the tests using the GNU Radio flowgraphs rely on the *Head* block to determine overall test duration; once a set number of samples have been processed by the flowgraph, the *Head* block will stop the flowgraph. Different flowgraphs may have vastly different throughputs, so a reasonable sample length that executes in seconds for one flowgraph (like the *Null_Test* flowgraph) could result in ridiculous test lengths of minutes or even hours for lower throughput flowgraphs (like the *GFSK_Loopback* flowgraph). For tests with many iterations, this quickly becomes unreasonable. Additionally, if the same sample count is used for different test configurations, then the chosen length for all tests should be determined by the fastest overall configuration of the flowgraph. For example, the number of samples that results in a runtime of a second or two on a single core test configuration may only execute for a fraction of a second with four cores enabled. If the fastest configuration (or a relatively fast configuration) executes for a reasonable length of time to producing low variance results, than any lower throughput configuration should also execute for a reasonable amount of time.

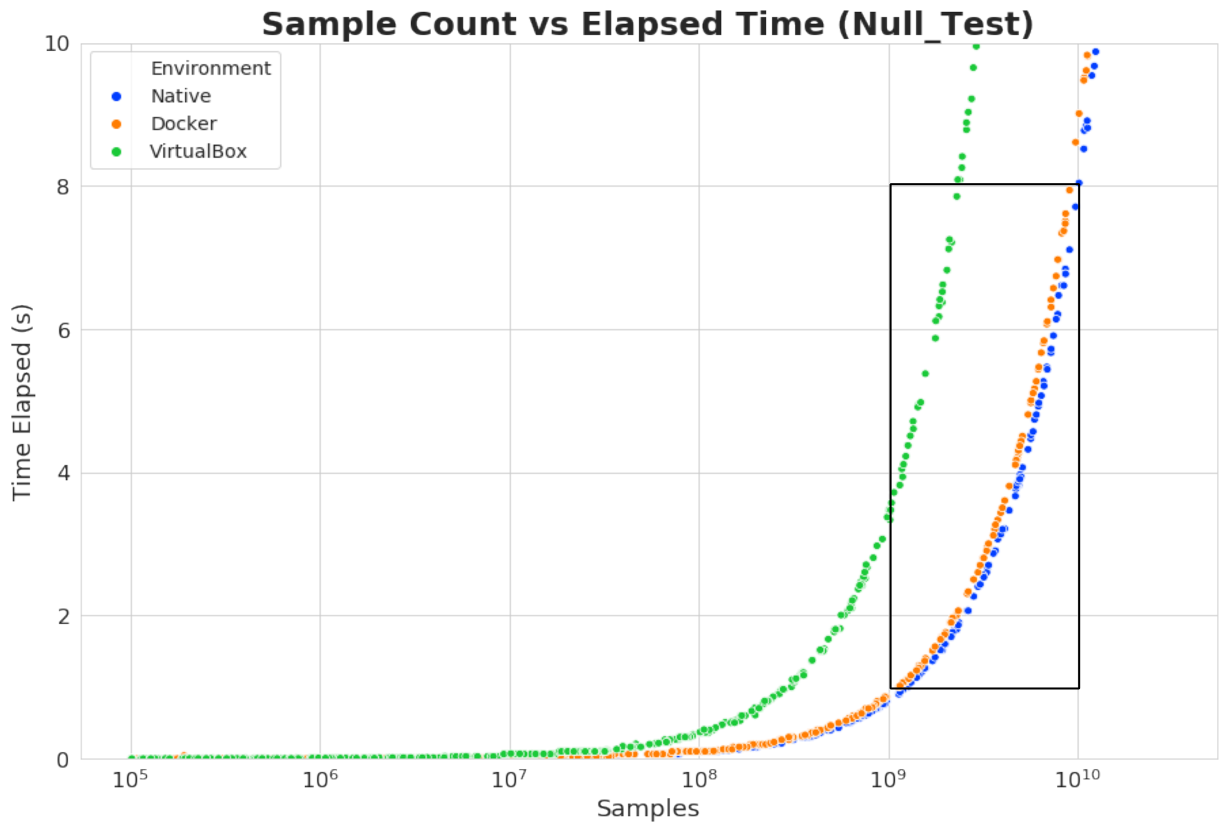


Figure 6.10: Sample count versus time elapsed. This graph is similar to Figure 6.8, but instead compares the number of processed samples with the elapsed time of the tests. Based on the results of Figures 6.9 and 6.8, the selected sample range is ideal for testing flowgraph throughput (for the *Null_Test* flowgraph).

6.4.4 VOLK Profile

The specific software configurations and implementations can also contribute to differences in the performance results of multiple tests. Within the GNU Radio framework, many of the builtin processing blocks utilize the VOLK library (Vector Optimized Library of Kernels) to accelerate signal processing [99, 100]. The VOLK library uses the vector processing capabilities of modern processors to process multiple samples in a single operation. These vector kernels use a Single-Instruction-Multiple-Data (SIMD) architecture versus the normal Multiple-Instruction-Multiple-Data architecture of the processor [101]. With the MIMD architecture, each core in the processor independently fetches instructions from memory and executes independently from the other cores. The SIMD components of the processor instead execute a single instruction or single set of instructions on multiple pieces of data at the same time. For signal processing applications, this can significantly boost the overall performance of the application.

Since modern processors can implement multiple versions of these SIMD components and instruction sets, the VOLK project provides a profiler for testing the different instruction sets and choosing the highest performing ones. Typically this generates a profile which lists the best performing components for each implemented kernel and is usually saved in the user's home directory. If the profile is not properly moved between different environments and systems, this can result in VOLK not using the same profile between tests and generating significantly different results. Figure 6.11 shows a comparison between tests using different VOLK configurations: 1) no set profile, 2) a generated profile, and 3) each kernel set to use the generic (non-SIMD) implementation. Interestingly, the highest performance was achieved without the use of a VOLK profile (Table 6.4), though the reasons why this may be the case are beyond the scope of this dissertation.

Table 6.4: Flowgraph performance comparison based on the enabled VOLK profile. This table compares the throughput performance of the *GFSK_Loopback* based on which VOLK profiles were enabled in the system. Interestingly, the highest performing tests did not use any VOLK profile. Figure 6.11 shows a plot of this data.

Profile	**Average (MBps)**	**Deviation**	**Difference**
None	1.83	0.01	-
Profiled	1.75	0.01	4.37%
Generic	1.43	0.01	21.86%

6.4.5 Randnf() Function

The specific functions that are called within a waveform can also cause unexpected results concerning the overall throughput of an application when executing in different environments. For example, some of the initial versions of our LiquidDSP throughput tests used the *randnf* function for adding noise to the signal during processing. This function was called within the main processing loop of the waveform, so it was constantly executing for each iteration of the signal processing. Figures 6.12 and 6.13 show the results of executing a waveform using this function and Table 6.5 shows the average performances when using the *randnf* function.

There are two surprising results from these tests. First, the single core application significantly outperformed the multi-core versions of the application. Each test was executed with the number of threads equal to the number of active cores on the system. The *randnf* function calls *rand()* under the hood which is not a thread-safe operation which may account

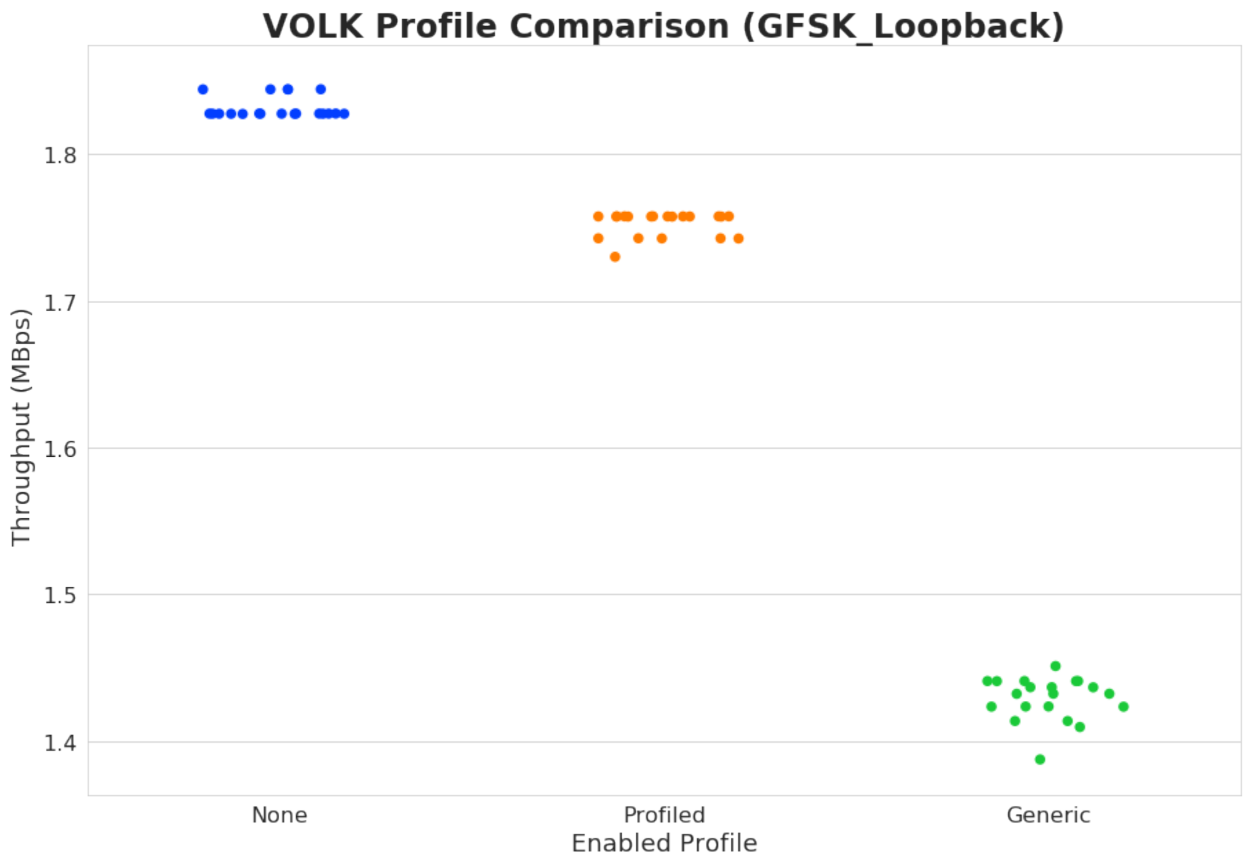


Figure 6.11: Flowgraph performance based on the enabled VOLK profile. This plot compares the throughput results of the *GFSK_Loopback* flowgraph using different VOLK configurations. The profiled configuration was generated through the *volk_profile* utility and the generic configuration forced the use of generic kernels. Interestingly, VOLK executed most efficiently when not using any configuration.

for this result. Interestingly, both multi-threaded tests on a single core and single threaded tests on a multi-core configuration reproduce the same results as the single core, single thread tests. Secondly, both the virtual machine and container environment outperform the native environment in multi-core configurations and the virtual machine environment has the highest performance overall.

This seems to indicate that, under the hood, the virtualized environments may use a different implementation for the *rand()* function that is used in the native environment (which could also be a security issue). Like the VOLK profile results, the specific reasons this occurs is out of scope of this work, but it is important to note how the software functions called within a waveform can have this type of significant effect on the overall performance results. For

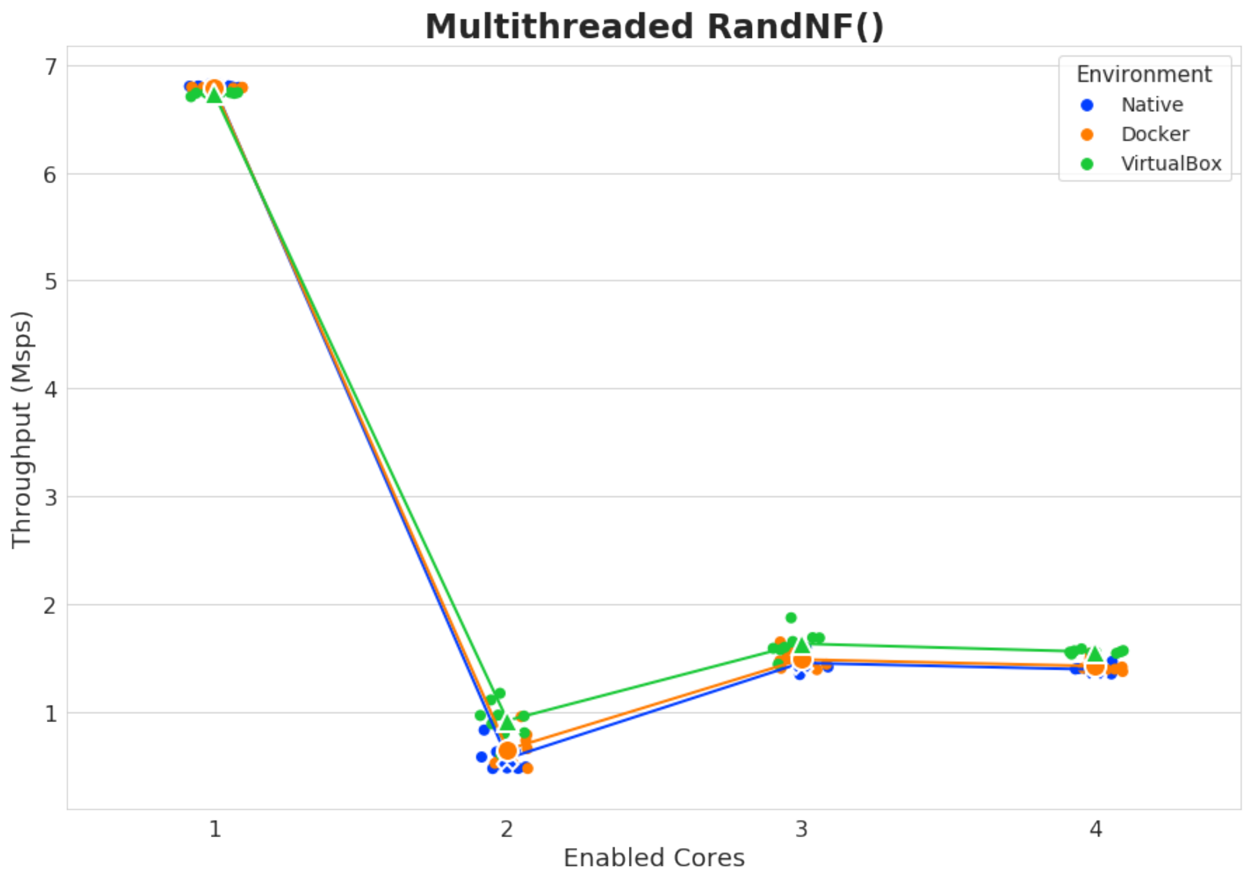


Figure 6.12: Flowgraph performance when calling *randnf* from LiquidDSP. This shows the throughput results of a LiquidDSP based waveform that is calling the *randnf* function during each iteration. Interestingly, the multi-core results show significant throughput overhead compared to the single core test, and the virtual machine environment performs the best in the multi-core environments. Figure 6.13 shows only the multi-core results of the same tests. Table 6.5 shows the average values of these results.

all of the LiquidDSP tests presented later in this chapter, the *randnf* function was either completely removed from the waveform or implemented in the initialization steps of the waveform which are not being timed.

6.4.6 Software Configurations

One final point to note is that the other configurations such as system dependencies or the software versions could also affect the throughput results of a flowgraph. The active VOLK configuration and *randnf* function are only two examples of this. If the versions of the

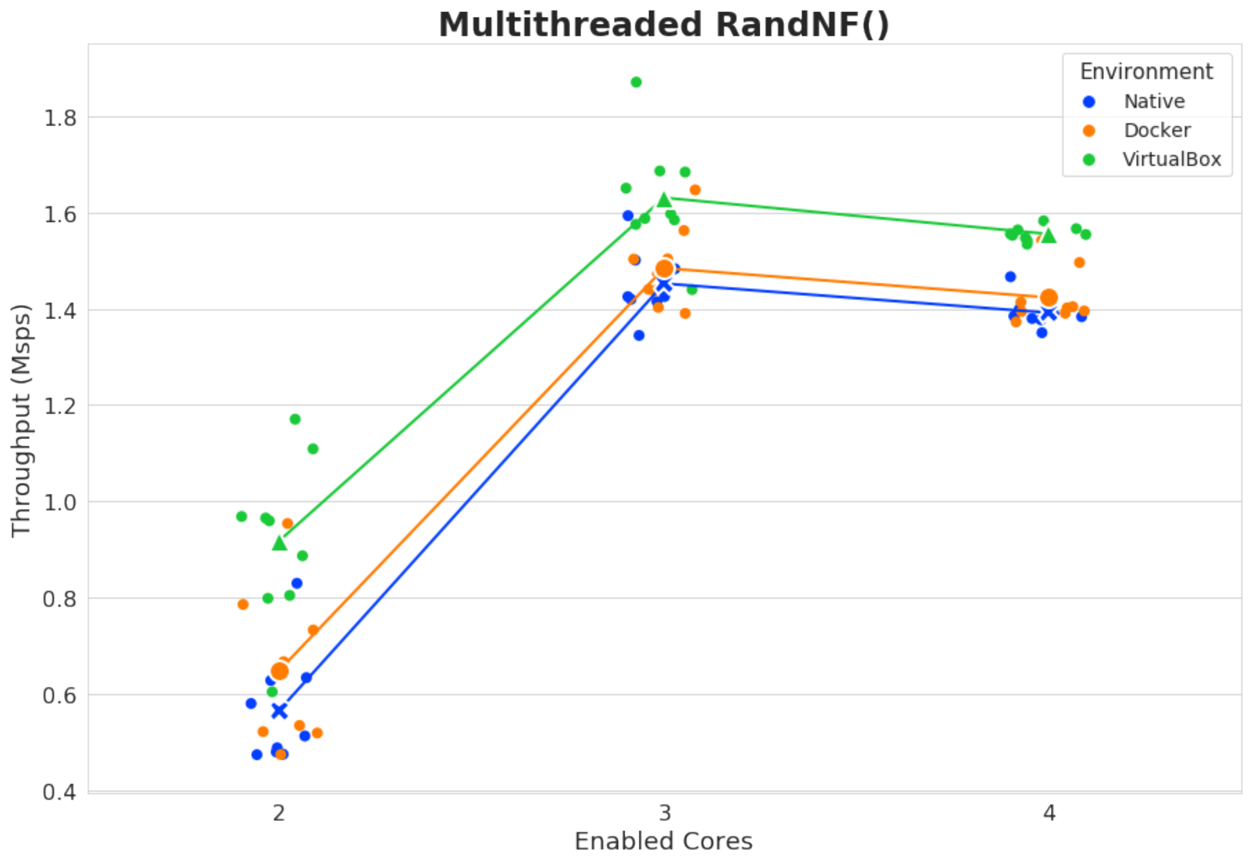


Figure 6.13: Flowgraph performance when calling *randnf* from LiquidDSP (multi-core). This is the same test as is shown in Figure 6.12, but only shows results from the multi-core configurations. This shows that both the virtual machine and container environments outperform the native environment when calling the *randnf* each iteration. Table 6.5 shows the average values of these results.

underlying operating system kernel or dependencies (like VOLK) differ between individual environments, the resulting performances can also be different. This becomes an issue since the focus of our performance testing presented in the next section is determining the overhead of the actual isolation environment itself rather than the overhead due to different versions or configurations of installed software. In this work, we did not focus on characterizing the overhead due to different software versions or operating systems, and instead attempted to keep the software versions and configurations as consistent as possible between the different environments.

Table 6.5: Waveform performance of a LiquidDSP waveform calling *randnf*. This table shows the throughput performance of a LiquidDSP waveform which is calling the *randnf* function each iteration while processing samples. Note that the multi-core performance takes a significant performance hit over the single core configurations, and the virtual machine and container environments both out-perform the native environment in multi-core configurations. Figures 6.12 and 6.13 show plots of these results.

Cores	**Environment**	**Average (Msps)**	**Deviation**	**Difference**
1	native	6.80	0.01	-
2		0.57	0.12	-
3		1.45	0.07	-
4		1.39	0.03	-
1	docker	6.79	0.01	0.16
2		0.65	0.16	-14.99%
3		1.48	0.08	-3.36%
4		1.42	0.06	-5.16%
1	virtualbox	6.74	0.01	0.83
2		0.92	0.17	-41.12%
3		1.63	0.12	-17.42%
4		1.56	0.01	-7.92%

6.4.7 Takeaways

While the specifics of why some of these behaviors occur are outside the scope of this dissertation, it is important to realize how the hardware and software configurations can significantly affect the measured throughput of a software waveform. Moving forward, our test framework was designed to address as many of these factors as possible in order to remove any variance in the results due to each. This allowed the overhead of the different isolation environments to be properly characterized without the results being obscured by other factors. First, all tests were forced to a fixed CPU frequency and all test processes were pinned to either the hyperthreaded or non-hyperthreaded cores. The chosen CPU frequency was slightly higher than 50% of the maximum CPU frequency to ensure the system did not overheat and damage the processor, and most tests were pinned to non-hyperthreaded cores in order to achieve the maximum possible performance from the flowgraphs. Second, all environments used similar software configurations as much as possible and all tests used the same testing workflow no matter which environment was being tested. All of the different isolation environments used the same base operating system as the native environment, and the versions of the

SDR frameworks (GNU Radio, VOLK, LiquidDSP) were kept consistent throughout as well. Any software updates to the native operating system were also applied to each isolation environment's operating system.

6.5 Results

In this section, we analyze the performance overhead of different isolation environments in terms of the impact to the maximum throughput of some example SDR waveforms. We executed the test waveforms in each isolation environment (containers and virtual machines) and compare the performance to the native environment to determine the overhead of the isolation. While the ultimate goal of our architecture is isolating individual blocks of a waveforms into different security environments, characterizing the overhead for a full waveform in an isolation environment gives us an initial understanding of the expected performance impact the isolation architecture has on an application.

We tested waveforms based on two different frameworks (GNU Radio and LiquidDSP) and multiple stress tests in the different environments (discussed in the previous section). No hardware front-ends were used with the SDR waveforms since the main goal was forcing a software bottleneck to determine the maximum throughput performance of the software stack itself. Unlike the waveforms, the stress tests execute with completely independent workers and indicate the minimal overhead that could be achieved in the isolation environments. For the tests, only non-shared, physical cores were used and the number of active cores was varied from 1 to 4 cores; all other unused cores were disabled.

While each SDR application will have unique overheads, the results from these tests provide a decent approximation of the overhead that could be expected for other applications and even identify where bottlenecks may exist within an SDR application executing in an isolated environment.

6.5.1 GNU Radio Results

Memory Performance

As mentioned before, some of the GNU Radio based tests were specifically designed to stress the framework and measure the overall memory bandwidth performance of a waveform. The *Null_Test* flowgraph attempts to force the system to be I/O bound in order to determine the upper limit of flowgraph throughput due to memory performance in the GNU Radio framework. As described earlier, there are no signal processing operations in the *Null_Test* flowgraph, and it only moves samples (32 bit real floats) through the application and framework. This effectively stresses the underlying GNU Radio buffers and scheduler; most operations are handling buffer pointers as data is "produced" and "consumed" by the blocks.

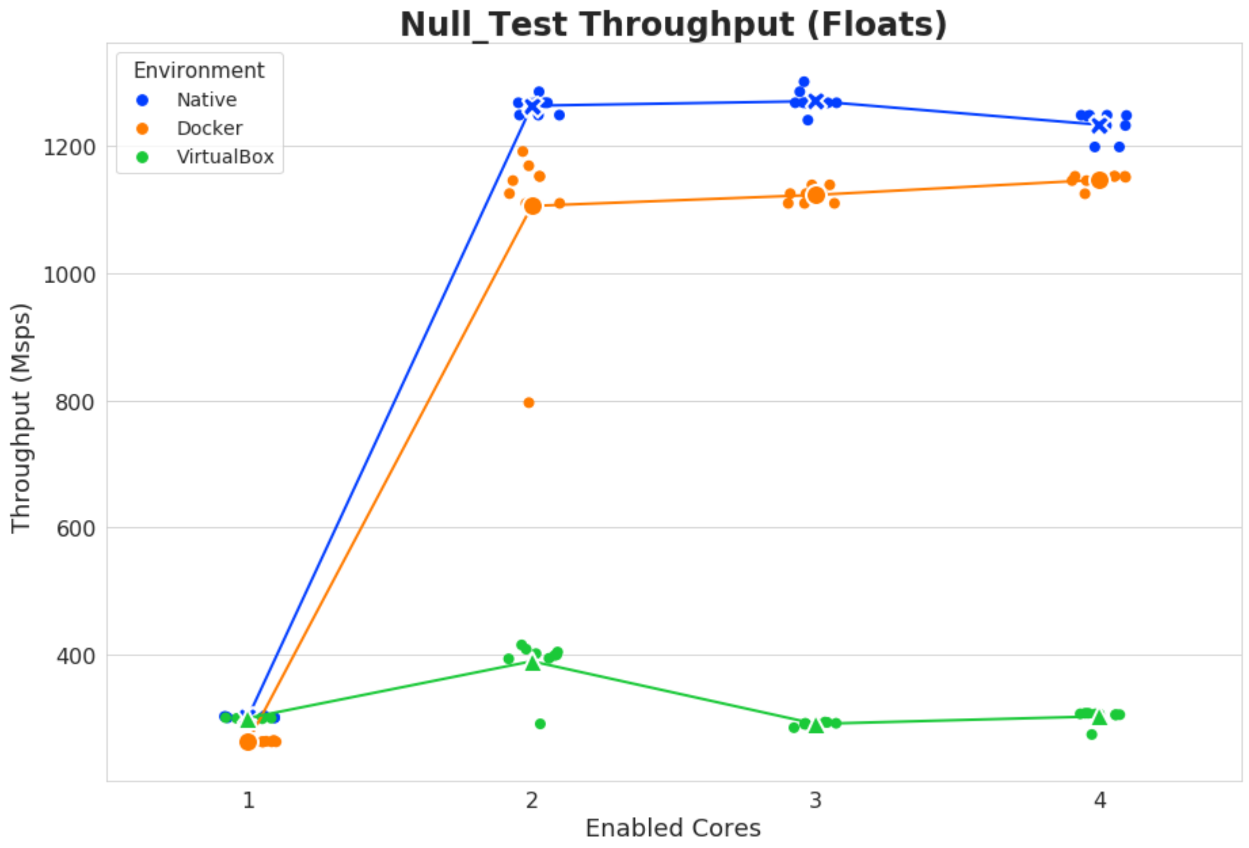


Figure 6.14: Null_Test flowgraph performance - This plot shows the throughput comparison of the *Null_Test* flowgraph implemented in GNU Radio and executing in different isolation mechanisms. The number of active cores varies from 1 to 4 using independent physical cores (no hyperthreaded cores). Colors denote the type of environment: Blue is native performance, orange is Docker performance, and green is a Virtual machine. Here, we can see that there is a large performance overhead for the *Null_Test* flowgraph in multi-core virtual machine environments. These results are also shown in Table 6.6.

By default, there are three blocks (*null_source*, *null_sink*, and *head*)in this flowgraph, but it can be configured to have any number of *copy* blocks that are also added after the source block. Increasing the number of blocks in the system can simulate larger flowgraphs in order to determine how well the scheduler can handle data passing through the flowgraph. Figure 6.14 and Table 6.6 show the throughput results of the *Null_Test* flowgraph in the native, container, and virtual machine environments. The throughput of each test was measured in megasamples per second (Msps).

The only operations executing in the *Null_Test* flowgraph are memory copy operations between a block's input and output buffers and updates to each block's buffer pointers as

Table 6.6: Null_Test flowgraph throughput - This table shows the throughput of the *Null_Test* flowgraph in different isolation environments. This flowgraph is designed to predominately stress the GNU Radio scheduler and buffers to determine the upper bound of memory performance of GNU Radio flowgraphs. This shows the percent overhead of the isolation environments as compared to native performance and the final column shows the difference between the container and virtual machine environments. Emphasized differences show where the virtual machine environment out-performed the container environments. Here, we can see that there is a large performance overhead for the *Null_Test* flowgraph in multi-core virtual machine environments.

Test	**Throughput (Msps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	301.83	263.30	300.26	12.77%	0.52%	*-12.25%*
2	1263.72	1106.02	389.61	12.48%	69.17%	56.69%
3	1270.91	1123.35	290.98	11.61%	77.10%	65.49%
4	1233.82	1147.17	302.94	7.02%	75.45%	68.43%

data is read from and written to these buffers. In the default configuration, this flowgraph only has three blocks, so there are only two shared buffers (between the *null_source* and *head* blocks and the *head* and *null_sink* blocks). So, the only operations are a memory copy in the *head* block and the corresponding update to its input and output buffer. With this limited amount of processing, this flowgraph quickly becomes I/O bound and demonstrates the upper throughput that would be possible even for the most simple of GNU Radio flowgraphs.

Since each block in a GNU Radio flowgraph executes within a separate thread, there is a processing increase by using multi-core configurations even with this extremely simple flowgraph. This flowgraph only consists of three threads in its default configuration, so there is a limited amount of parallelism possible. Even then, the dual-core tests show a massive increase in throughput over the single core configurations of the flowgraph for both the native and container environments. However, since there is limited parallelism within this flowgraph (only three blocks and threads exist), there is no additional benefit gained by increasing the number of cores.

Overall, the flowgraph in the native environment showed the best performance and the container environment showing 12% or less overhead between the different configurations. For both the native and container environments, the multi-core configurations showed a 4x performance increase over the single-core configuration. However, the virtual environment showed a massive performance loss for multi-core configurations. Interestingly, the virtual environment out-performed the container environment for a single core configuration

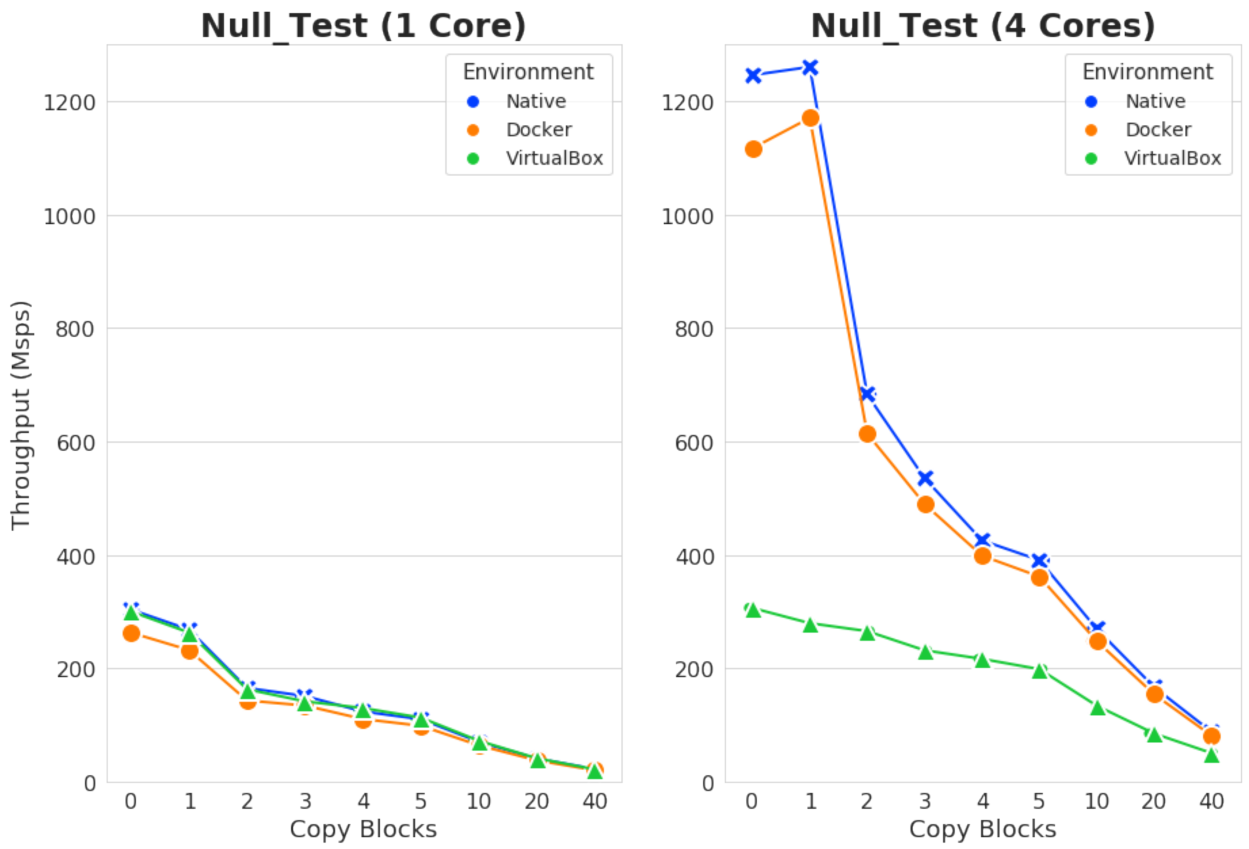


Figure 6.15: Null_Test throughput with multiple copy blocks (blocks vs throughput) - This plot shows the performance of the *Null_Test* flowgraph with additional *copy* blocks included in the flowgraph. Both plots show the throughput performance of the flowgraph from 0 to 40 copy blocks added to simulate longer flowgraphs. The plot on the left shows the single core configuration with the plot on the right showing a quad-core configuration. These results are also shown in Table 6.7.

and showed practically no overhead compared to native performance, but for all multi-core configurations, the virtual environment showed 70% lower performance than the native environment. Also, the multi-core configurations for the virtual environment showed effectively the same performance as the single core configuration.

Figures 6.15 and 6.16 and Table 6.7 show variations of the *Null_Test* flowgraph where additional *copy* blocks were included in the flowgraph. The number of *copy* blocks included in the flowgraph was varied from 0 to 40. Since this added significantly more parallelism to the flowgraph, increasing the number of cores does increase overall performance when there are a larger number of blocks in the flowgraph. In Figure 6.15, almost all of the tested configurations and environments show a performance increase when moving from a single-core to

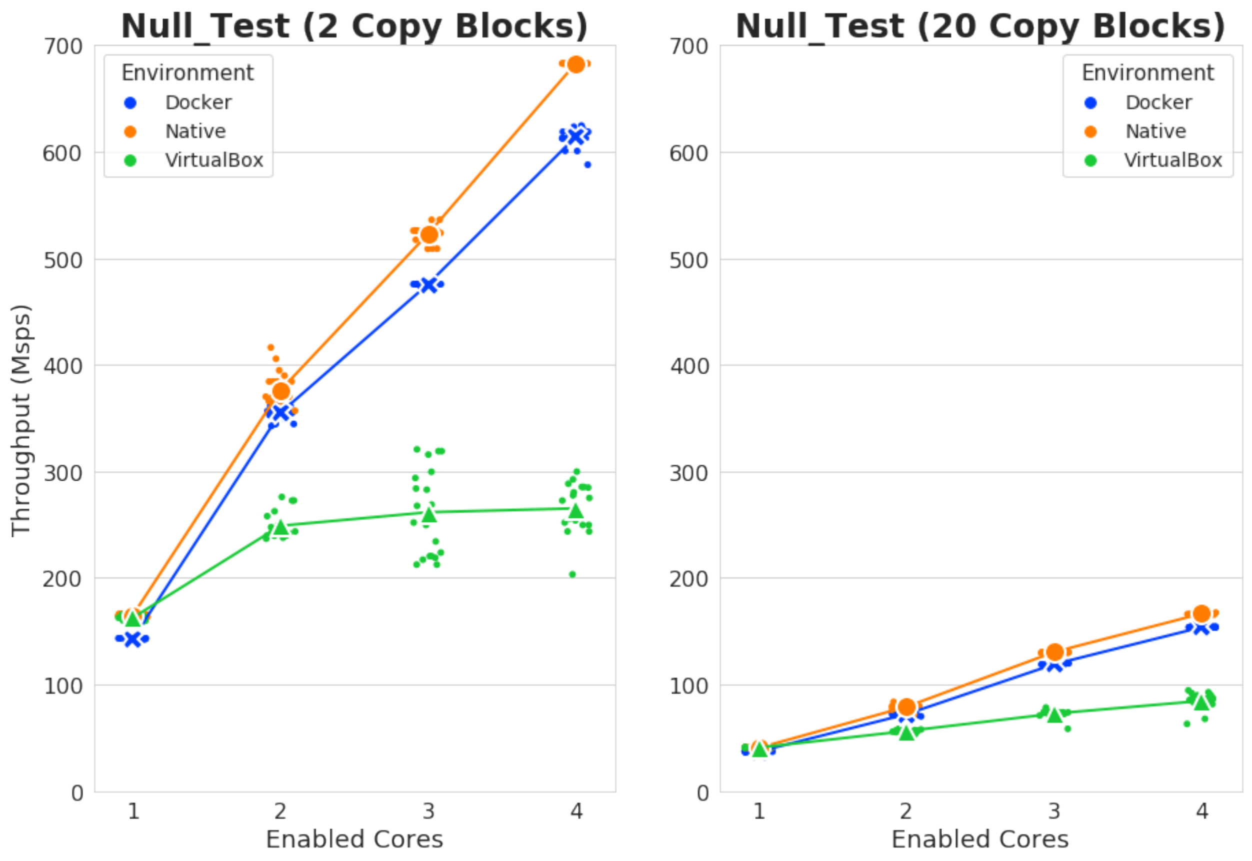


Figure 6.16: Null_Test throughput with multiple copy blocks (cores vs throughput) - This plot is similar to Figure 6.15, but plots the number of active cores against the flowgraph throughput. The plot on the left shows the results of a flowgraph with only a single copy block. The plot on the right shows the same comparisons for a flowgraph with 20 copy blocks. These results are also shown in Table 6.7.

a quad-core system. The native and container environments show the greatest performance increase when moving to the multi-core system. Interestingly, the virtual environment also sees a performance increase for the quad-core configuration as more additional blocks are included in the flowgraph, but compared to the other environments this increase is rather minimal.

Figure 6.16 shows similar results but compares the number of active cores rather than the number of *copy* blocks to the flowgraph throughput. Here, the performance increase is almost linear as the number of cores increases for the flowgraph with 20 copy blocks. With more copy blocks in the flowgraph, the virtual environment also performs better in multi-core configurations (which is to be expected with the increase in parallelism), but again the overall it still shows significant overhead when compared with the native and docker

Table 6.7: Null_Test flowgraph throughput (with copy blocks) - This table shows the throughput of the *Null_Test* flowgraph with additional copy blocks added into the flowgraph. Here, we show the results of the single-core and quad-core tests with 0 to 40 additional *copy* blocks in the flowgraph. This shows the percent overhead of the isolation environments as compared to native performance and the final column shows the difference between the container and virtual machine environments. Emphasized differences show where the virtual machine environment out-performed the container environment. Similar to Table 6.6, we can see that the virtual machine environment out-performed the container environment for all single-core configurations but incurs a large overhead for the quad-core tests.

Test		**Throughput (Msps)**			**Overhead**		
Cores	**Blocks**	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	0	303.30	262.60	299.97	13.42%	1.10%	*-12.32%*
1	1	268.02	231.93	262.80	13.46%	1.94%	*-11.52%*
1	2	165.05	143.00	162.71	13.36%	1.42%	*-11.94%*
1	3	150.80	134.06	141.38	11.10%	6.24%	*-4.86%*
1	4	123.53	110.22	129.64	10.77%	-4.95%	*-15.72%*
1	5	109.89	98.34	112.62	10.51%	-2.49%	*-13.00%*
1	10	69.90	63.77	71.82	8.77%	-2.75%	*-11.52%*
1	20	40.69	37.27	40.95	8.41%	-0.64%	*-9.05%*
1	40	22.00	19.86	21.41	9.75%	2.69%	*-7.06%*
4	0	1245.75	1116.67	306.08	10.36%	75.43%	65.07%
4	1	1260.12	1170.22	279.28	7.13%	77.84%	70.71%
4	2	682.46	613.66	265.27	10.08%	61.13%	51.05%
4	3	534.97	489.22	231.42	8.55%	56.74%	48.19%
4	4	425.26	398.27	216.33	6.35%	49.13%	42.78%
4	5	390.44	361.48	198.23	7.42%	49.23%	41.81%
4	10	269.11	247.66	133.76	7.97%	50.29%	42.32%
4	20	166.54	153.80	84.74	7.65%	49.12%	41.47%
4	40	87.81	80.95	50.21	7.82%	42.82%	35.00%

environments in multi-core systems.

The most surprising results from the *Null_Test* flowgraph tests are the multi-core, multi-copy throughputs in the virtual environment. In the native and containerized environments, moving to the multi-core configuration resulted in a significant performance gain over the single-core tests. However, in the case of the virtual machine environment, the performance of the quad-core configuration never exceeds the fastest single-core configuration (which was the 0 *copy* block test) and there are minimal performance increases as more blocks are added. Based on the large jump in performance of the container and native environments, the expectation was a similar performance increase should have been observed when moving to quad-core virtual machine configurations. Another surprising result is the virtual machine tests always out-performed the container environments when in the single core configuration.

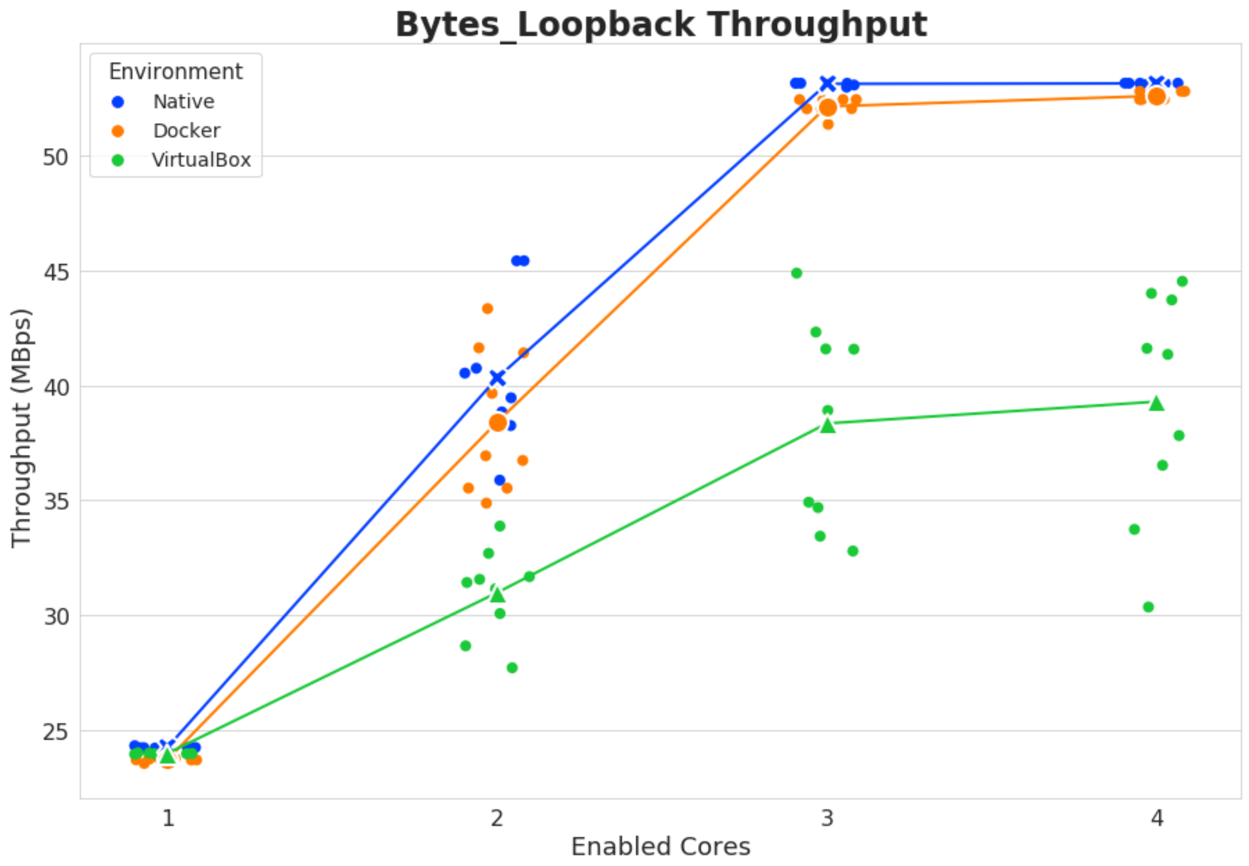


Figure 6.17: Bytes_Loopback flowgraph throughput - This plot shows the throughput of the *Bytes_Loopback* flowgraph executing in different isolation environments. The number of active cores varies from 1 to 4 using independent physical cores (no hyperthreaded cores). Colors denote the type of environment: Blue is native performance, orange is container performance, and green is a virtual machine. Similar to the *Null_Test* results, this flowgraph also has a large overhead when executing in the virtual machine environment. These results are also shown in Table 6.6

While the *Bytes_Loopback* flowgraph is similar to the *GFSK_Loopback*, it is also designed to stress the GNU Radio scheduler like the *Null_Test* flowgraph. The modulation and

Table 6.8: Bytes_Loopback flowgraph throughput - This shows the throughputs of the *Bytes_Loopback* flowgraph in different isolation environments. Like the *Null_Test* flowgraph, this flowgraph also predominately stresses the GNU Radio scheduler and includes some additional data processing blocks. Similar to previous results, the virtual machine environment shows the highest overhead for multi-core configurations and out-performs the container environment for single-core tests.

Test	**Throughput (MBps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	24.24	23.70	23.98	2.20%	1.08%	*-1.12%*
2	40.32	38.40	30.98	4.76%	23.10%	18.34%
3	53.12	52.13	38.34	1.85%	27.80%	25.95%
4	53.14	52.59	39.29	1.02%	26.00%	24.98%

demodulation blocks in the loopback flowgraphs are the most processing intensive and likely become bottlenecks in the system. Since the *Bytes_Loopback* flowgraph does not include these modulation and demodulation stages and has *copy* blocks in place of these blocks, it forces more stress on the data handling blocks and the scheduler.

Figure 6.17 and Table 6.8 show the throughput results of the *Bytes_Loopback* flowgraph in the native, container, and virtual machine environments. The *Bytes_Looback* flowgraph shows similar behavior to the *Null_Test* flowgraph when moving from the single-core to multi-core configurations. This flowgraph has several more blocks than the default *Null_Test* flowgraph, so there are more buffers and overall functionality that can better utilize multi-core configurations. Since there is actual data processing (though still minimal) in this flowgraph, the overall throughputs are not as high as the *Null_Test* flowgraph. But, there is still a large performance increase from single-core to dual-core configurations and a smaller increase to triple-core and quad-core configurations. The virtual environment again out-performed the container environment for single-core configurations, but it showed the lowest performance for multi-core configurations. These tests showed much lower overheads than the *Null_Test*. The container environment showed less than 5% overhead for all configurations, and the virtual environment showed 1% for single-core and around 25% for multi-core configurations.

Overall, these results show that the memory overhead for GNU Radio flowgraphs executing within containerized environments is relatively low. In virtualized environments the flowgraphs take massive performance hits and operate with 50-70% lower performance in many cases. Memory performance seems to be a major bottleneck for GNU Radio flowgraphs in virtualized environments. This trend is not too surprising since virtual machines also virtualize the memory system which would produce more overhead in the overall system.

Signal Processing Performance

While the *Bytes_Loopback*, *GFSK_Loopback*, and *GMSK_Loopback* flowgraphs are very similar, the goal of the latter flowgraphs is to force the system to be CPU bound rather than I/O bound. These flowgraphs do include the modulation and demodulation stages to better represent an actual SDR waveform that includes both transmit (modulation) and receive (demodulation) components within the application. The implemented modulation and demodulation blocks cause the flowgraph to be predominately CPU bound. Overall the *GFSK_Loopback* and *GMSK_Loopback* flowgraphs are nearly identical with slightly different modulation schemes. The results of executing the *GFSK_Loopback* flowgraph in the container and virtual environments are shown in Table 6.9 and are plotted in Figure 6.18. The throughput for these tests was measured in megabytes per second (MBps).

Table 6.9: GFSK_Loopback flowgraph throughput - This shows the throughput comparison of various GNU Radio flowgraphs executing in different isolation environments and shows the percent overhead of the isolation environment as compared to the flowgraph performance in the native environment. Overall, the results of the *GFSK_Loopback* are extremely similar in terms of percent overhead to the *Bytes_Loopback* results presented earlier.

Test	**Throughput (MBps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	0.57	0.55	0.55	3.94%	2.45%	*-1.49%*
2	1.02	1.00	0.85	2.09%	16.69%	14.60%
3	1.53	1.52	1.11	0.62%	27.40%	26.78%
4	1.84	1.82	1.32	1.40%	28.14%	26.74%

These results show that even with the added signal processing blocks, this flowgraph has basically the same performance in terms of additional overhead as the *Bytes_Loopback* flowgraph. The overall throughput is significantly lower because this flowgraph is now CPU bound, but the percentage overhead between the different environments and the multi-core configurations is essentially the same. The container environment has a relatively low throughput overhead (maximum of about 4%), while the virtual machine can have a much higher overhead (maximum of about 28%). Again, for single-core configurations the virtual environment out-performs the containerized environment and has a very low performance overhead.

Both the scheduler/buffer stressing tests and the signal processing tests indicate that the

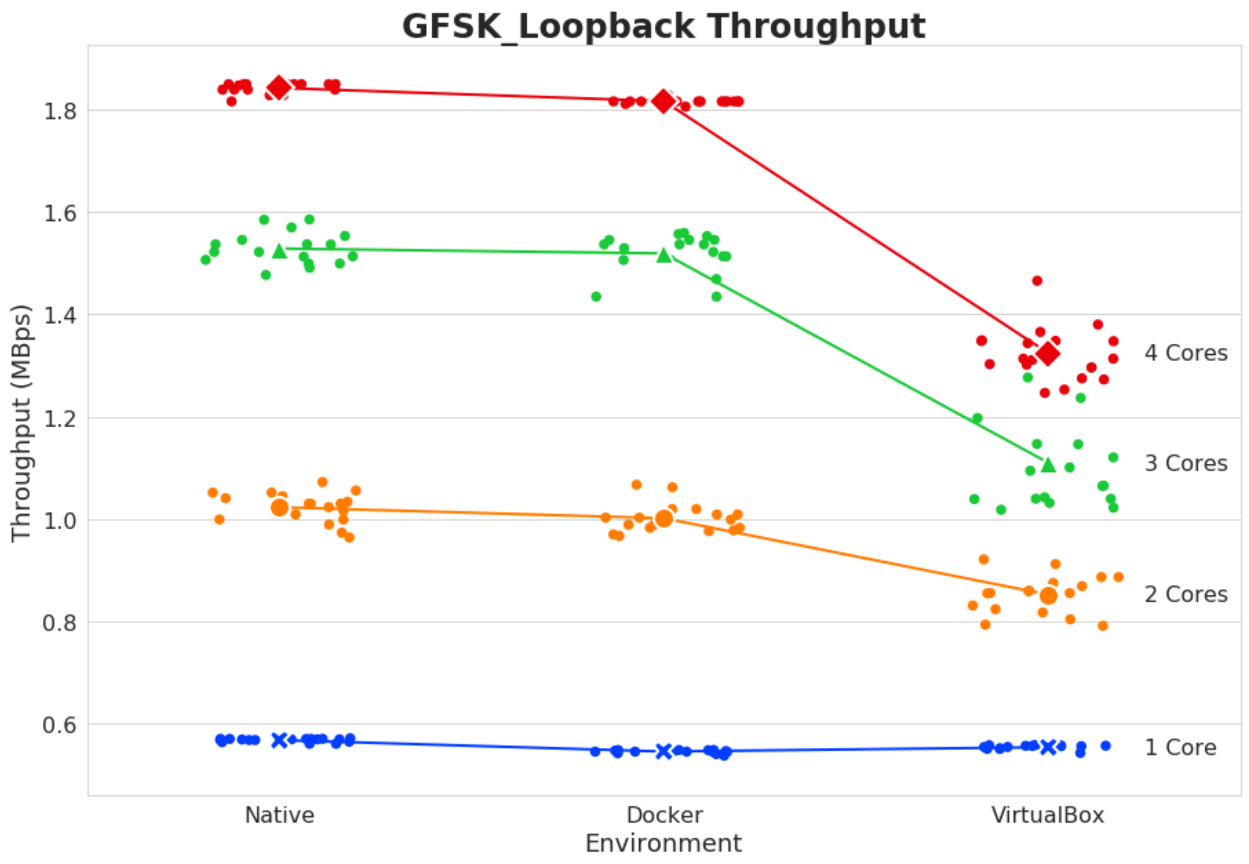


Figure 6.18: GFSK_loopack flowgraph throughput comparison - Throughput comparison of the *GFSK_Loopback* flowgraph implemented in GNU Radio executing in different isolation environments. The number of active cores varies from 1 to 4 using independent physical cores (no hyperthreaded cores). Colors denote the number of active processor cores: 1 - Blue, 2 - Orange, 3 - Green, and 4 - Red. These results are also shown in Table 6.9

overhead added to a GNU Radio flowgraph when executing in container or virtual machine environments is significantly driven by the memory performance within the GNU Radio framework.

6.5.2 LiquidDSP Results

In addition to the GNU Radio flowgraphs, we also tested several SDR applications that utilized the LiquidDSP framework. Specifically, we tested multi-threaded versions of a filtering component and the *FlexFrame* framing module. The results for the different tests described in Section 6.3 are shown below.

Filtering Performance

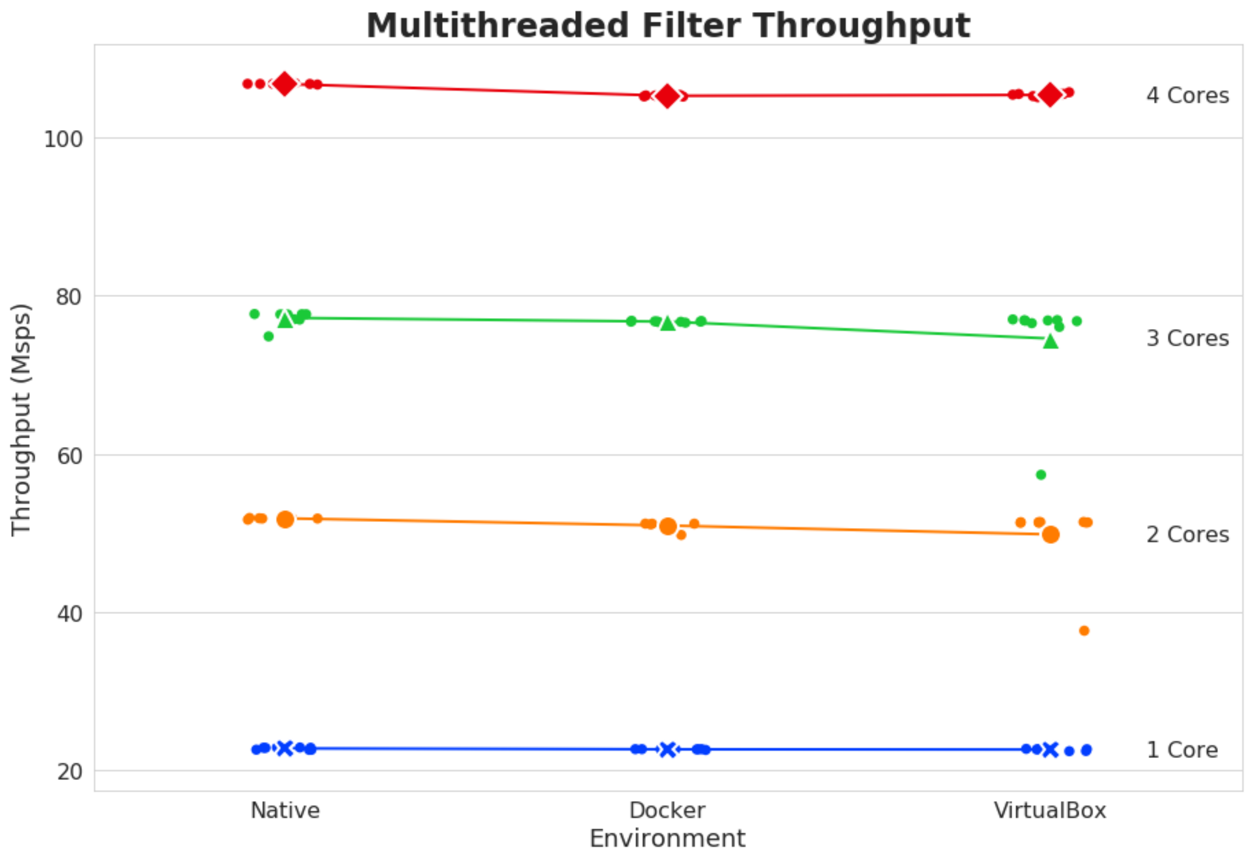


Figure 6.19: LiquidDSP multi-threaded filter throughput - This shows the results of the multi-threaded filter benchmark based on the LiquidDSP framework. Here, we plot the throughput of the test versus the isolation environment, where the colors represent the number of active cores. These tests show very little overhead for the isolation environments with containers having less than 2% overhead and virtual machines having less than 4% overhead. Table 6.10 shows the results of these tests.

The filtering benchmark tests the LiquidDSP FIR filter components to predominately stress the CPU with signal processing operations. Unlike the previous GNU Radio flowgraphs, there is no synchronization required between the individual threads executing in the test. For these tests, a single thread was allocated for each active core and the throughput was measured in megasamples per second (Msps).

The results for the filtering test are shown in Figure 6.19 and Table 6.10. These results are vastly different than the GNU Radio results and show practically no overhead for the filtering operation in any isolation environment. The container environment incurred less

Table 6.10: LiquidDSP multi-threaded filter throughput - This table shows the throughput results of the multi-threaded filter benchmark using the LiquidDSP FIR filter in different isolation environments. Here, the results show very little overhead when executing the LiquidDSP filter operations in the different environments.

Test	**Throughput (Msps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	22.73	22.62	22.58	0.50%	0.64%	0.14%
2	51.83	50.92	49.80	1.75%	3.92%	2.17%
3	77.20	76.72	74.58	0.62%	3.39%	2.77%
4	106.78	105.28	105.40	1.40%	1.30%	*-0.10%*

than 2% overhead and the virtual machine environments had less than 4% overhead.

FlexFrame Performance

Table 6.11: LiquidDSP multi-threaded FlexFrame throughput (no synchronziation) - This table shows the throughput results of the multi-threaded LiquidDSP FlexFrame benchmark (without synchronization) in different isolation environments. Here, the results show very little overhead when using the LiquidDSP FlexFrame module in the different environments.

Test	**Throughput (Mbps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	1.34	1.32	1.33	0.84%	0.44%	*-0.40%*
2	2.72	2.69	2.70	1.29%	1.01%	*-0.28%*
3	4.11	4.07	4.07	1.07%	1.05%	*-0.02%*
4	5.51	5.45	5.45	0.93%	0.96%	0.03%

Like the filtering benchmarks, the LiquidDSP *FlexFrame* benchmarks tested also predominately stressed the CPU. All digital communication waveforms have some form of structuring (or framing) transmitted data. The FlexFrame module implements a basic framing structure that can be easily integrated into systems to transmit a wireless signal. The performance of this module in different isolation environments is a good indicator of what overhead can be expected when adding similar isolation to production systems. We tested three variations

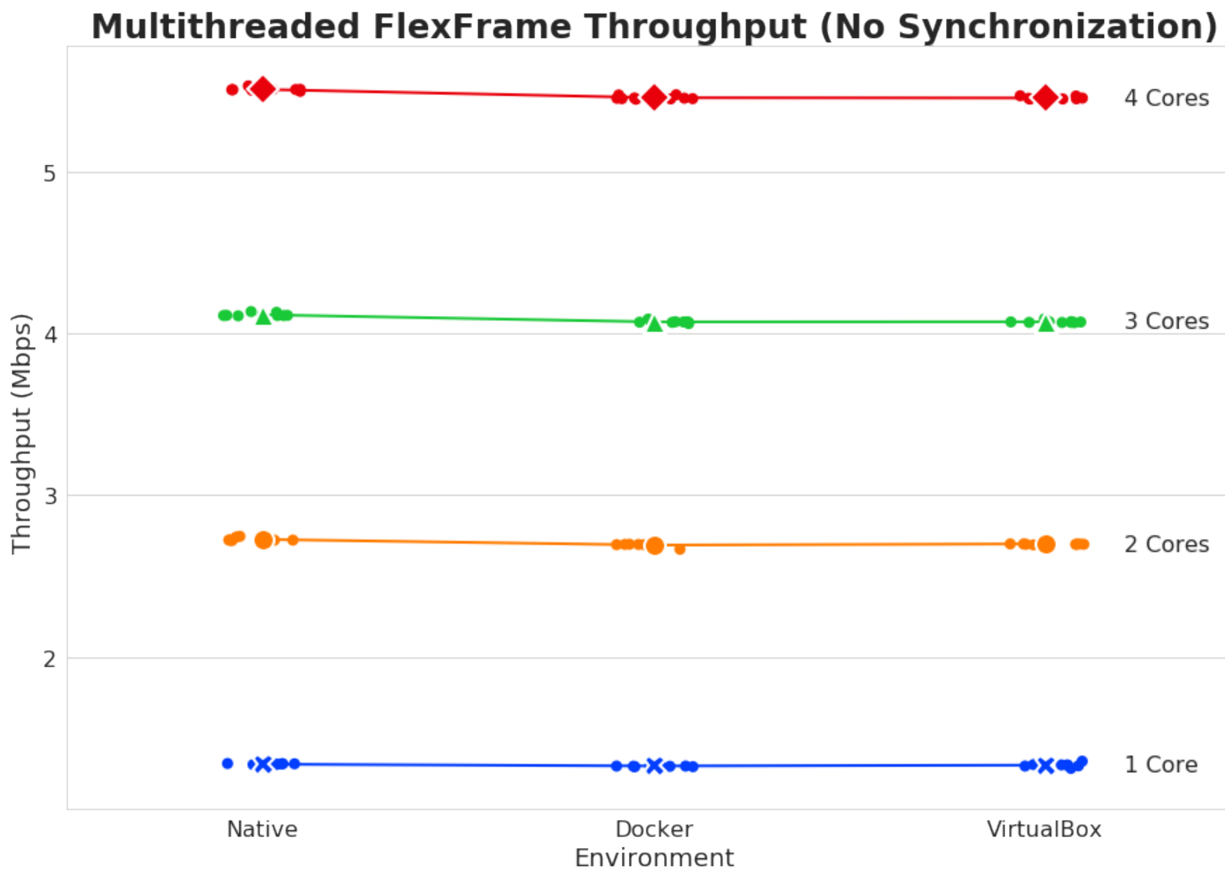


Figure 6.20: LiquidDSP multi-threaded FlexFrame throughput (no synchronization) - This shows the results of the multi-threaded FlexFrame benchmark with no synchronization between threads. Here, we plot the throughput of the test versus the isolation environment, where the colors represent the number of active cores. These tests show very little overhead for the isolation environments with both containers and virtual machines showing around 1% overhead compared to native performance. Table 6.11 shows the results of these tests.

of the FlexFrame benchmark that included: 1) no synchronization between threads, 2) synchronization with locks shared between all threads, and 3) synchronization using a linked list to track generated frames. The throughput for all of these tests was measured in megabits per second (Mbps).

Results for the tests without thread synchronization are shown in Figure 6.20 and Table 6.11. In this case, a worker thread consisting of a single frame generator and frame synchronizer pair was started for each active CPU in the system. Overall, there was very little overhead observed when executing these tests in the different isolation environments, with both the container and virtual machine environments having around 1% overhead across every con-

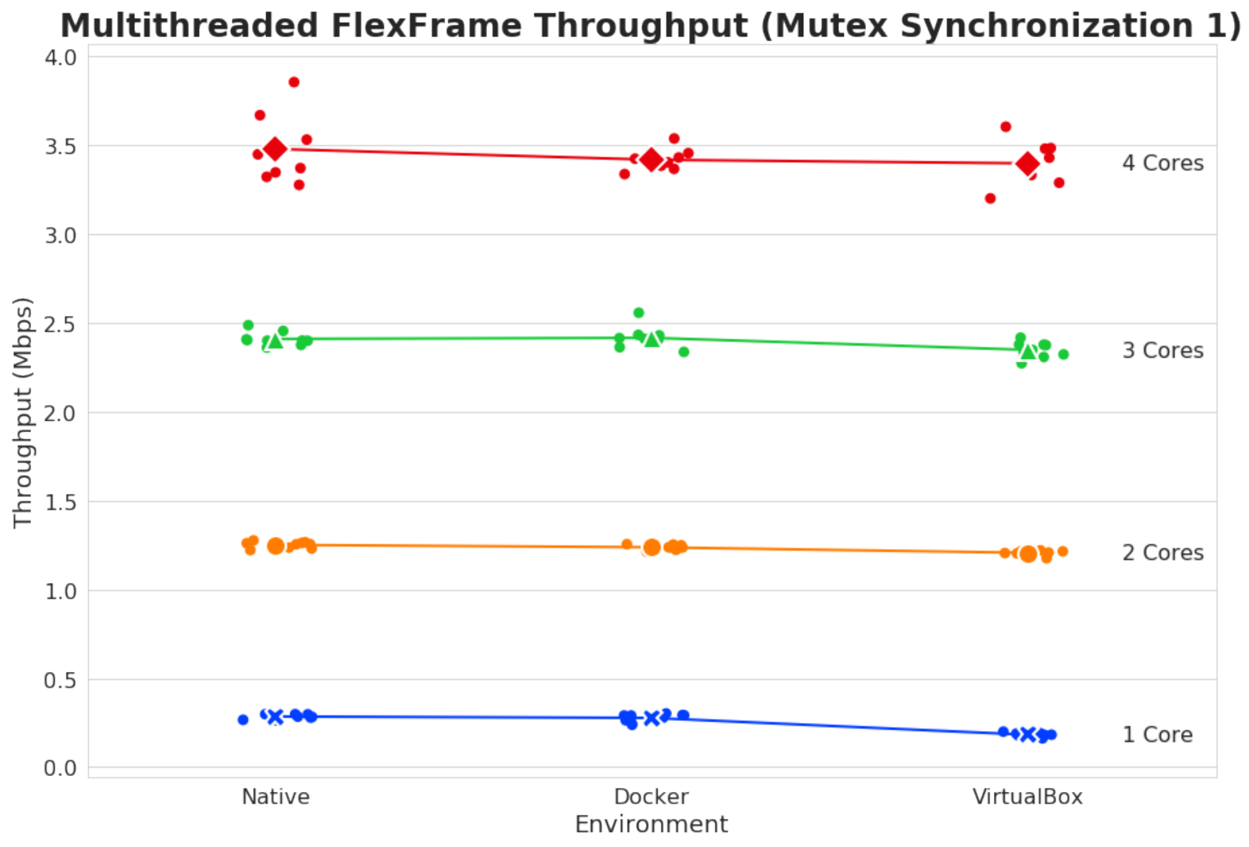


Figure 6.21: LiquidDSP multi-threaded FlexFrame throughput (with synchronziation) - This shows the results of the multi-threaded FlexFrame benchmark with synchronization between a single transmitter thread and multiple receiving threads. Here, we plot the throughput of the test versus the isolation environment, where the colors represent the number of active cores. These tests show very little overhead for the container environment with less than 3% overhead for all configurations. The virtual machine environment has a high overhead for the single-core configuration (36%), but relatively low overhead for multi-core environments (less than 4%). Table 6.12 shows the results of these tests.

figuration. There was roughly an equal increase in performance for every additional core added to the system. This trend was expected since there was no synchronization between executing threads and only non-hyperthreaded cores were enabled.

The results for the benchmark tests with thread synchronization are shown in Figures 6.21 and 6.22 and Tables 6.12 and 6.13. In these tests, a single transmitter thread generated frames and passed a pointer for a memory buffer containing generated samples to a receiver thread which processed the generated frame. For the first synchronization method, a lock was allocated for each receiver thread which the transmitter needed to obtain before checking

Table 6.12: LiquidDSP multi-threaded FlexFrame throughput (with synchronization) - This table shows the throughput results of the multi-threaded LiquidDSP FlexFrame benchmark (with thread-to-thread synchronization) in different isolation environments. These results show very little overhead for the FlexFrame module in containerized environments, with slightly higher overheads for multi-core virtual machine environments. The most overhead is observed in the single-core virtual machine environment.

Test	**Throughput (Mbps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	0.28	0.28	0.18	2.79%	36.16%	33.37%
2	1.25	1.24	1.21	1.09%	3.58%	2.49%
3	2.41	2.41	2.35	-0.25%	2.56%	2.81%
4	3.48	3.42	3.40	1.74%	2.33%	0.59%

Table 6.13: LiquidDSP multi-threaded FlexFrame throughput (with linked list) - This table shows the throughput results of the multi-threaded LiquidDSP FlexFrame benchmark (with linked-list synchronization) in different isolation environments. These results are similar to the results shown in Table 6.12 where very little overhead for the FlexFrame module is seen in containerized environments and with multi-core virtual machine environments. Again, the most overhead is observed in the single-core virtual machine environment.

Test	**Throughput (Mbps)**			**Overhead**		
Cores	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
1	0.52	0.50	0.46	2.77%	11.36%	8.59%
2	1.74	1.75	1.73	-0.67%	0.62%	1.29%
3	3.19	3.18	3.16	0.55%	1.02%	0.47%
4	4.76	4.73	4.72	0.73%	0.87%	0.14%

the status of the receiver and passing pointers. The second method used a shared linked-list for synchronization; the transmitter thread generated frames and added pointers to the list and the receiver threads pulled these pointers and processed frames when ready. The number of receiver threads started was equal to the number of active cores in the system.

Overall, the second synchronization method performed better and showed a lower overhead across most of the tested configurations. Both variations performed well in the containerized environment with less than 3% overhead for all tests. The multi-core virtual machine tests

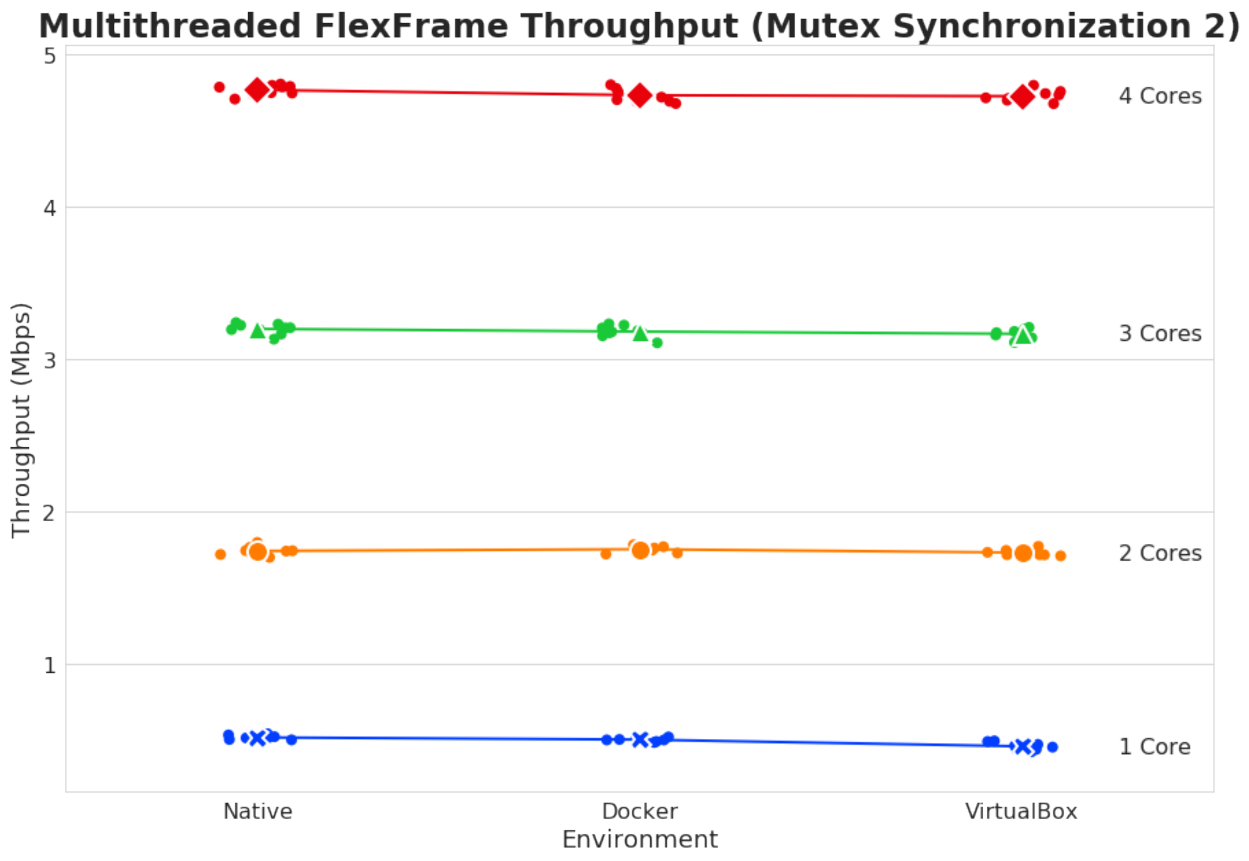


Figure 6.22: LiquidDSP multi-threaded FlexFrame throughput (with linked list) - This shows the results of the multi-threaded FlexFrame benchmark with synchronization between a single transmitter thread and multiple receiving threads using a linked list storing pending frames. Here, we plot the throughput of the test versus the isolation environment, where the colors represent the number of active cores. These tests show very little overhead for the container environment with less than 3% overhead for all configurations. The virtual machine environment has a high overhead for the single-core configuration (11%), but almost no overhead for multi-core environments (about 1%). Table 6.13 shows the results of these tests.

also showed very little overhead with the first synchronization method having 2-4% overhead and the second method having around 1% overhead.

The single core configurations actually showed the highest overhead for both the containerized and virtual machine environments. In the containerized environment, the single-core shows less than 2% additional loss compared to the multi-core configurations, which is still small. However, the single-core performance in the virtual environments shows a large performance overhead; the first synchronization method incurred a 36% loss in performance and the second method showed an 11% performance loss. This overhead could be caused by the transmitter and receiver threads both executing on the same core and therefore competing for available resources on the system.

6.5.3 Stressor Results

As mentioned before, we also executed several stress tests using the *stress-ng* utility which helps characterize the overhead expected for different components in the system when executing in isolated environments. The stress tests mainly focused on three components: the CPU, memory, and the kernel. For each test, a worker thread was launched for each core currently enabled for that test. Tests executed for a fixed length of time and returned statistics including the number of bogus operations per second (bogo/s) achieved. These results are not comparable between different stressors since each set of operations for a stressor is unique. The results for the stress tests described in Section 6.3 are shown below.

Table 6.14: CPU stressor performance results - This table shows the results of multiple CPU stressors (*matrix*, *bsearch*, and *hsearch*) executing in different isolation environments. Overall, there was very little performance loss for CPU intensive tasks executing in the container and virtual machine environments. The average performance loss for the container tests was less than 0.5% and less than 1.5% for the virtual machine tests.

Test		**Operations/second**			**Overhead**		
Stressor	**Cores**	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
matrix	1	354.20	353.74	350.98	0.13%	0.91%	0.78%
	2	709.57	709.07	694.85	0.07%	2.07%	2.00%
	3	1064.51	1064.51	1050.00	0.00%	1.36%	1.36%
	4	1420.03	1417.66	1403.44	0.17%	1.17%	1.00%
search (binary)	1	301.34	300.88	298.85	0.15%	0.83%	0.68%
	2	603.35	602.66	594.00	0.11%	1.55%	1.44%
	3	905.43	904.95	893.11	0.05%	1.36%	1.31%
	4	1207.34	1204.19	1188.54	0.26%	1.56%	1.30%
search (hash)	1	2991.87	2955.52	2956.19	1.21%	1.19%	*-0.02%*
	2	6003.41	5937.25	5910.33	1.10%	1.55%	0.45%
	3	8999.18	8931.08	8898.54	0.76%	1.12%	0.36%
	4	11996.67	11893.71	11860.63	0.86%	1.13%	0.27%

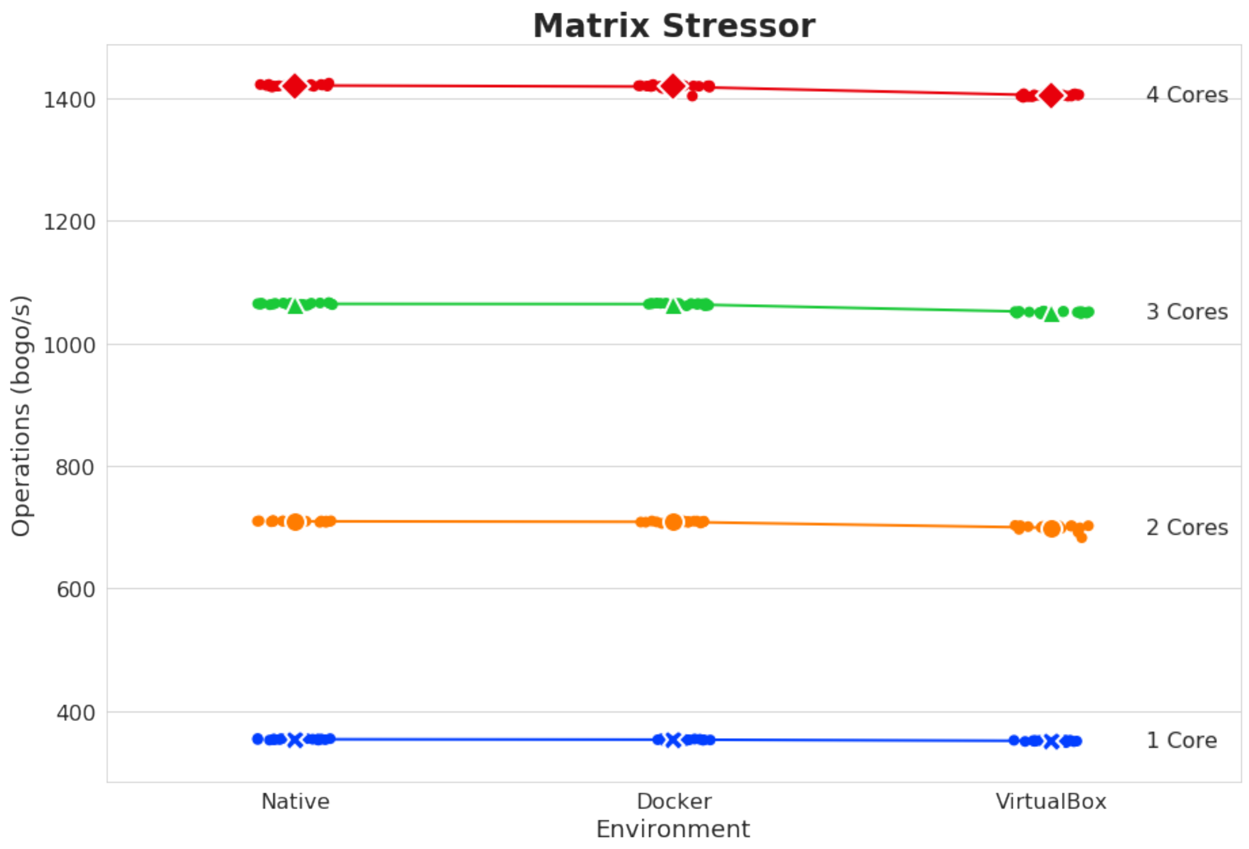


Figure 6.23: Matrix stressor results - This shows the results of the *matrix* stressor executed in different isolation environments. The *matrix* performs a mix of floating point, cache, and memory operations which mainly stress the CPU performance in the system. Results show a small overhead for both the container and virtual environments with a 2% or less loss in performance from the native environment. The results of the *matrix* stressor tests are shown in Table 6.14.

Figure 6.28 and Table 6.14 show the results for the *matrix*, *bsearch*, and *hsearch* stressors that mainly stressed the CPU performance in the system. These tests show very little overhead (or none) when running in the different isolation environments. The virtual machine results showed a maximum of 2% loss for the *matrix* stressor when compared to the native performance. On average, the performance loss for the container tests was less than 0.5% and less than 1.5% for the virtual machine tests. Overall, these results show that the CPU performance itself does not suffer much overhead when executing in an isolated environment.

The *stream* and *vm* stressors were used to stress memory performance in the system and determine the memory overhead of the isolated environments. The results for the *stream* stressor are shown in Figure 6.24 and Table 6.15. This test stresses memory streaming

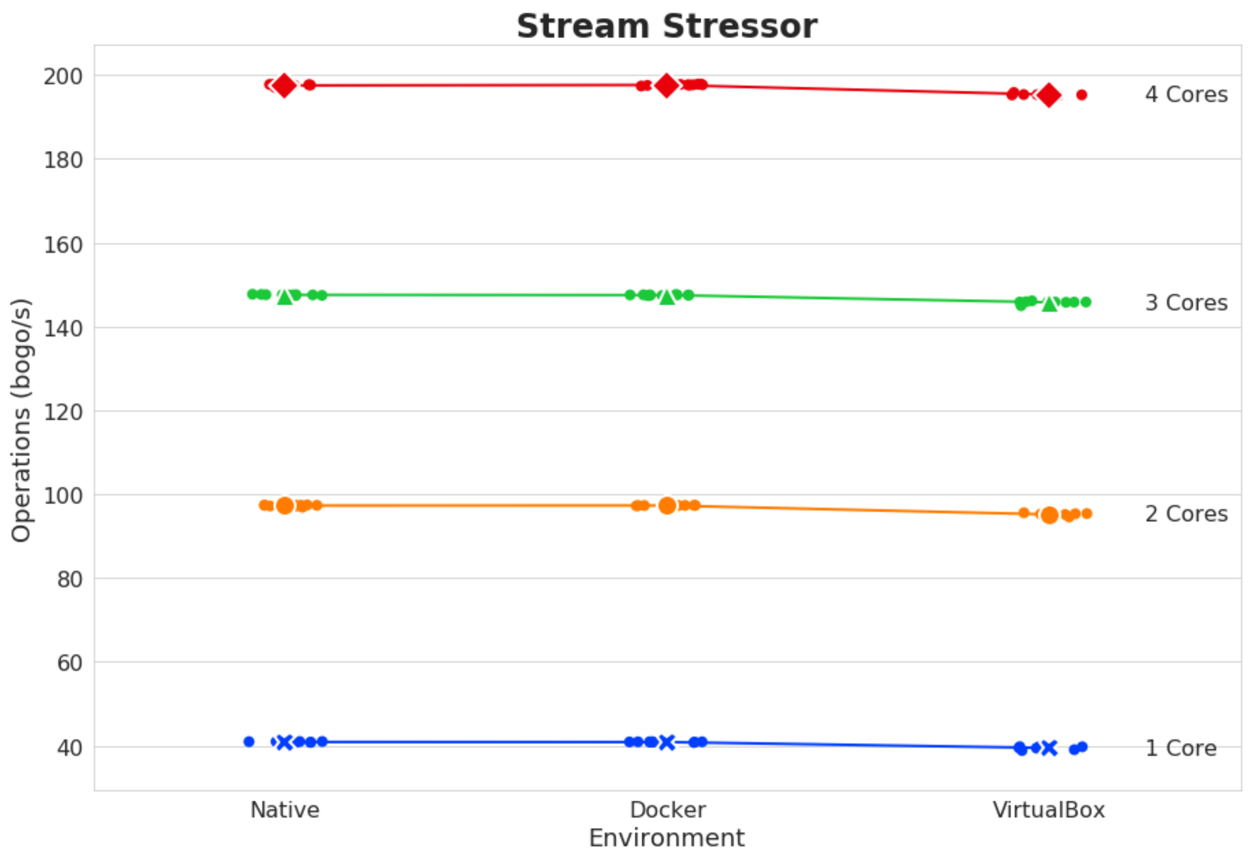


Figure 6.24: Stream stressor results - This shows the results of the *stream* stressor executed in different isolation environments. This test stresses memory streaming performance by allocating multiple buffers that are 4+ times larger than the CPU cache and performing multiple floating point operations (copy, scale, add, triad) between the buffers. Results show a minor performance loss (3.5%) for the virtual machine environment and no overhead for the container environment compared to the native performance. The results of the *stream* stressor tests are shown in Table 6.15.

performance by allocating multiple buffers that are 4+ times larger than the CPU cache and performing multiple floating point operations (copy, scale, add, combined) between the buffers in a manner similar to how a DSP algorithm would process samples. Results show a minor performance loss (3.5%) for the virtual machine environment and no overhead for the container environment when compared to the native performance. Also, the single-core virtual machine tests showed the highest amount of overhead for all of the tested configurations.

The *vm* stressor continuously allocates and deallocates memory and stresses memory performance through one of several methods for manipulating the buffers. In our testing, we

Table 6.15: Stream (memory) stressor performance results - This table shows the results of the *stream* memory stressor executing in different isolation environments. These results show no performance loss for the container environment and a small performance loss (3.5%) for the virtual machine environment compared to the native performance. Also, the multi-core virtual machine tests showed lower overhead than the single core configuration.

Test		**Operations/second**			**Overhead**		
Stressor	**Cores**	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
stream	1	40.88	40.86	39.45	0.05%	3.50%	3.45%
	2	97.27	97.29	95.16	-0.02%	2.17%	2.19%
	3	147.54	147.49	145.76	0.03%	1.21%	1.18%
	4	197.49	197.61	195.30	-0.06%	1.11%	1.17%

focused on five specific stress methods described below:

- **flip** - Inverts a single bit of each byte in the buffer per loop until the entire byte is inverted in 8 iterations
- **incdec** - Loop through the buffer twice incrementing each byte by a set value on the first pass and decrementing to the original value on the second pass
- **read64** - Read 32 x 64 bit chunks of memory from a buffer in a single operation
- **write64** - Write 32 x 64 bit chunks of memory to a buffer in a single operation
- **all** - Iterate over all possible stress methods

Figures 6.25 and 6.26 plot the performance using the *flip* and *write64* memory operations, and Table 6.16 shows the results of all tested stress methods. For many of the tests, the container and virtual machine stress tests were on par with the performance of their native counterparts. The results showed no performance overhead for the *flip* and *incdec* stress methods and for most configurations testing *all* of the stress methods. Only the quad-core virtual machine tests showed a small overhead (2.7%) when testing *all* stressor methods. However, the *read64* and *write64* methods did incur a 2-7% loss compared to the native performance. Memory reading performance in the container and virtual machine tests were comparable, but the virtual machine tests out-performed the containerized tests when stressing memory writes.

The largest overhead for any of the stress tests was seen in the *switch* and *context* stressors. Both of these stressors rapidly force context switching to occur at either the thread or process

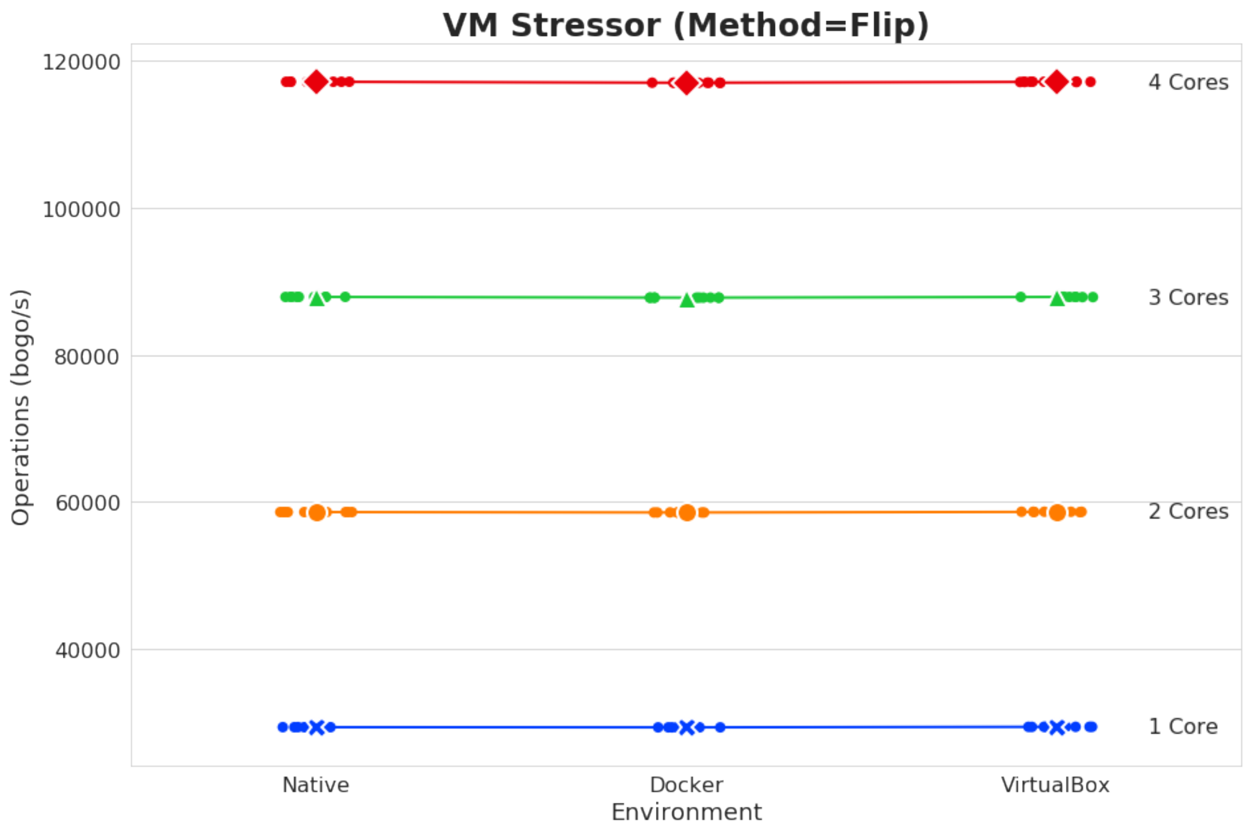


Figure 6.25: Virtual memory stressor results (flip method) - This shows results of the *vm* stressor in different environments using the *flip* operation. The *vm* stressor continuously allocates and deallocates memory and writes to the mapped buffers using various methods. For the *flip* operation, a single bit is inverted each loop until the entire byte is inverted in 8 iterations. The *flip* method did not incur a performance loss in the container and virtual machine environments. These results are shown in Table 6.16.

level which stresses kernel performance. The *context* stressor forces context switching among multiple threads, and the *switch* stressor forced context switches between processes passing messages through pipes.

Many SDRs are implemented as multi-threaded applications, and as the data and samples are passed to different blocks in the application, significant amounts of context switching can occur. This is essentially true for waveforms using the GNU Radio framework as each block in the waveform executes in a separate thread. Significant overhead due to context switching can easily affect the overall throughput of an SDR waveform. So, understanding the impact of context switching in isolated environments is important.

For the *context* stress tests, both the container and virtual machine isolation environments

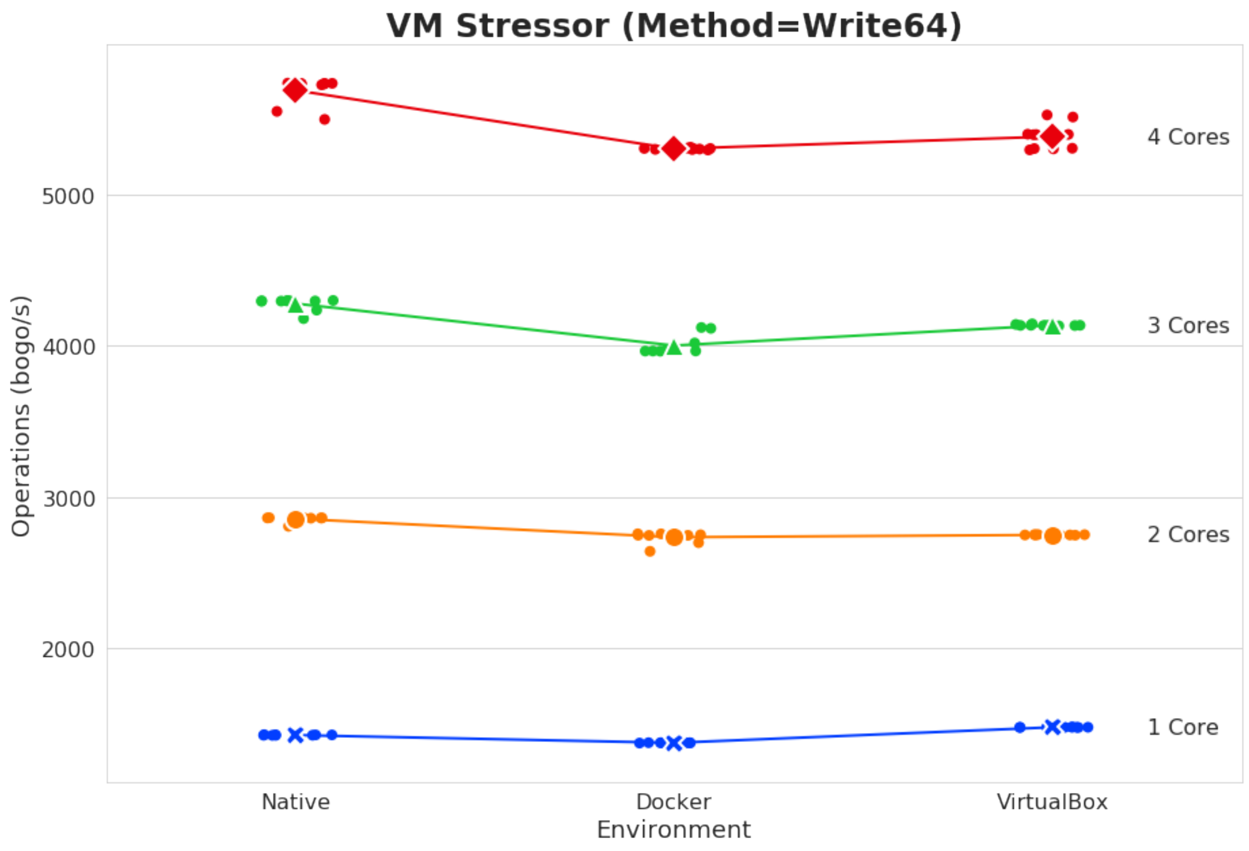


Figure 6.26: Virtual memory stressor results (write64 method) - This shows results of the *vm* stressor in different environments using the *write64* operation. This test writes 32 x 64 bit chunks to the allocated memory buffers per operation. Unlike the *flip* methods, the *write64* method does incur a 3-7% loss compared to the native performance. In this test, the virtual machine environment typically out-performed the containerized environment. These results are shown in Table 6.16.

experienced an 8-10% loss in performance compared to the native environment. When using the *switch* stressor, the container environment suffered a massive performance loss (around 70%) for all configurations. The virtual environment fared much better and had a maximum of 16% loss for a quad-core configuration and lower overheads for fewer active cores. Interestingly, the virtual machine environment again out-performed the container environment for all of the context switching stress tests.

This overhead in the container environment is likely because containers are a form of operating system virtualization where all isolation is enforced at the kernel level. The kernel creates a new namespace with separate copies of internal data structures and uses this to build the isolated environment for containerized processes. So, rapid context switching between these

Table 6.16: Virtual memory stressor performance results - This table shows the performance comparison of the *vm* memory stressor executing in different environments. Several different methods were tested for this stressor. The *flip* and *incdec* methods both showed no performance overhead in the different environments, while the *read64* and *write64* methods had a 2-7% overhead for various configurations. For the *write64* tests, the virtual machine tests out-performed the containerized tests. When executing all available stressor methods (*all*), there was no overhead for most of the tested configurations.

Test		**Operations/second**			**Overhead**		
Method	**Cores**	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
all	1	53759.84	53714.37	53835.01	0.08%	-0.14%	*-0.22%*
	2	107497.71	107398.37	107587.93	0.09%	-0.08%	*-0.18%*
	3	161258.86	161064.61	161250.30	0.12%	0.01%	*-0.12%*
	4	215008.42	214759.74	209140.14	0.12%	2.73%	2.61%
flip	1	29290.73	29266.79	29337.76	0.08%	-0.16%	*-0.24%*
	2	58578.04	58519.92	58606.60	0.10%	-0.05%	*-0.15%*
	3	87872.15	87770.81	87867.07	0.12%	0.01%	*-0.11%*
	4	117166.05	117027.65	117155.44	0.12%	0.01%	*-0.11%*
incdec	1	130178.51	130076.75	130398.97	0.08%	-0.17%	*-0.25%*
	2	260348.80	260107.27	261219.07	0.09%	-0.33%	*-0.43%*
	3	390502.60	390174.95	392280.71	0.08%	-0.46%	*-0.54%*
	4	520710.66	520502.35	523519.11	0.04%	-0.54%	*-0.58%*
read64	1	1226.54	1172.53	1329.12	4.40%	-8.36%	*-12.77%*
	2	2449.96	2352.69	2349.90	3.97%	4.08%	0.11%
	3	3658.43	3516.53	3492.49	3.88%	4.54%	0.66%
	4	4724.87	4595.07	4487.93	2.75%	5.01%	2.27%
write64	1	1428.53	1377.21	1480.05	3.59%	-3.61%	*-7.20%*
	2	2857.18	2734.81	2750.66	4.28%	3.73%	*-0.55%*
	3	4277.76	4000.89	4135.32	6.47%	3.33%	*-3.14%*
	4	5686.34	5299.80	5380.73	6.80%	5.37%	*-1.42%*

namespaces and loading the appropriate data structures can create significant overhead for the system. For the virtualized environment, context switches for the guest kernel cause

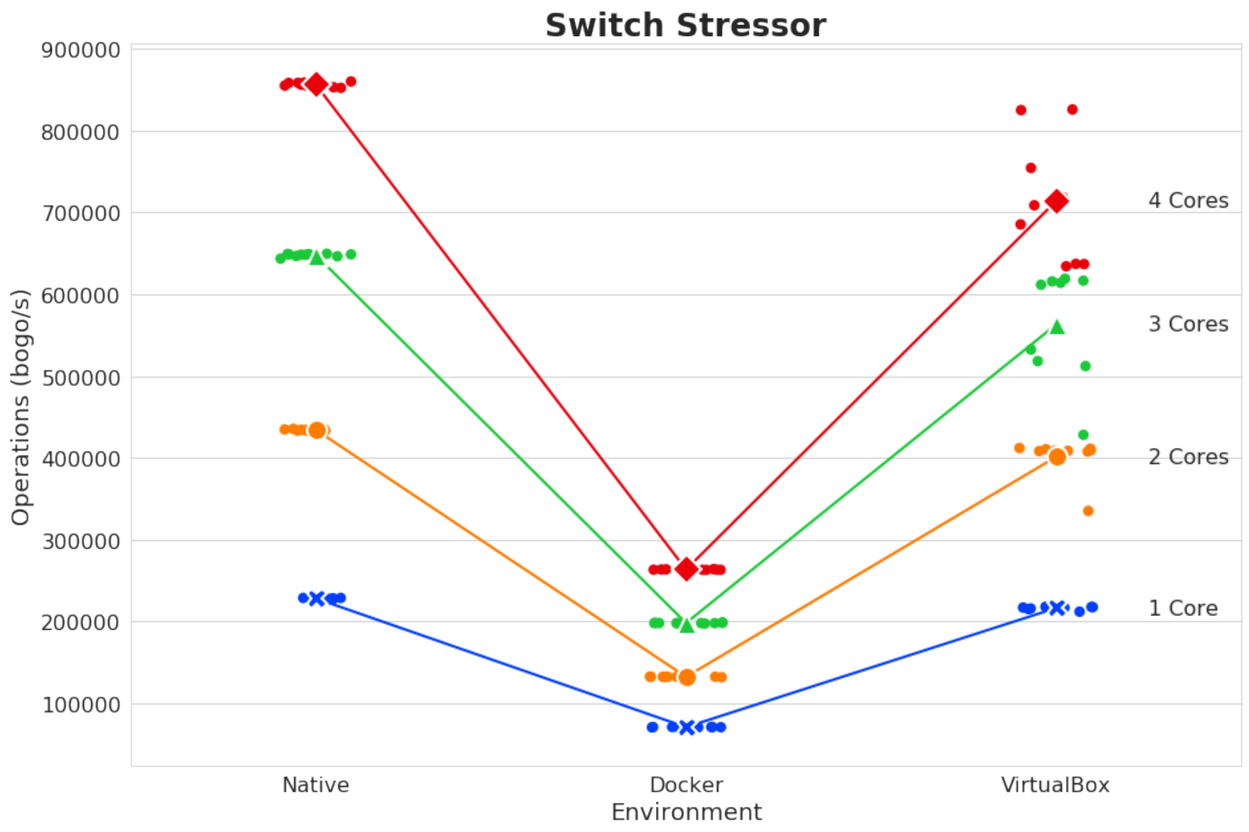


Figure 6.27: Context switching stressor results - This shows the results of the *switch* stressor executed in different isolation environments. This test forces context switching to occur by sending messages to child processes through pipes. For this test, the container environment incurred a massive overhead (around 70%) versus the native system performance. The virtual environment fared much better and had a 16% maximum overhead for a quad-core configuration. Table 6.17 shows the results of the *switch* stressor tests.

overhead in terms of the hypervisor. The guest kernel would schedule the process to execute on a virtual CPU and the hypervisor would then schedule the process on a host CPU and switch that CPU to a virtualized mode. But, context switches in the guest and host environments are similar since no namespaces are used in the environment. In SDR applications with a large number of threads (and therefore more context switching), overhead from the context switching could significantly affect the overall waveform throughput.

Table 6.17: Context switching stressor performance results - This table shows the performance of the context switching stressors (*switch* and *context*) in different isolation environments. The goal of these tests was stressing the kernel performance through forcing context switches between threads and processes. Both the container and virtual machine environments show significantly higher performance overheads than the CPU and memory stressors. Specifically for the *switch* stressor, the container environment incurs a 70% loss in performance compared to the native performance. In all tests, the virtual machine environment out-performs the container environment.

Test		**Operations/second**			**Overhead**		
Stressor	**Cores**	**Native**	**Docker**	**Virtual**	**Docker**	**Virtual**	**Difference**
switch	1	228181.78	70903.21	216485.55	68.93%	5.13%	*-63.80%*
	2	434138.77	132341.08	401205.75	69.52%	7.59%	*-61.93%*
	3	647606.59	198102.47	562901.63	69.41%	13.08%	*-56.33%*
	4	855894.06	263420.09	713725.88	69.22%	16.61%	*-52.61%*
context	1	1523.73	1362.69	1378.25	10.57%	9.55%	*-1.02%*
	2	1530.35	1374.76	1402.31	10.17%	8.37%	*-1.80%*
	3	1530.39	1376.95	1403.44	10.03%	8.30%	*-1.73%*
	4	1532.51	1381.67	1401.78	9.84%	8.53%	*-1.31%*

Split Flowgraphs

Since the ultimate goal of our defense-in-depth architecture is splitting a monolithic SDR waveform into multiple isolated segments, the final set of performance tests focused on the impact performance of the split flowgraphs. The overhead of a SDR waveform using a defense-in-depth architecture will be unique to that application based on the chosen isolation and IPC methods. However, the test results shown below do help provide a general understanding of how this architecture will affect application performance.

For these tests, the *Null_Test* and *GFSK_Loopback* flowgraphs used in previous tests were divided into separate sub-flowgraphs and these sub-flowgraphs were connected using the IPC methods described earlier in Section 6.3.4. Flowgraphs were tested in the native environment to measure the overhead of the added IPC channel and also in separate container environments to measure the overhead of both the added isolation and IPC channels. Also, all tests were executed with 4 active CPU cores that were not hyperthreaded. Setting up the shared IPC channels was straightforward since the container environment shares the same host kernel and can also share parts of the filesystem. Additional blocks were also written for the GNU Radio flowgraph to implement the domain socket and named pipes IPC mechanisms in the flowgraph.

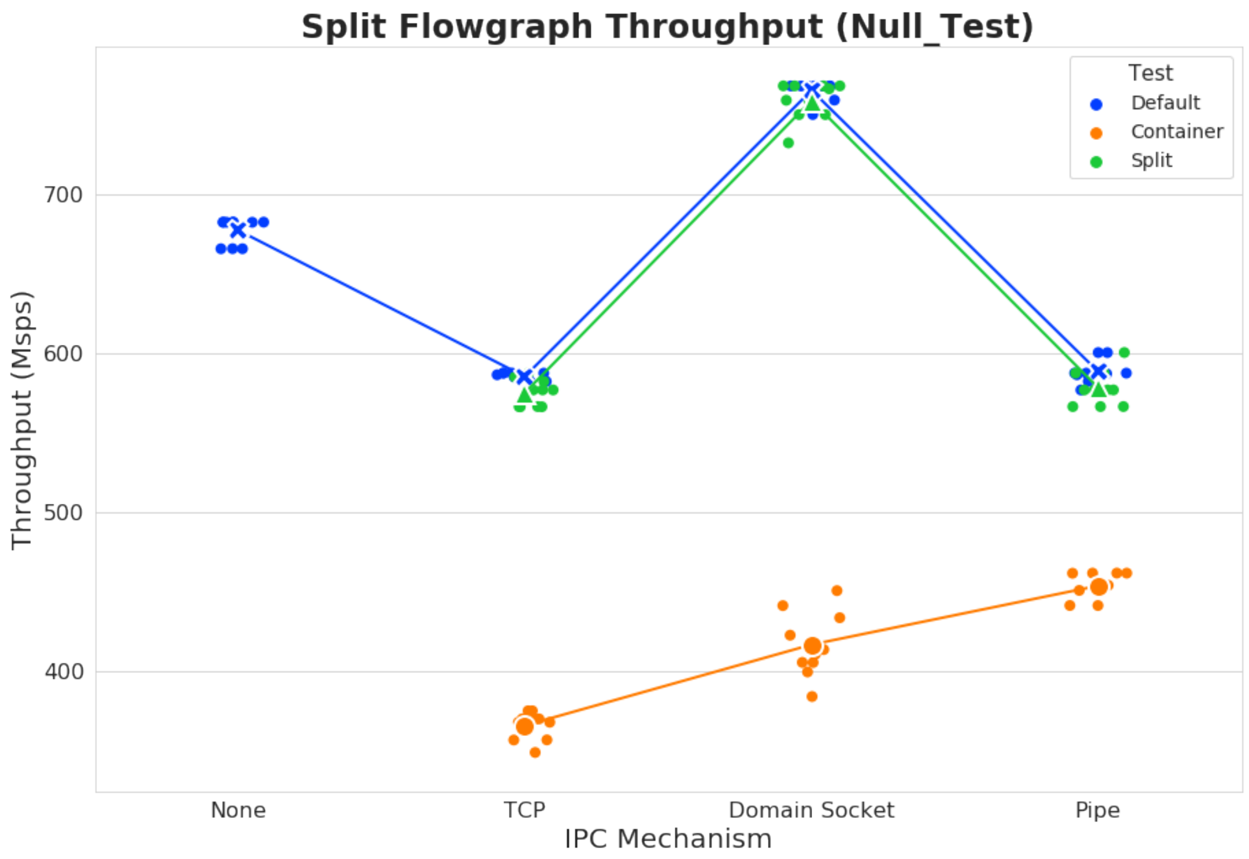


Figure 6.28: Split Null_Test flowgraph performance - This plot shows the throughput comparison of the *Null_Test* flowgraph in a split configuration (multiple processes) with different IPC mechanisms used to connect the sub-flowgraphs. The left-most category (*None*) shows the results for the default monolithic flowgraph with 2 extra copy blocks added to mirror the additional GNU Radio buffers added with the IPC blocks in the split flowgraphs. The colors correspond to how each test is launched and is described in Section 6.3.4. The blue and green lines show indicates tests executed in the native environment and orange indicates tests with each flowgraph in a separate container. These results are also shown in Table 6.18.

The results for the split *Null_Test* flowgraph are shown in Figure 6.28 and Table 6.18. Here, the performance of the split flowgraph configurations is compared to the performance of the original monolithic flowgraph (IPC is *None*). The monolithic flowgraph included two additional copy blocks added to the flowgraph which mirrored the additional blocks required due to the IPC mechanisms. This ensures the number of GNU Radio buffers allocated is similar between all tested flowgraphs.

Table 6.18: Split *Null_Test* flowgraph performance - This table shows the throughput of the *Null_Test* flowgraph in a split configuration with different IPC mechanisms connecting the flowgraph (TCP, Domain Sockets, and Pipes). These tests measure the impact of splitting the flowgraph into separate segments using the defense-in-depth approach and compare this to the native performance. In this case, the *Null_Test* flowgraph suffers a very large performance loss when executing as split flowgraphs in separate containers. Using Pipes to connect the separate flowgraphs provided the best performance out of the tested IPC mechanisms.

Test			**Overhead**	
IPC	**Environment**	**Throughput (Msps)**	**Overall**	**Container**
None	Default	677.50	-	-
TCP	Multiprocessing	585.39	13.60%	-
	Separate Processes	575.15	15.11%	-
	Separate Containers	365.78	46.01%	37.51%
Domain Socket	Multiprocessing	765.52	*-12.99%*	-
	Separate Processes	759.10	*-12.04%*	-
	Separate Containers	416.69	38.50%	45.57%
Pipe	Multiprocessing	588.42	13.15%	-
	Separate Processes	578.29	14.64%	-
	Separate Containers	453.76	33.02%	22.89%

Considering only the IPC overhead of the split flowgraphs, the TCP and named pipes methods showed 13-15% lower performance when compared to the monolithic flowgraph, and interestingly, the domain socket performance exceeded the monolithic native performance by 12%. When executing each sub-flowgraph in a separate container, all three tested IPC methods showed a large overhead (33-46%) with the named pipes mechanism showing the best performance (33% overhead) of the three. Also, the container performance using named pipes shows around a 26% overhead when compared with the original two *copy* block *Null_Test* flowgraph performance executing in the container environment (shown back in Table 6.7.

The *Null_Test* flowgraph does not do any signal processing and is really only stressing the memory performance of the GNU Radio flowgraph in separate containers and connected over an IPC channel. These flowgraphs showed high overhead when compared to the single monolithic flowgraph in the native and container environments, but still out-performed the single flowgraph executing in the virtual machine environment. The tests with the *GFSK_Loopback* shown next give a better indication of the overhead for an actual waveform using the defense-in-depth architecture and using container isolation.

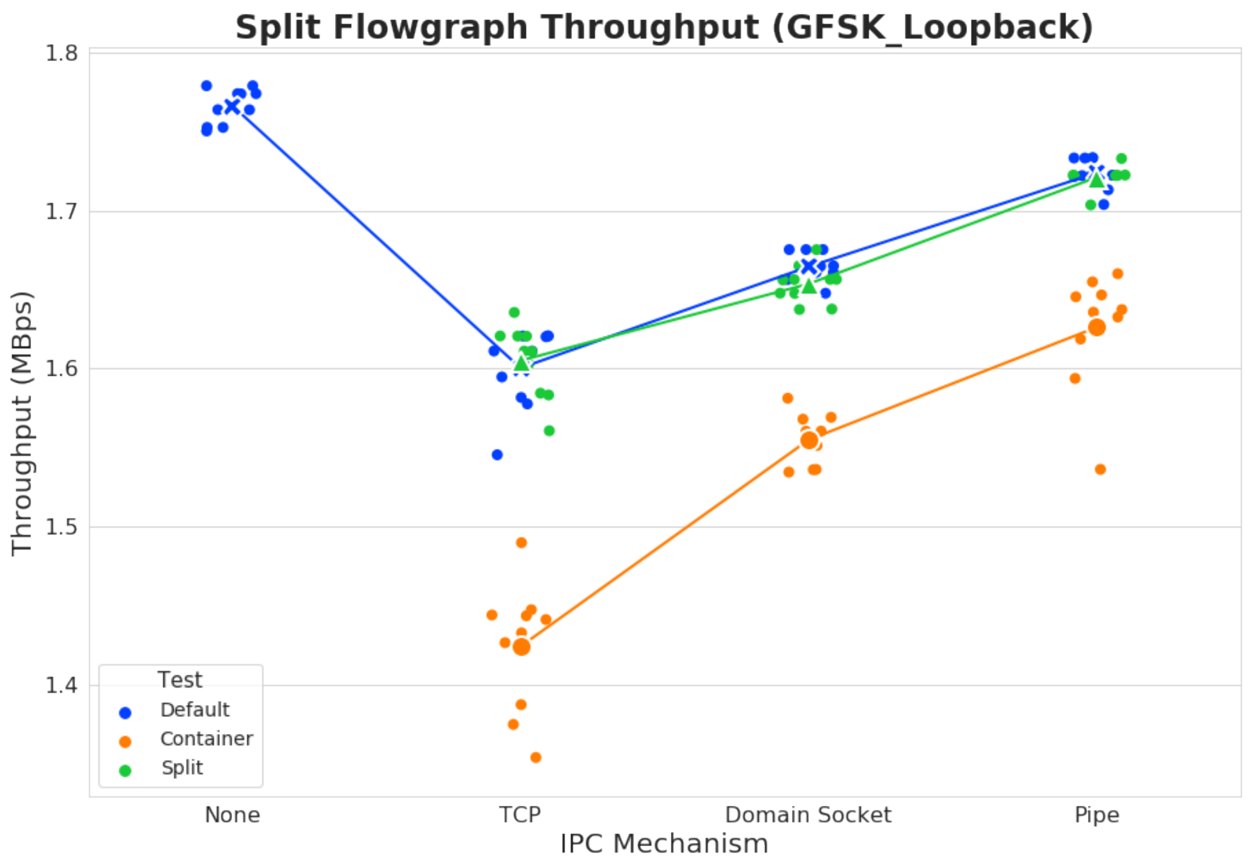


Figure 6.29: Split GFSK_Loopback flowgraph performance - This plot shows the throughput comparison of the *GFSK_Loopback* flowgraph in a split configuration (multiple processes) with different IPC mechanisms used to connect the sub-flowgraphs. The colors correspond to how each test is launched and is described in Section 6.3.4. The blue and green lines show indicates tests executed in the native environment and orange indicates tests with each flowgraph in a separate container. These results are also shown in Table 6.19.

The results for the split *GFSK_Loopback* flowgraph are shown in Figure 6.29 and Table 6.19. These tests were executed in the same fashion as the *Null_Test* tests shown previously. Compared to the *Null_Test* results, the overhead in the split *GFSK_Loopback* flowgraph due to the IPC channel is lower and overall less than 10%. The named pipes IPC method again show the best overall performance with less than 3% overhead for the multi-process native tests. Moving the sub-flowgraphs into separate containers does add more overhead, but the loss in performance is still less than 8% for this configuration. Like the *Null_Test* flowgraph, all of the containerized, split flowgraph tests still out-performed the virtual machine tests for the original monolithic flowgraph shown in Table 6.9.

Table 6.19: Split *GFSK_Loopback* flowgraph performance - This table shows the throughput of the *GFSK_Loopback* flowgraph in a split configuration with different IPC mechanisms connecting the flowgraph (TCP, Domain Sockets, and Pipes). These tests measure the impact of splitting the flowgraph into separate segments using the defense-in-depth approach and compare this to the native performance. Depending on the IPC mechanism used, this flowgraph can incur a large performance overhead when executing as separate flowgraphs in different containers (around 20% for TCP). Using Pipes to connect the separate flowgraphs provided the best performance and only resulted in an 8% loss in performance.

Test			**Overhead**	
IPC	**Environment**	**Throughput (MBps)**	**Overall**	**Container**
None	Default	1.77	-	-
TCP	Multiprocessing	1.60	9.40%	-
	Separate Processes	1.61	9.12%	-
	Separate Containers	1.42	19.37%	11.00%
Domain Socket	Multiprocessing	1.66	5.78%	-
	Separate Processes	1.65	6.34%	-
	Separate Containers	1.56	11.95%	6.55%
Pipe	Multiprocessing	1.72	2.38%	-
	Separate Processes	1.72	2.55%	-
	Separate Containers	1.63	7.93%	5.68%

One small difference to note is the split *GFSK_Loopback* flowgraph did not include extra *copy* blocks for the native tests for mirroring the additional GNU Radio buffers and IPC blocks that exist for the split flowgraphs. If the no IPC native flowgraph did include these extra two *copy* blocks, its performance would slightly decrease which also reduces the overhead of the split flowgraph configurations.

6.5.4 Additional Takeaways

There are several additional takeaways to consider in the context of all of the different test results.

1. On average, all of the GNU Radio flowgraphs tested on single-core virtual machines out-performed the same flowgraphs executing within a container. (The multi-core

container flowgraphs usually out-performed the virtual machines.)

One possible explanation for this behavior on single core configurations is indicated by the overhead of the context switching stressors within containerized environments. These stressors surprisingly showed that the overhead for context switching in a containerized system is greater than in virtualized environments; the *switch* stressor showed a significant (almost 60% greater overhead on average) for containers over virtual machines. Based on how containers are implemented, there is additional overhead involved with the kernel scheduling containerized processes and threads due to the separate namespaces and data structures that need to be processed. In a virtualized environment, the guest kernel does not have this overhead of switching between namespaces (since no containers are executing in the guest). If a virtualized kernel is allowed to execute without significant interruption by the host hypervisor, then the performance could more closely match that of the native system.

GNU Radio flowgraphs use a thread-per-block scheduler to better utilize multi-core systems. So, rapid context switches between the threads in the flowgraph occur which could cause the additional overhead for the container tests. Reducing the amount of context switching within a flowgraph could help improve the overall performance of the waveform.

2. The GNU Radio framework is not well optimized for executing within multi-core virtual machines.

   While the performance of flowgraphs in virtual machines does slightly increase as the number of cores increases, this increase is minimal in comparison to the native and container environments. Even though there is minimal parallelism that can be achieved with the *Null_Test* simple flowgraph, the native and container tests do show a 4x performance increase moving to a dual-core configuration. Increasing the number of active cores beyond this does not significantly change the performance since there is a limited number of threads to execute.

   But, for the virtual machine environment the overhead for the multi-core systems greatly increases with additional cores. The most significant increase in overhead is with the *Null_Test* flowgraph (with no copy blocks) which jumps from practically no overhead on a single core virtual machine to 70-80% overhead for two or more cores. The flowgraph's performance never significantly increases past the baseline performance of a single core for multi-core tests. On a containerized system the highest overhead for the same flowgraph is 12%. When the flowgraph does have multiple copy blocks, there is a performance increase when adding additional cores, but this performance increase is minimal when compared to the increase of the native and containerized environments. This increase is also due to the additional threads in the flowgraph, so there is some advantage with multi-core systems. A similar behavior is observable in the *GFSK_Loopback* and *Bytes_Loopback* flowgraphs where the virtual machine environment sees 15-30% overhead for multi-core configurations while the containerized version shows only 1-2% overhead for the same flowgraph.

There is likely a major bottleneck within the GNU Radio framework that is not optimized for execution in a virtual environment, which causes a significant loss of performance compared to containers. If the framework could be better optimized for execution within a virtual machine environment, the overhead of using virtual machines may be significantly reduced and more comparable to the performance of the containerized version.

3. Even though GNU Radio applications have a higher overhead when executing with multi-core virtual machines, the overhead for single cores is relatively small. The main intent of our security architecture is isolating every component of the waveform, either individually or in a small group, into separate environments. If the required resources for a block is relatively small, isolating this block within a single core virtual machine may not impact the overall throughput of the waveform that significantly. As the waveform is divided into smaller components, the overall parallelism of each partial flowgraph is reduced, so the overhead of the multi-core environments may become less critical. The single-core environments incurred a relatively low overhead, so executing multiple partial waveforms in single-core virtual environments may be a feasible solution with little overhead compared to the native environment.

4. There could be multiple types of optimization that can be applied to individual systems to reduce the overhead of the defense-in-depth solution. For example, multiple other IPC mechanisms exist that have not been tested; some of these may have even less overhead.

6.5.5 Conclusions

There are a few conclusions that can be drawn from the results of our performance tests. First, the performance overhead for implementing SDRs using the defense-in-depth architecture and isolating components within the waveform can be minimal due to the isolation itself. Most of the LiquidDSP and stressor tests showed little to no performance overhead for either the container of virtual machine environments. The GNU Radio container tests with the single flowgraphs also showed relatively low performance overhead when executing in the containerized environment. Since the main intent of the defense-in-depth architecture is segmenting waveforms into smaller isolated environments, the IPC connections between these environments will also add some overhead to the system. Our example test waveforms using this type of configuration also did not show significant performance loss with IPC mechanisms like named pipes. These results validate that the defense-in-depth architecture for SDRs is a feasible approach for developing secure systems, and production systems should be designed in this manner moving forward.

Second, the virtual machine environment does significant overhead to the performance of

GNU Radio flowgraphs in multi-threaded/multi-core applications. Virtual machines provide significantly more isolation overall than containers due to the guest environments, so the extra overhead is expected. However, other test results (multi-core containers) indicate that GNU Radio is not optimized for virtual machine environments which suggests there are be methods to optimize how a flowgraph is executed within a virtual machine and significantly improve performance for multi-core configurations. If developing a high-assurance radio system, this is still a feasible isolation option even with the higher overhead since security is the main design requirement of the system.

Third, each unique system will require different trade-offs and configurations in terms of isolation or performance when implementing a defense-in-depth architecture. As more wireless systems transition to SDR implementations, security becomes a major concern, so some type of isolation should always be implemented within waveforms. For example, virtual machines likely create too much overhead to be acceptable security mechanisms for applications requiring very high-performance. In these situations, containers provide a more lightweight solution that could add security to the application but still achieve high-performance. On the other hand, security is a primary requirement for high-assurance systems, so virtual machines provide a better solution even if the resulting performance overhead is very high.

Sometimes a hybrid approach can provide the best overall solution and balance of security and performance. For example, containers could be used to isolate lower security risk components that require higher performance and throughput, and virtual machines could isolate the higher risk components within the system. Typically, the highest risk components within a waveform are the higher layer network blocks in the OSI stack. Lower level blocks like the physical layer require more overall performance but also can be less of a security risk since these layers process the signal and samples rather than data.

6.6 Summary

One of the main issues to consider when developing a SDR application with the defense-in-depth architecture is the trade-offs between overall processing performance and security of the system. As more isolation is added within the waveform itself, the amount of processing overhead increases which reduces overall performance. We specifically focused on measuring the maximum throughput of different waveforms in both a containerized and virtualized environment to understand and characterize the overhead due to each. Depending on the requirements for a specific application, the overhead for some environments may prevent it from being a feasible solution. Some performance loss may be acceptable for high-risk components in order to gain the additional security from the isolation, but not all software radio applications can tolerate the loss in overall performance.

In this chapter, we tested two common isolation environments, container (Docker) and virtual machines (VirtualBox), by executing several different flowgraphs and stressor utilities in order to measure the maximum throughput. Our results show that in most cases, the performance impact of the isolation environment can be minimal in result in a small percentage (less than 5%) loss due to overhead. However, not all applications are optimized for some environments and, in this case, we showed that some GNU Radio flowgraphs had additional bottlenecks that significantly limited their performance in virtual environments. We also tested the performance of split flowgraphs that were connected with different types of IPC to better simulate an application using the defense-in-depth architecture. Generally speaking however, the overall performance overhead of using isolation environments for software radio waveforms can be minimal, so the defense-in-depth architecture should be the basis of communication systems moving forward.

Chapter 7

Example Implementations

In this chapter, we present some of the challenges to integrating the defense-in-depth model into existing frameworks and present several methods of building a waveform based on our model and using the GNU Radio framework and Linux Containers. We focus on the GNU Radio platform since it is one of the most popular open source frameworks can be used to implement many different types of radio applications. Containers provide the main isolation mechanism for each of these approaches mainly because it provides additional security with a low amount of additional performance overhead.

7.1 Applications

Cloud Radio Access Networks

The main concept behind the Cloud-RAN is separating the monolithic base-station into smaller services that can be co-located into a centralized cloud infrastructure rather than being geographically separated. This allows the system to better handle demand and be able to scale itself to meet increasing network traffic or high user demand during peak hours of congestion on the network or RF spectrum. Since all of the core network components are all co-located, this also greatly reduces the latency and increases the total bandwidth available to the core network itself. Cloud-RANs, specifically, are a perfect candidate for this architecture because the base-stations are already virtualized and executing within a centralized data center. The key to building a secure C-RAN network is ensuring that each virtual base-station is fully isolated from the underlying environment. We show an example implementation of a virtual base-station and C-RAN in Figure 7.1.

Virtualization Ground Station

The concept of a virtualized ground station is a key component to supporting future Ground-Stations-as-a-Service. The goal with this concept is providing an infrastructure and framework that allows customers to develop software waveforms for various spacecraft that can execute on the virtualized ground station. No matter what physical hardware actually exists at a specific location, the interface would all be the same.

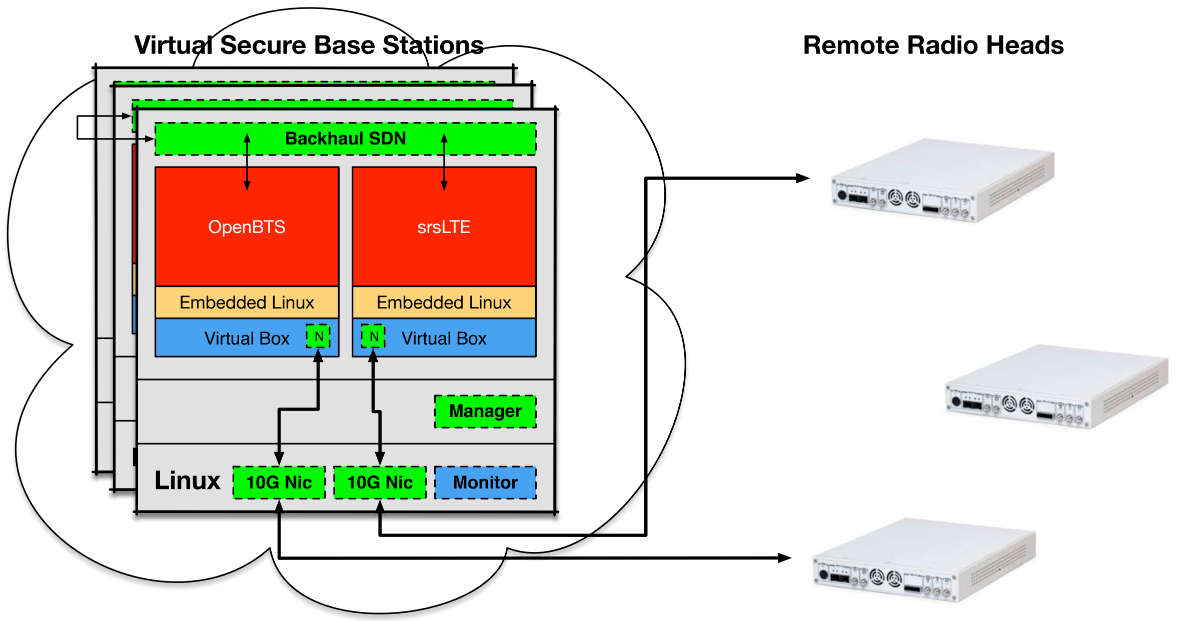


Figure 7.1: Example CRAN (Cloud Radio Access Network) architecture. Example of a software radio system in a Cloud Radio Access Network built using the defense-in-depth approach for secure SDRs.

Core to this concept is allowing third-party waveforms to execute on ground station hardware. Our defense-in-depth model is a perfect approach for ensuring that vulnerabilities or malicious code that exists within the third party waveforms are unable to affect either the underlying platform or other customer's applications and data. By isolating each application within its own domain, it is also possible that multiple customer's waveforms can execute at the same time for a specific pass completely independently of each other.

This can be further extended into a model like the Cloud-RAN to develop distributed ground stations. In this case, the main computing resources are centrally located, while the physical antennas and radio hardware would be distributed geographically.

7.1.1 Example Implementation

The intent of this defense-in-depth architecture is separating all of the components of a communications system into isolated environments for added security. However, the specifics of how components are separated within an implementation is dependent on many factors such as the security and performance requirements. As such, applications developed using this architecture can differ heavily between implementations.

A simplistic form of this architecture is executing a radio waveform as a monolithic application within a single isolated environment. A downside to this methodology is an exploit of a component in the waveform can compromise the entire instance. However, the isolation environment would prevent the attack from compromising the underlying host and therefore other radio applications.

C-RAN architectures already follow this simplistic version since each base-station is virtualized in a datacenter. An example implementation of a virtual base-station is shown in Figure 7.1. In this example implementation, two software radio based, cellular network applications (srsLTE [23] and OpenBTS [22]) are executing on a host with Oracle's VirtualBox platform [14] providing isolation between the applications (with a hardened Linux kernel as the guest). Each environment is bridged through VirtualBox's built-in network stack to Ettus Research Universal Software Radio Peripherals (USRPs) [75] that can support optical connections back to the host system. VirtualBox's built-in network stack bridges the VM's network interface to the host device allowing the software radio to access the hardware. The network stack also allows for creating host-only networks that allow the application environments to communicate together or connect to other services. A firewall can monitor traffic between the software waveforms on the virtual machines as well as monitor network traffic to the remote radio head devices.

However, this simplistic approach does not accomplish the full intent of the architecture which is to isolate all components of the radio into separate environments. Another example, based on the example application used to demonstrate the effect of an exploit on a waveform (shown in Figure 4.5), is more secure implementation where components like the *frame_sync* and *router* blocks are placed within their own virtual environments. Since the source block for this flowgraph only handles transporting samples, it is a lower risk and can execute in a container. The same VirtualBox network stacks could implement the IPC functionality used to connect the *frame_sync* and *router* blocks together. In this example, if the exploit demonstrated earlier were attempted, it would only affect the virtual environment executing the *router* block rather than the full waveform.

7.2 Challenges

The main challenges to building a waveform using our defense-in-depth methodology deal with managing the now distributed nature of the waveform. Specifically, this includes process management, data flow management, and scheduling. Since different components of the waveform are now executing in isolated environments, there must be some mechanism that exists to start each portion of the application, as well as, providing the connections between components.

GNU Radio versus REDHAWK

For our implementations provided in this chapter, we chose to use the GNU Radio framework as the basis for our applications. This raises the question of why it makes sense to choose GNU Radio instead of choosing the REDHAWK framework, which is already built for distributed systems. Our main motivation for choosing GNU Radio over REDHAWK is based on the underlying scheduler models and data flow models used by the REDHAWK framework versus GNU Radio. This is the biggest difference between the two frameworks.

REDHAWK applications are distributed, by nature, and are mainly designed to execute on systems that provide a significant amount of computing resources, such as a datacenter. GNU Radio, on the other hand, is designed to execute a single monolithic flowgraph that could be executed on resource-constrained, embedded systems. To that extent, both frameworks use very different scheduler models: REDHAWK utilizes a process-per-block model and GNU Radio uses a thread-per-block model.

- **Process-per-block**: This is the main scheduling model used for the REDHAWK framework. Each component within a REDHAWK application is executed as its own process that is launched by a *Device Manager* process that is running on each *Node* of a REDHAWK *Domain*. This model has the advantage that a failure in a specific component of the system will only crash its respective process and will not affect the overall executing application.
- **Thread-per-block**: This is the scheduling model used by the GNU Radio Framework, and is typically the default model used for SDR applications. In this model, a waveform or application runs as a single process on the system and executes separate threads for each component or block in the waveform, which are created by the main thread and can be scheduled independently by the underlying operating system Unlike the process-per-block model, if a fault occurs in one of the child threads in the system (usually a component), the operating system will abort that process which causes the entire application to crash.

Since REDHAWK is already designed to targeted distributed systems, it would be rather straightforward to build a waveform using the defense-in-depth approach. In fact, the process-per-block model already provides a decent amount of isolation between components in an executing waveform. As we mentioned before, if a fault or exploit occurs within a specific component, only that specific component will crash rather than the entire waveform. However, if an exploit such as the stack buffer overflow exploit existed in a REDHAWK implementation, the attacker could still easily compromise the system. The process-per-block isolation will not provide a significant enough isolation to contain that type of attack. Improving the amount of isolation within a REDHAWK application can be accomplished in a variety of ways. Each node within the application can be defined as a different isolated security zone and can be executed either in a container, a virtual machine, or another physical machine altogether. Since REDHAWK already utilizes the CORBA framework to connect different components together, the data flow simply relies on the network connections between the different containers or machines that are executing each node.

Depending on the sensitivity level of each component or its risk factor, different levels of isolation can be achieved depending on the specific implementation for that node. Executing each node on an individual, physical system provides the most isolation, while nodes residing containers would provide the least. The tradeoff to isolation is performance; if nodes executing on physically separate systems would result in less throughput and higher latency for the entire application. A second tradeoff to consider in this case is the available resources of each system executing a specific node. If multiple nodes are executing on the same system in either virtual machines or containers, there may be additional computing overhead with scheduling components to execute. Components and nodes executing by themselves on a single physical system may not exhaust available resources and the limiting factor for performance would be the specific component's throughput. Components on a shared node may not be able to execute at their full performance due to overhead from the shared isolation mechanisms (i.e. multiple virtual machines).

The biggest issue with choosing the REDHAWK framework is that it is designed as a distributed system and not suitable for embedded systems. Part of the goal for our research is to apply this model to waveforms that would execute in an embedded environment, so computing resources are at a premium. With REDHAWK, one of the biggest issues is the use of the CORBA framework; each component in the system is interconnected to other blocks over the network stack. Compared to GNU Radio, this provides a significant performance overhead compared to GNU Radio's shared buffers model. Since each GNU Radio block executes in a thread rather than a process, the scheduler simply creates buffers that are shared between the different blocks. For our implementations, GNU Radio provides the best approach, because we can use a process-per-zone scheduling model where the blocks within a specific security zone are all executing in the same process. If a block is a very high security risk, it can be executed alone in a process, which would mirror the REDHAWK scheduling model. This methodology gives the advantage that low-risk blocks in the same security domain can execute with the lowest amount of overhead between them, which reduces the overall latency of the system and increases the overall performance.

In this section, we present three different methods of implementing a waveform based on the defense-in-depth model using the GNU Radio framework. Each method has its own tradeoffs concerning scheduling and data flow, which are discussed as well.

7.2.1 Networking Based

Our first method uses a similar design philosophy to that of REDHAWK. While each security zone runs in a single process rather than process-per-block, the data connection between each zone is handled through the network stack. Using this method requires no additional changes to the core of the GNU Radio framework as GNU Radio already supports multiple methods for connecting flowgraphs through the network stack, such as Socket PDUs, UDP, TCP, and ZeroMQ These networking blocks can be configured in GNU Radio to execute in either client or server mode, which defines their behavior once the flowgraph starts. A container would be started for each specific section of the flowgraph and the networking blocks would be configured to connect using the container's bridge network connection.

Advantages

- The simplest method to construct a defense-in-depth waveform using the existing blocks in GNU Radio.
- Requires no modification to the GNU Radio runtime.

Disadvantages

- Connections between different zones use the networking stack similar to REDHAWK, which would increase the amount of latency and overhead in the system. Each connection between zones requires at least a source and sink networking blocks which means there are multiple memory copy operations between buffers in the waveform.
- Requires splitting an existing flowgraph and adding new network blocks to each separate flowgraph.
- Requires a user to manually configure the network connections at design time unless an external script provides parameters to the flowgraph at startup.
- Moving a block into a separate zone requires adding several new networking blocks between it and any downstream blocks.

To simplify the process of starting an application using this method, an external script can be used to manage much of the process of creating and starting containers and connecting specific waveforms together. The main disadvantage of this method is using GNU Radio's networking functionality between flowgraphs executing in different security zones. A variant of this method utilizes pipes or domain sockets rather than the full network stack. Container applications can set up a temporary file system that can be shared by multiple containers. Each container can use the shared file system to create the pipe or domain socket that can then be opened by the downstream flowgraph. Using pipes rather than the entire network stack will improve the overall performance. However, this variant still requires additional source and sink blocks in each flowgraph which increases the latency over the non-segregated waveform. For example, each of the additional blocks in the flowgraph must perform a memory copy operation to move data from the GNU Radio buffer to the output network buffer or pipe. However, this is still preferable to REDHAWK's process-per-block model since it does not require the networking connections between *every* block in the waveform.

7.2.2 Custom Buffers Based

Our second method does require some modification to the GNU Radio runtime to enable blocks to allocate custom buffers rather than letting the runtime itself allocate each buffer for a block. We have previously done some work to enable this support in GNU Radio mainly to better support using Graphics Processing Units (GPUs) to accelerate the signal processing functionality. An example of how the custom buffer feature was designed for GPUs is shown in Figure 7.3 and Figure 7.2 shows an example without custom buffers. This method includes defining a new buffer type that uses the shared memory faculties in the operating system to share the buffer between two different sub-flowgraphs in the waveform.

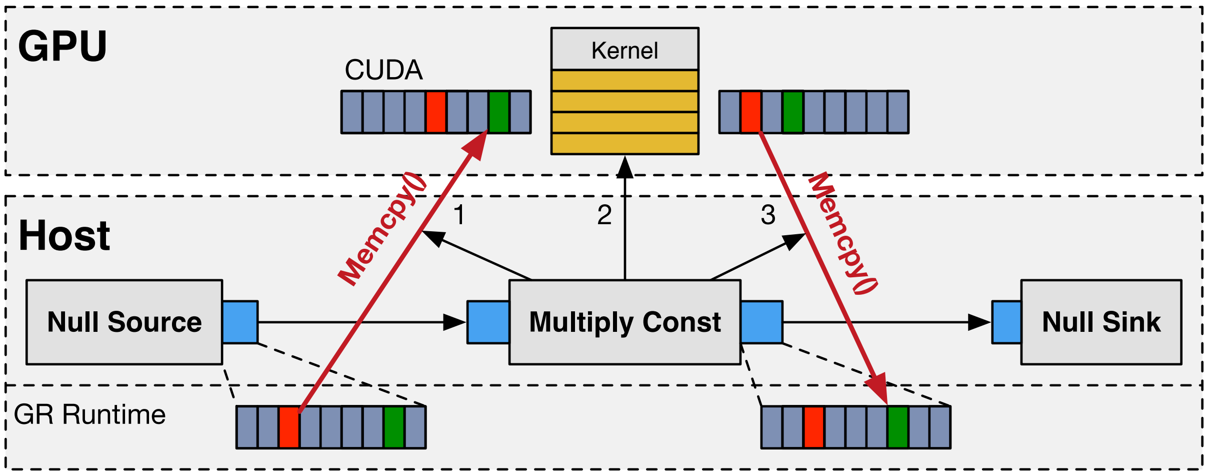


Figure 7.2: Example of a GNU Radio flowgraph without custom buffer support for GPUs - Each GPU based block must copy memory from the GNU Radio buffer to the GPU buffer before execution. The network based method for a secure GNU Radio waveform would require similar memory operations.

With only the custom buffer support changes in the GNU Radio runtime, the rest of the functionality for this method can be included in a separate out-of-tree module. The custom buffer method can be integrated into a flowgraph in two major ways. If the flowgraph is built using custom blocks, then the developer can simply create and pass the custom buffer to the runtime when the flowgraph is constructed. However, if the flowgraph is using only built-in blocks, a similar mechanism to the network based method would need to be used. In this case, a source and sink block that supported the shared memory blocks could be added to the flowgraph for each external connection. The advantage of this method over the networking method is the reduced number of memory operations during execution. In

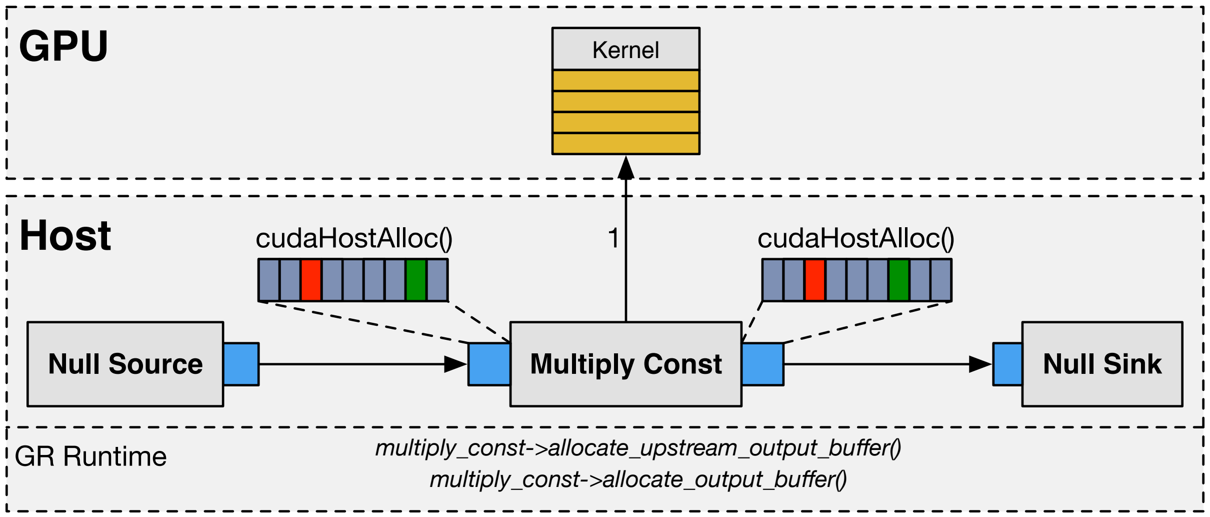

GPU
Kernel
Host
cudaHostAlloc()
1
cudaHostAlloc()
Null Source
Multiply Const
Null Sink
GR Runtime
multiply_const->allocate_upstream_output_buffer()
multiply_const->allocate_output_buffer()

Figure 7.3: Example of a GNU Radio flowgraph with custom buffer support for GPUs - Each GPU based block in the waveform can allocate memory that is shared with the GPU, avoiding additional memory copy operations from GNU Radio owned buffers to the GPU owned buffers.

our previous GPU work enabling custom buffers, a block could specify whether it would allocate and own its upstream or downstream buffer. Because of this, a source block could allocate its downstream buffer as the shared memory buffer and a sink block could allocate its upstream buffer as a shared memory buffer. In that situation, the corresponding blocks in the flowgraph would be reading and writing directly from the shared buffers, so no additional memory copy operations are required.

Advantages

- Reduces the amount of memory copy operations over the network based approach. Upstream and downstream blocks would be able to directly access the shared buffers.
- Once custom buffer support is enabled and the blocks are created, it can be straightforward to connect multiple sub-flowgraphs.

Disadvantages

- Does require some additional modifications to the GNU Radio runtime.
- Requires additional blocks to exist in each sub-flowgraph for connecting to the upstream and downstream zones.

- Requires splitting an existing flowgraph and adding new network blocks to each separate flowgraph.
- Requires a user to manually add blocks to configure the specific zones.
- Moving a block into a separate zone requires adding several new networking blocks between it and any downstream blocks.

7.2.3 Shared Memory Based

The final method for containerizing GNU Radio builds upon the previous method. In this case, we heavily modify the GNU Radio runtime such that it can directly allocate shared memory blocks rather than requiring additional separate blocks. The GNU Radio Companion can also be modified to directly allocate each sub-flowgraph within its own security zone and setup and allocate the shared buffers between each sub-flowgraph. The main challenge in this methodology is handing the signaling between flowgraphs.

Advantages

- Directly allocate shared memory buffers between sub-flowgraphs.
- Reduces the amount of memory copy operations over the network based approach. Upstream and downstream blocks would be able to directly access the shared buffers.
- Does not require modification to built-in blocks or new blocks added to the flowgraph.
- Creating secure zones could be done directly in the GNU Radio Companion or the main flowgraph.

Disadvantages

- Requires significant modification of the GNU Radio runtime and scheduler so it is aware of the shared memory blocks and the other schedulers running in child processes.

7.3 GNU Radio Defense-in-Depth Framework

One of the byproducts of the test framework presented in the previous chapters is that it also serves as the basis for developing a framework for deploying GNU Radio applications with our defense-in-depth architecture. The primary goal for this security framework was providing a nearly identical interface to developing flowgraphs using the defense-in-depth model as

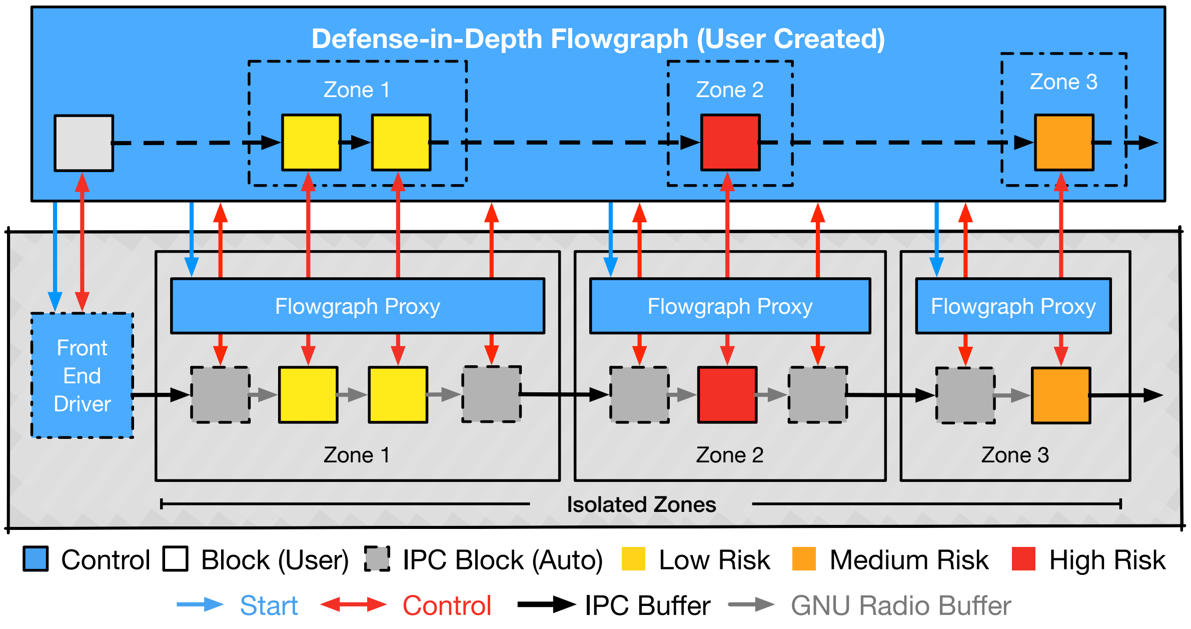


Figure 7.4: Defense-in-Depth GNU Radio Framework - This is an example of a containerized GNU Radio waveform using our framework. In this case, the Flowgraph controller is responsible for launching different container environments which each automatically start the waveform proxy. Once executing, the controller can then remotely build GNU Radio waveforms and automatically add the proper inter-process communication blocks for the waveform to operate properly.

normal GNU Radio flowgraphs. This helps to significantly reduce the work required to port flowgraphs to the new model. Basically, this security framework acts as an additional proxy layer to GNU Radio flowgraphs executing in an isolated environment. A diagram of the basic components of this new framework is shown in Figure 7.4.

7.3.1 Components

Our security framework utilizes a similar architecture to the test framework presented earlier. It is also developed in Python and heavily utilizes the Pyro4 package. The main components of the framework include the isolation images, environment managers, remote flowgraph proxy, and finally the local flowgraph controller, zones, and block classes.

- Similar to the testing framework, the security framework also requires pre-configured isolation environments that have GNU Radio and any other out-of-tree modules or dependencies installed in the system. The security framework, specifically the flowgraph proxy, must also be installed and configured to automatically execute once the environment is launched. When launching the environments, the security framework can dynamically configure any resource limitations that were selected by the user.

- The environment managers are effectively the same as the experiment classes from the testing framework. These consist of all the required code to start and configure the isolation environments for the flowgraph components. Each environment class manages a single type of isolation and uses the appropriate API to start each image and configure the network or device access required for the flowgraph.
- The remote flowgraph proxy is similar in concept to the worker proxy from the test framework as it allows for remotely executing GNU Radio flowgraphs. The major difference between this and the worker proxy and GNU Radio worker is that flowgraphs can be dynamically constructed and do not have to be preinstalled in the image. Blocks are added to the remote flowgraph by specifying the block's module name and passing any arguments that are normally passed to the block constructor. The remote flowgraph then attempts to dynamically load the appropriate module and allocate the requested block with the given settings. Once the block is created, other functions can be used to remotely call functions on the allocated block.
- The local flowgraph and block classes are the main classes a developer would inherit from and interact with when developing an application with the security framework. These classes serve two major roles: 1) they serve as interfaces to the remote proxies and flowgraphs and 2) manage the overall execution of the application (starting, stopping, interconnecting different zones, etc). The zones represent the executing isolation environment as serve as proxies to handle different aspects of each environment such as adding blocks or configuring the system.

  The block classes are returned by the zones or the flowgraph controller when adding new blocks to a zone and serve as proxies to the remote blocks themselves. Any function called on the local block class is automatically forwarded to the appropriate remote block and the return value passed back to the calling function. These block classes dynamically create instance methods so the block's API is nearly identical to that of the remote block; certain functions normally available in GNU Radio blocks cannot be called remotely due to their implementation. This should not be an issue for most implementations.

  Finally, the flowgraph controller is responsible for managing the overall execution of the flowgraph. This includes automatically setting up the IPC mechanisms and blocks between the different zones to connect the entire flowgraph. The specific IPC mechanism used can be defined by the user, but the user does not need to directly manage the IPC mechanisms themselves.

The security framework is intended to be very straightforward and nearly identical to use as normal GNU Radio Python flowgraphs. The only major difference between a normal GNU

Radio flowgraph and the security flowgraph is the requirement to first create zones before blocks can be added to the zone. The isolation environment may take a significant amount of time to initialize, so the setup of the security framework is overall longer than a normal GNU Radio flowgraph. Once the isolation environment is running, the framework will connect to the remote flowgraph proxy and allow new blocks to be added to the environment. Since the framework itself is implemented in Python and relies on the Pyro4 package for implementing the proxy, $C++$ flowgraphs are not supported. However, the blocks themselves can be written in either Python or $C++$ just like a normal GNU Radio flowgraph. Also, unlike the test framework, there are no configuration files. Building a flowgraph consists of creating a class that inherits from the controller flowgraph class (which itself implements the same API as a normal GNU Radio top_block, but does inherit from the top_block), adding the appropriate zones to the controller flowgraph, adding the appropriate blocks to the zones, and connecting and starting the flowgraph like a normal GNU Radio flowgraph.

7.4 Summary

In this section, we have addressed some of the challenges associated with developing a defense in depth implementation such as process management, data flow management, and scheduling. We also presented three different methodologies for implementing such a waveform using the GNU Radio framework and Linux Containers. Each of the described methods has advantages and disadvantages such as the amount of integration required in the GNU Radio framework and the overhead associated with executing separate, but interconnected flowgraphs in an isolated environment.

Chapter 8

Conclusions

8.1 Summary

As the industry shifts more toward implementing wireless communication systems completely in software rather than hardware, there are new attack vectors that exist against vulnerabilities in those systems. Many times research in wireless security is focused more on the specific wireless protocols or protection schemes rather than the radio implementations themselves. Since software radios typically have a much faster development life cycle versus their hardware counterparts, these common types of mistakes can be easily overlooked during development and exist in production systems. While many defenses have been developed against common vulnerabilities and are implemented in desktop systems, many embedded systems either do not have the same defenses or they are improperly configured leaving them vulnerable to attack. Secure coding strategies is the most effective method to combat these vulnerabilities, but it is difficult to catch every bug that may lead to a vulnerability in a very complex system,

In this dissertation, we discuss how these vulnerabilities can be used to attack a host system and we demonstrate an example exploit of a vulnerability in a software radio implementation simulating a basic IoT sensor node using GNU Radio. A correctly constructed frame transmitted by an attacker would allow them to inject shellcode into the executing waveform and completely hijack its execution context. We also present several recent examples of hackers attacking the wireless firmware on the co-processors of embedded and mobile devices and using these exploits to ultimately control the host platform.

We then present a survey of the different types of security threats that exist against software radio as well as some of the existing research in securing software radios. Much of this research focuses on securing the waveform download process, securing the radio configuration, or enforcing the correct policy for the specific system. Some research has focused on building models for secure radios, but these models fail to address the possibility of a waveform being compromised.

We then present our defense-in-depth architecture for building secure software radios which uses separation mechanisms to isolate every component within the radio. The goal of this architecture is to provide a minimal, secure core that the rest of a software radio can be built upon. By reducing the core of the system to a minimal Trusted Computing Base and isolation every component in the system, the effectiveness of an exploit against a particular component in the system will be greatly reduced.

Our model introduces a new security plane to systems which provides the isolation mechanism for all other components as well as some device and policy management, and the inter-process communication for connecting components within the system. The specific isolation mechanism will vary due to performance requirements and available hardware support. Examples of these mechanisms include virtualization, containers, sandboxing, and microkernels. The control logic is responsible for managing, monitoring, and updating the system. The application/service layer is where the main body of an implement exists; any software waveforms, system services, and end-user applications would exist at this layer.

The goal of developing this architecture is creating a secure model for software radios that can be implemented in a variety of different manners. We show an example of how this can be applied to GNU Radio flowgraphs for building secure SDR applications.

8.2 Future Work

The research presented in this dissertation focuses on the architecture for secure SDR systems and the initial performance characterization of that architecture. The breadth of wireless systems and SDR security is massive and the work here only barely examines it. There are several directions moving forward for additional work based on this research. Here, we provide a few future directions mainly focused on performance evaluation, system optimization, and architecture application.

8.2.1 Performance Evaluation

Our work focused mainly on maximum flowgraph throughput within different isolation environments, but there are many more performance aspects to consider. The next big step moving forward would be evaluating the throughput analysis for different inter-process communications mechanisms, and focusing on a latency analysis incurred for the environments and IPC mechanisms. Latency analysis is a more complicated measurement that is highly dependent on the system configuration.

Additional performance research could also consider more software frameworks/applications, computing hardware, software configurations, or environments based on the initial performance evaluations presented here. We only characterized flowgraph performance on desktop operating systems and hardware, but many production systems would likely execute on embedded or cloud-computing hardware which could prove to have different overheads based on system support. Other research could consider the type of operating system for both the host and guest environments, such as real-time operating systems for guest environments, and the performance impact of executing multiple isolation environments simultaneously. We chose only to test the LiquidDSP and GNU Radio SDR frameworks, but many more applications and frameworks exist like srsLTE for developing SDR base-stations.

8.2.2 System Optimization

Our results presented for GNU Radio indicate that the scheduler is not well optimized for virtual environments. There is likely optimization that could be completed within the GNU Radio runtime to improve the overall performance and reduce the overall overhead in virtual environments; this could focus either on the buffering system or scheduler itself depending on what is the major bottleneck.

Many of the future performance tests mentioned above could also apply to system optimization by determining what are the best system configurations that provide the highest performance. Though, if the isolated application is still capable of meeting performance requirements, optimization may not be the highest priority. Other optimization tasks could include optimized implementations for IPC between isolated zones and optimized resource scheduling for guest environments.

8.2.3 System Implementations

Another major future direction moving forward is implementing the architecture in production systems and future applications. The architecture is mainly focused on SDRs, but any wireless system with a software component or interface can use this approach. Also, non-wireless networking systems could be developed in a similar manner for added security; this includes technologies such as Software Defined Networks (SDNs) and Network Function Virtualization (NFV).

Many embedded systems are resource constrained and might not be able to support operating systems that provide even lightweight isolation systems like containers. Another approach such as a micro-kernel or separation kernel would be the best option for isolation but, many times this is a more complex effort due to the lack of full frameworks like GNU Radio. Having a framework like GNU Radio capable of executing with real-time operating systems or micro-kernels would be an interesting research path moving forward.

Bibliography

[1] A. Nordrum. The internet of fewer things [news]. *IEEE Spectrum*, 53(10):12–13, October 2016.

[2] Mark Hung. Leading the iot - gartner insights on how to lead in a connected world. Technical report, Gartner, 2017.

[3] Ericsson mobility report. Technical report, Ericsson, June 2019.

[4] Jason M. McGinthy. *Solutions for Internet of Things Security Challenges: Trust and Authentication*. PhD thesis, Virginia Tech, 2019.

[5] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *IEEE Communications Surveys Tutorials*, 21(3):2702–2733, thirdquarter 2019.

[6] Nvidia sdr (software defined radio) technology: The modem innovation inside nvidia i500 and tegra 4i. Technical report, NVIDIA, 2013.

[7] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[8] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, Santa Clara, CA, August 2019. USENIX Association.

[9] Pranav S Ambavkar, Pranit U Patil, BB Meshram, and Pamu Kumar Swamy. Wpa exploitation in the world of wireless network. *Int J Adv Res Comput Eng Technol*, 1(4):609–618, 2012.

[10] F.T. Sheldon, John Mark Weber, Seong-Moo Yoo, and W. David Pan. The insecurity of wireless networks. *Security Privacy, IEEE*, 10(4):54–61, July 2012.

[11] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1313–1328, New York, NY, USA, 2017. ACM.

[12] Joshua Wright and Johnny Cache. *Hacking Exposed Wireless: Wireless Security Secrets & Solutions*. McGraw-Hill Education Group, 3rd edition, 2015.

[13] A. Scott, T. J. Hardy, R. K. Martin, and R. W. Thomas. What are the roles of electronic and cyber warfare in cognitive radio security? In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, Aug 2011.

[14] Oracle vm virtualbox. https://www.virtualbox.org, 2019.

[15] Docker. https://www.docker.com, 2019.

[16] J. Mitola. Software radio architecture: a mathematical perspective. *IEEE Journal on Selected Areas in Communications*, 17(4):514–538, April 1999.

[17] Yuan Lin, Hyunseok Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 89–101, 2006.

[18] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From soda to scotch: The evolution of a wireless baseband processor. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 152–163, Nov 2008.

[19] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Anysp: Anytime anywhere anyway signal processing. *IEEE Micro*, 30(1):81–91, Jan 2010.

[20] Y. Chen, S. Lu, H. S. Kim, D. Blaauw, R. G. Dreslinski, and T. Mudge. A low power software-defined-radio baseband processor for the internet of things. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 40–51, March 2016.

[21] B. Bloessl, M. Segata, C. Sommer, and F. Dressler. Towards an open source ieee 802.11p stack: A full sdr-based transceiver in gnu radio. In *2013 IEEE Vehicular Networking Conference*, pages 143–149, Dec 2013.

[22] Range Networks. OpenBTS. https://github.com/RangeNetworks/openbts, 2019.

[23] Software Radio Systems. srsLTE. https://github.com/srsLTE/srsLTE, 2019.

[24] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srslte: An open-source platform for lte evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, WiNTECH '16, pages 25–32, New York, NY, USA, 2016. ACM.

[25] Johannes Pohl and Andreas Noack. Universal Radio Hacker: A Suite for Wireless Protocol Analysis. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, IoTS&#38;P '17, pages 59–60, New York, NY, USA, 2017. ACM.

[26] Jean-Michel Picod, Arnaud Lebrun, and Jonathan-Christofer Demay. Bringing Software Defined Radio to the Penetration Testing Community. 2014.

[27] E. Blossom. Gnu radio: Tools for exploring the radio frequency spectrum. *Linux Journal*, June 2012.

[28] GNU Radio. https://www.gnuradio.org, 2019.

[29] Redhawk. https://redhawksdr.github.io/, 2019.

[30] Joseph D. Gaeddert. LiquidDSP. http://liquidsdr.org, 2019.

[31] Alexandru Csete. Gqrx SDR. http://gqrx.dk, 2019.

[32] Kevin Reid. shinysdr. https://kpreid.github.io/shinysdr/, 2019.

[33] Eugene Grayver. *Why SDR?*, pages 9–35. Springer New York, New York, NY, 2013.

[34] J. Mitola and G. Q. Maguire. Cognitive radio: making software radios more personal. *IEEE Personal Communications*, 6(4):13–18, Aug 1999.

[35] Ashwin Amanna and Jeffrey H Reed. Survey of cognitive radio architectures. In *IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the*, pages 292–297. IEEE, 2010.

[36] A. B. MacKenzie, J. H. Reed, P. Athanas, C. W. Bostian, R. M. Buehrer, L. A. DaSilva, S. W. Ellingson, Y. T. Hou, M. Hsiao, J. Park, C. Patterson, S. Raman, and C. R. C. M. da Silva. Cognitive radio and networking research at virginia tech. *Proceedings of the IEEE*, 97(4):660–688, April 2009.

[37] J. Wu, Z. Zhang, Y. Hong, and Y. Wen. Cloud radio access network (c-ran): a primer. *IEEE Network*, 29(1):35–41, Jan 2015.

[38] E. Grayver, A. Chin, J. Hsu, S. Stanev, D. Kun, and A. Parower. Software defined radio for small satellites. In *2015 IEEE Aerospace Conference*, pages 1–9, March 2015.

[39] Daniel Estévez. Introducing gr-satellites. https://destevez.net/2016/08/introducing-gr-satellites/, August 2016.

[40] S. Hitefield, Z. Leffke, M. Fowler, and R. W. McGwier. System overview of the virginia tech ground station. In *2016 IEEE Aerospace Conference*, pages 1–13, March 2016.

[41] J. A. Stankovic. Research directions for the internet of things. *IEEE Internet of Things Journal*, 1(1):3–9, Feb 2014.

[42] NIST. Common vulnerability scoring system. https://nvd.nist.gov/vuln-metrics/cvss, 2019.

[43] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, July 1974.

[44] J. M. Rushby. Design and verification of secure systems. *SIGOPS Oper. Syst. Rev.*, 15(5):12–21, December 1981.

[45] Jim Alves-Foss, Carol Taylor, and Paul Oman. A multi-layered approach to security in high assurance systems., 01 2004.

[46] Jon Erickson. *Hacking: The Art of Exploitation, 2nd Edition*. No Starch Press, San Francisco, CA, USA, second edition, 2008.

[47] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Security Privacy*, 12(4):63–67, July 2014.

[48] Pax Non-Executable Stack (NX). https://pax.grsecurity.net/docs/noexec.txt.

[49] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, Bellevue, WA, 2012. USENIX.

[50] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, 1998.

[51] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[52] Mendel Rosenblum, Tal Garfinkel, and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, May 2005.

[53] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[54] Marc L. Lichtman. *Antifragile Communications*. PhD thesis, Virginia Tech, 2016.

[55] G. Baldini, T. Sturman, AR. Biswas, R. Leschhorn, G. Godor, and M. Street. Security aspects in software defined radio and cognitive radio networks: A survey and a way ahead. *Communications Surveys Tutorials, IEEE*, 14(2):355–379, Second 2012.

[56] Telecommunication networks security requirements. Technical Report E.408, ITU-T, May 2004.

[57] Raquel L. Hill, Suvda Myagmar, and Roy Campbell. Threat analysis of GNU software radio. In W. W. Lu, editor, *Proceedings - 6th World Wireless Congress, WWC*, pages 383–388. 2005.

[58] Ralf-Philipp Weinmann. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, pages 2–2, Bellevue, WA, 2012. USENIX Association.

[59] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1), April 2017.

[60] Nitay Artenstein. BROADPWN: Remotely Compromising Android and iOS Via a Bug in Broadco'S Wi-Fi Chipsets, July 2017.

[61] Ben Seri and Gregory Vishnepolsky. BlueBorne - Technical Report. Technical Report, Armis.

[62] L. B. Michael, M. J. Mihaljevic, S. Haruyama, and R. Kohno. A framework for secure download for software-defined radio. *IEEE Communications Magazine*, 40(7):88–96, July 2002.

[63] Alessandro Brawerman, Douglas Blough, and Benny Bing. Securing the Download of Radio Configuration Files for Software Defined Radio Devices. In *Proceedings of the Second International Workshop on Mobility Management &Amp; Wireless Access Protocols*, MobiWac '04, pages 98–105, New York, NY, USA, 2004. ACM.

[64] H. Uchikawa, K. Umebayashi, and R. Kohn. Secure download system based on software defined radio composed of FPGAs. In *The 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, volume 1, pages 437–441 vol.1, September 2002.

[65] T. Doan M. Togooch J. Takada K. Sakaguchi, C. Fung Lam and K. Araki. Acu and rsm based radio spectrum management for realization of flexible software defined radio world. *IEICE Trans. Communications E Series B*, 86(12):3417–3424, December 2003.

[66] AG. Fragkiadakis, E.Z. Tragos, and IG. Askoxylakis. A Survey on Security Threats and Detection Techniques in Cognitive Radio Networks. *Communications Surveys Tutorials, IEEE*, 15(1):428–445, 2013.

[67] A. Attar, H. Tang, A. V. Vasilakos, F. R. Yu, and V. C. M. Leung. A Survey of Security Challenges in Cognitive Radio Networks: Solutions and Future Research Directions. *Proceedings of the IEEE*, 100(12):3172–3186, December 2012.

[68] J. M. Park, J. H. Reed, A. A. Beex, T. C. Clancy, V. Kumar, and B. Bahrak. Security and Enforcement in Spectrum Sharing. *Proceedings of the IEEE*, 102(3):270–281, March 2014.

[69] Y. Zhang, G. Xu, and X. Geng. Security Threats in Cognitive Radio Networks. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 1036–1041, September 2008.

[70] Eimear M Gallery and Chris J Mitchell. *Trusted computing technologies and their use in the provision of high assurance SDR platforms*. 2006.

[71] Chunxiao Li, A. Raghunathan, and N. K. Jha. An architecture for secure software defined radio. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 448–453, April 2009.

[72] David Murotake and Antonio Martín. A high assurance wireless computing system (hawcs®) architecture for software defined radios and wireless mobile platforms. 2009.

[73] S. Hitefield, V. Nguyen, C. Carlson, T. O'Shea, and T. Clancy. Demonstrated llc-layer attack and defense strategies for wireless communication systems. In *2014 IEEE Conference on Communications and Network Security*, pages 60–66, Oct 2014.

[74] S. D. Hitefield, M. Fowler, and T. C. Clancy. Exploiting buffer overflow vulnerabilities in software defined radios. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1921–1927, July 2018.

[75] Ettus Research. Usrp x310 high performance software defined radio. https://www.ettus.com/all-products/x310-kit/.

[76] Kernel-based virtual machine. https://www.linux-kvm.org/page/Main_Page, 2018.

[77] Kubernetes. https://kubernetes.io, 2018.

[78] Katacontainers. https://katacontainers.io, 2018.

[79] Firecracker - MicroVMs. https://firecracker-microvm.github.io, 2019.

[80] Selinux. http://selinuxproject.org/page/Main_Page, 2018.

[81] Apparmor. https://wiki.ubuntu.com/AppArmor, 2018.

[82] S. H. VanderLeest. The open source, formally-proven sel4 microkernel: Considerations for use in avionics. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9, Sep. 2016.

[83] Kathleen Fisher. Using formal methods to enable more secure vehicles: Darpa's hacms program. *SIGPLAN Not.*, 49(9):1–1, August 2014.

[84] H. Liang, Q. Hao, M. Li, and Y. Zhang. Semantics-based anomaly detection of processes in linux containers. In *2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*, pages 60–63, Oct 2016.

[85] A. S. Abed, T. C. Clancy, and D. S. Levy. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops (GC Wkshps)*, pages 1–5, Dec 2015.

[86] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE Symposium on Security and Privacy*, pages 297–312, May 2011.

[87] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. HYPERSHELL: A practical hypervisor layer guest OS shell for automated in-vm management. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 85–96, Philadelphia, PA, 2014. USENIX Association.

[88] Yaml: Yaml ain't markup language. https://yaml.org.

[89] Pallets Project. Jinja template engine. https://www.palletsprojects.com/p/jinja/.

[90] Pyro - Python Remote Objects. https://pythonhosted.org/Pyro4/.

[91] Colin Ian King. stress-ng: A stress-testing swiss army knife. https://elinux.org/images/5/5c/Lyon-stress-ng-presentation-oct-2019.pdf, October 2019.

[92] Stress-ng manual page. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html.

[93] Unix Domain Socket - Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/unix.7.html, 2019.

[94] FIFO - Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/fifo.7.html, 2019.

[95] Intel. Intel Turbo Boost Technology in Intel CoreTM Microarchitecture (Nehalem) Based Processors. Technical report, November 2008.

[96] Sparsh Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *CoRR*, abs/1401.0765, 2014.

[97] Marr, Deborah T.; Binns, Frank; Hill, David L.; Hinton, Glenn; Koufaty, David A.; Miller, J. Alan; Upton, Michael. Hyper-Threading Technology Architecture and Microarchitecture. Technical report, 2002.

[98] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[99] N. West, D. Geiger, and G. Scheets. Accelerating software radio on arm: Adding neon support to volk. In *2015 IEEE Radio and Wireless Symposium (RWS)*, pages 174–176, Jan 2015.

[100] Vector-Optimized Library of Kernels. `http://libvolk.org`, 2019.

[101] Gregory Lento. Optimizing Performance with Intel Advanced Vector Extensions. Technical report, Intel, September 2014.