LYNXTUN

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GALİP ORAL OKAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF CYBER SECURITY

SEPTEMBER 2018

Approval of the thesis:

LYNXTUN

submitted by **GALİP ORAL OKAN** in partial fulfillment of the requirements for the degree of **Master of Science in Cyber Security Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, **Graduate School of Informatics** _______________

Assoc. Prof. Dr. Cengiz Acartürk
Head of Department, **Cyber Security** _______________

Prof. Dr. Nazife Baykal
Supervisor, **Information Systems, METU** _______________

Dr. Cihangir Tezcan
Co-supervisor, **Mathematics, METU** _______________

Examining Committee Members:

Prof. Dr. Ali Aydın Selçuk
Computer Engineering, TOBB ETÜ _______________

Prof. Dr. Nazife Baykal
Information Systems, METU _______________

Assoc. Prof. Dr. Cengiz Acartürk
Cognitive Science, METU _______________

Assoc. Prof. Dr. Ali Doğanaksoy
Mathematics, METU _______________

Asst. Prof. Dr. Elif Sürer
Multimedia Informatics, METU _______________

Date: _______________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Galip Oral Okan

Signature :

ABSTRACT

LYNXTUN

Okan, Galip Oral
M.S., Department of Cyber Security
Supervisor : Prof. Dr. Nazife Baykal
Co-Supervisor : Dr. Cihangir Tezcan

September 2018, 134 pages

Lynxtun is a VPN solution that allows the creation of a secure tunnel between two hosts over an insecure network. The Lynxtun Protocol transmits fully encrypted datagrams with a fixed size and at a fixed interval using UDP/IP. Our custom authenticated encryption scheme uses the AES-256 block cipher and modified version of GCM mode in order to decrypt and authenticate datagrams efficiently. It ensures traffic flow confidentiality by maintaining a constant bitrate that does not depend on underlying communication. In this sense, it provides unobservable communication. This constitutes a difficult engineering problem. The protocol design allows implementations to fulfill this requirement. We analyze factors that influence realtime behavior and propose solutions to mitigate this. We developed a full implementation for the GNU/Linux operating system in the C programming language. Our implementation succeeds in performing dispatch operations at the correct time, with a tolerance on the order of microseconds, as we have verified empirically.

Keywords: Network Security, Unobservable Communication, TFC, VPN

ÖZ

LYNXTUN

Okan, Galip Oral
Yüksek Lisans, Siber Güvenlik Bölümü
Tez Yöneticisi : Prof. Dr. Nazife Baykal
Ortak Tez Yöneticisi : Dr. Cihangir Tezcan

Eylül 2018 , 134 sayfa

Lynxtun güvenilir olmayan bir ağ ile bağlı iki bilgisayar arasında güvenli bir tünel kurulmasını sağlayan bir VPN çözümüdür. Lynxtun Protokolü, sabit boyutta ve tamamen şifreli paketlerin UDP/IP kullanılarak sabit aralıklarla gönderilmesini öngörür. Şifreleme ve doğrulama için kendi geliştirdiğimiz AES-256 blok şifresi ve modifiye edilmiş GCM moduna dayalı bir yöntem kullanılır. Veri aktarım hızını asıl haberleşmeden bağımsız olarak sabit tutarak trafik akışı gizliliği sağlanır. Bu açıdan, gözlemlenemez haberleşmeye olanak sağlar. Bu zor bir mühendislik problemidir. Protokol, uygulamaların bu şartı fiilen sağlayabilmelerini mümkün kılacak şekilde tasarlanmıştır. Tespit ettiğimiz gerçek zamanlı işleyişi etkileyebilecek unsurlara yönelik çeşitli çözüm önerileri geliştirdik. Protokolün tam kapsamlı bir uygulamasını C dilini kullanarak GNU/Linux işletim sistemi için geliştirdik. Deneysel olarak doğruladığımız üzere, uygulamamız mikrosaniyeler düzeyindeki bir hata payı ile gönderim işlemlerini doğru zamanlarda gerçekleştirmektedir.

Anahtar Kelimeler: Ağ Güvenliği, Gözlemlenemez Haberleşme, TFC, VPN

Annem, Babam ve Biricik Ezupufuma

ACKNOWLEDGMENTS

The day that I met Prof. Dr. Nazife BAYKAL has been one of the turning points in my life. I was truly inspired by her vision. It was that day that I decided to enter the field of cybersecurity. Now, I have the great pleasure of thanking her for all that she has done for me. It has been a privilege to have her as my thesis supervisor.

I would express my deepest gratitude to Assoc. Prof. Cengiz ACARTÜRK. This study would not be possible without his wise guidance and support.

I would like to thank Assist. Prof. Aybar Can ACAR, Assist. Prof. Şeyda ERTEKİN and Assoc. Prof. Ali DOĞANAKSOY. Their suggestions have been extremely valuable to me in performing this research. However, their influence on my work goes far beyond this. I have learned much from them over the years, and I can only hope that I shall be able to produce work worthy of their teachings.

I would like to specially thank my co-advisor Dr. Cihangir TEZCAN. It is from him that I learned cryptography, and I now share his deep enthusiasm for the subject. If I have been able to complete this study, it is because of his unwavering support and excellent advice. From a technical perspective, his influence on my work has been crucial. And yet, I have something more important to thank him for. His friendship.

TABLE OF CONTENTS

CHAPTERS

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

1.1.1 Security Implications of Information Technology

Whole of human civilization is rooted in our ability to create, acquire, record, process and share information. Throughout history, important technological developments in this area have often led to significant progress, transforming human society. The very ability to use language to encode and communicate information is among the defining characteristics of our species. Despite the fact that interconnected programmable digital computers are a very recent development, the magnitude of their impact has been such that they have become an integral part of society. The internet has become the preeminent medium through which communication takes place. As a result, we are able to create, acquire, record, process and share information with unprecedented effectiveness, and at a scale that is incomparable to what was possible for most of our history. Within a very short timespan, the advent of computer technology has transformed society thoroughly. These changes appear to be irreversible, as it is highly unlikely that society will revert to what it was like before computers were introduced. On the other hand, the transformation appears to be far from complete. Technology continues to progress at an astounding pace, making the situation highly dynamic and volatile. Our society is faced with the challenge of understanding the ramifications of such sudden and all-encompassing change. This is a difficult task indeed, and can at times require thinking about the assumptions that underly human society. One such assumption is that secrets can be kept.

While it is true that human society thrives on our ability to use information, it also requires that access to information can be restricted in compliance with our social constructs. We require confidentiality of personal, commercial and national secrets. Information technology gives us tools that make information more accessible. While we derive great benefit from this, it also makes it more difficult to prevent unauthorized parties from gaining access to confidential information.

1.1.2 Secure Network Communication

In order to discuss secure communication, we first have to develop an understanding of what communication is. At the highest level, we can define communication to be the conveyance of information. For the purposes of our analysis, we require a more precise definition. In his seminal paper, Shannon [50] presents a mathematical theory of communication. He states that "the fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point." The semantics of the message are irrelevant. All that is important is that the message is selected from a set of possible messages. The problem is conveying this selection to the other side. This can be done, for instance, by encoding the message in a signal that is carried by some physical phenomenon, such as electromagnetic waves. This definition of communication provides a precise definition of what information is. In the context of digital computers, information is represented as binary strings. Text can be represented as binary data by using an encoding scheme such as ASCII or UTF-8. Analogue signals that make up images and sounds can be sampled to produce a binary representation. While text, images and sound recordings are different in terms of semantics, this has no relevance to the problem of network communication which is concerned with the transmission of binary strings.

In this study, we focus on network communication that is based on packet switching. In particular, we focus on IP networks. The IP protocol defines the mechanisms through which a packet of data can be sent from one network host to another. Each IP packet contains a header that specifies the origin and the destination, the amount of data contained in the packet, in addition to various other fields that can influence how the packet is to be routed across the network. Each packet travels from the source to the destination, and can traverse several hops (intermediary network hosts) along the way. Each of the hosts along the path share a physical link, such as an ethernet connection or a WiFi connection. An attacker with access to any of the network hops or the links are able to observe the packets. Unless a network is completely isolated, it must be assumed to be insecure. This is especially true in the case of the internet.

When we consider IP communication, we see that it can be modeled as a stochastic process X_i, where

$$X_i = (\Xi_i, \zeta_i, \tau_i, \eta_i) \tag{1.1}$$

X_i is the information associated with the i'th IP packet, with Ξ_i being the message, ζ_i its size, τ_i the time at which it was observed on the network and η_i its *direction*, which is its origin and destination. When we consider point-to-point communication between two hosts, η_i reduces to a single bit of information. These are the observable features of network traffic. Additional features can be derived, such as the average data transmission rate over a period of time, the total amount of data exchanged over a period of time, and the burstiness of traffic. Imperfections due to the network (delays, corruption, etc.) can be expected to lead to slight variations in observed features, depending on where and when the observation is made. Other than this, what is seen by the intended recipient and any observer on the network is equivalent.

The message Ξ_i is, by and large, the most important part. It is the reason why communication takes place at all. People talk when they have something to say, because they have something to say. Otherwise, there is silence. Similarly, computers send packets when they have data to send. It is of the utmost importance that an attacker cannot see Ξ_i if the communication is to be confidential. But how can this be achieved in light of what we have said in the previous paragraph? How is it that two people look at the same data, and yet only one of them is able to understand it? The answer to this question is provided by cryptography.

Virtually all secure network protocols rely on cryptography to solve this problem. Diffie and Hellman [23] say that "the best known cryptographic problem is that of privacy: preventing the unauthorized extraction of information from communications over an insecure channel." This is precisely the problem as we have presented it. In an encrypted network protocol, the plaintext message Ξ_i is replaced by the ciphertext message $E_K(\Xi_i)$, where E is an encryption algorithm and K is the encryption key. The cipher and the encryption key combined define a mathematical transformation that maps a plaintext message to a corresponding ciphertext message. This transformation has an inverse, $D_{K'}$, where D is the decryption algorithm and K' is the decryption key[1]. The crucial aspect is that is should be computationally infeasible to derive the decryption transformation without prior knowledge of the decryption key, which is assumed to have been delivered in advance only to the intended recipient over a secure channel. Moreover, ciphers are designed to hide inherent redundancy in plaintext through *confusion* and *diffusion*, so that ciphertext is statistically indistinguishable from randomly generated data. This way, $E_K(\Xi_i)$ that is observable on the network appears to have been randomly generated and only the intended recipient who has access to the correct decryption key can recover the hidden message Ξ_i.

While the use of cryptography is necessary to secure network communication, it is not sufficient. The problem is due to the fact that network communication is made up of a sequence of related messages. Encrypting message content, that is Ξ_i, ensures that the message is confidential when treated in isolation. However, this has no effect on observable patterns in network traffic that arise from the relationships between multiple messages and that can be used to make inferences about the content of the messages. These patterns manifest themselves in generated timing, size and direction information.

Sending a message necessarily generates corresponding values for the auxiliary features of time, size and direction. However, by virtue of the fact that these are also values selected from a possible set, they can be considered to be messages themselves. These auxiliary messages can encode semantic information that is complementary to the content of the messages. We now present two examples to show how this can be the case.

For our first example, suppose that Alice is in love with Bob, as of yet unbeknownst to him. Alice sends Bob a text message, professing her love. The words "I Love You." are sent using an instant messaging app on Alice's smartphone. As is customary of such apps, Alice receives a notification when Bob sees the message. Bob now knows. How will he respond? Alice starts to experience increasing feelings of restlessness with every passing second. "Why is he taking so long to answer?" she wonders. Perhaps it is because he is looking for the right words to inform her that her love is unreciprocated without causing her too much agony. Surely the impossibility of such a task could account for a prolonged delay. This delay carries semantic information, that is related to the content of Bob's response.

[1] In a symmetric cipher, the decryption key is the same as the encryption key.

As a second example, consider Shakespeare's *A Midsummer Night's Dream*. The verses represent messages that characters send to one another. While the precise times at which these messages are sent is a matter of artistic delivery, the fixed ordering of these impose timing constraints. The script also specifies who is speaking. If we were to redact all of this information by having a single actor recite the entire script, shuffling the order of the lines at random, the play would become unintelligible. The content of each of the individual messages remains intact, yet it is no longer possible to understand what is being communicated. In this example, we see that auxiliary features can encode semantic information pertaining to the communication that is not explicitly available from the messages themselves.

Similar examples also exist in the field of network communication. Various types of network activity will result in distinct timing patterns that will be reflected in network traces. An attacker can then use this information to make inferences about the confidential content even if the packets are encrypted. For example, a shell session over SSH transmits data at each keystroke. The timestamps of recorded packets will mirror the typing behavior. Streaming videos will be characterized by significant data flow in one direction while the buffer is being filled that is later throttled down. Web browsing exhibits spurious increases in data rate when the user clicks a new link and the page is being loaded.

As we have said, securing messages in isolation by encrypting them is not tantamount to securing the entire communication. This is a well known problem in the field of network communication. The problem is that of *Traffic Flow Confidentiality (TFC)*. TFC countermeasures depend on *traffic shaping*. Essentially, traffic shaping involves explicitly specifying when data is sent independently from the underlying communication. There are two main ways in which this could be done. First, data that is ready to be sent may be delayed until a scheduled time in the future. Second, if a dispatch is due but there is not enough actual data to send, *dummy* data should be generated to make up for the deficiency.

1.1.3 The Cost of Security

In optimization theory, it is a well known fact that introducing an additional constraint can never improve the optimal solution. Security requirements are constraints. Therefore, implementing security countermeasures will *always* incur some amount of overhead. The generated overhead can involve additional computational or network resources being used, or latencies being introduced. For a given usecase, there is a limit to the amount of overhead that can be justified.

The use of encryption is no exception to this principle. Additional work has to be done in order to encrypt and decrypt the data, which takes time and uses computational resources. This is likely to have been a contributing factor to the reluctance of certain popular websites from switching to HTTPS, in spite of the fact that they handle sensitive personal data. The additional overhead is detrimental to user experience. However, the importance of encrypting network communication has become widely acknowledged. It is important to note that there is an ongoing trend such that regulatory legislation is being produced that requires services that handle sensitive data to be compliant with various information security standards, and the use of encryption is a central theme.

It is more difficult to justify TFC countermeasures. Unlike the case with encryption, data obtained from auxiliary features like timing information can only be used to compromise confidentiality in an indirect way. Such attacks are often described as *side-channel attacks*. Delaying data that is ready to be sent increases network latency, and interactive network communication requires latency to be as low as possible. Sending dummy data increases the use of network resources, which are a precious commodity. A constant bitrate (CBR) protocol is one that sends data at a fixed rate.

Our assessment is that the amount of overhead generated to provide full TFC is not justified in most usecases. As such, most secure network protocols do not strive for full TFC. However, the fact that this is not required in most usecases does not imply that there are no usecases where it is required.

It is misleading to think of overhead as a waste of resources. If overhead is necessary to ensure security, then the resources are not wasted. They are the cost of security. Whether or not the cost is justified depends on the security requirements. But there is also the issue of whether the solution is actually *viable*.

Network bandwidth availability is much greater now than in the past. People regularly use the internet to watch HD quality movies, which is a leisure activity. Doing so requires gigabytes worth of data to be transferred. Let us assume a typist writes at a rate of 5 keystrokes per second. Using ASCII encoding, this comes out to 5 bytes per second and about 422 KB per day. Suppose we were to implement, perfectly, an encrypted, constant bit-rate (CBR) system that can transmit 5 bytes per seconds. If an operator types in a message, it gets encapsulated in the data that is sent. Otherwise, dummy data is sent. Since data is being sent every second, the maximum latency introduced is one second. Accounting for overhead due to packet headers and matters related to encryption, we can safely assume that dedicating a bandwidth of several megabytes per day bidirectionally will be sufficient to implement such a system. This is negligible when compared to watching a single movie per day.

A CBR protocol can also be used to watch a movie. Specifically, if the CBR matches the bandwidth necessary to stream the movie, then there will be no overhead. The overhead arises when there is no actual communication. Running a prolonged CBR connection that allows streaming movies through it is similar to streaming movies all day, everyday.

1.1.4 Research Objectives

Our primary research objective is to develop a system that can be used to secure network communication over an insecure network, with the level of security being analogous to what is possible through the use of cryptography in the context of static data. Furthermore, we want the system to run on a general purpose operating system running on general purpose hardware.

To interpret this requirement, let us elaborate on what we mean by the level of security provided by cryptography in the context of static data. Suppose that there is a 4 GB thumb drive. This data is static, as in it does not change over time. Furthermore, no additional data is generated in the future. Performing statistical analysis on the data indicates that it is indistinguishable from randomly generated data. The aim of cryptography is to ensure that ciphertext is statistically indistinguishable from randomly generated data. Unless we have prior knowledge of the correct decryption transformation to use, all we can say is that the data *might* be ciphertext. In other words, not only are we able to make any inferences to the content of the plaintext, we cannot even determine whether or not there is a hidden plaintext at all. However, there is one thing that we can say with certainty. If the data is ciphertext, then the corresponding plaintext cannot contain more information (entropy-wise) then can be encoded in 4 GB. In other words, one cannot take a single bit and claim that it is an encrypted version of the collected works of Shakespeare.

As we have discussed, encrypting packets does not lead to the same level of security in the context of network communication. Unlike the example above, network communication is *dynamic*, and patterns in observable features of network traffic that are not protected by encryption can be used to compromise security. Achieving the same level of security requires regulating all observable features of the generated network traffic. This requires using cryptography to protect individual packets, and also ensuring Traffic Flow Confidentiality.

A stronger statement of our security requirement is to say that the communication should be *unobservable*. This is analogous to what we have said about not being able to definitively identify ciphertext. Of course, an attacker will always be able to observe the generated network traffic. However, none of the observable features of this traffic should be correlated with the properties of actual communication that may be taking place. In other words, the network traffic should be statistically invariant to the underlying communication. An attacker should not be able to reliably determine whether or not actual communication is taking place. The only thing that an attacker should be able to say with certainty is the maximum amount of data that may have been exchanged within a certain period of time.

We have one additional security requirement that does not have a counterpart in the static case. The communication should be *undisturbed*. In essence, it is a matter of protecting the ongoing communication from external influences, intentional and malicious, or otherwise. This involves the authentication of data (which also relies on cryptographic methods) and resilience against data injection and modification attacks, replay attacks, and denial of service attacks.

In the field of security research, one has to accept that there is no such thing as absolute security. On the other hand, absolute security is not necessary. Information security requirements should not be considered independently of how they relate to the lives of actual humans. The fact that we have a functioning society is proof that we have thus far been successful in attaining sufficiently high levels of security for most purposes. On the other hand, there are certain usecases that require stricter security requirements. While perfect security is unattainable, that is no reason not to aim for it. It is by aiming for perfect security that we can come as close to it as possible. Also, such an endeavour allows us to discover what keeps us from achieving it and further our understanding of security in general.

In this study, we set upon such an endeavor. We set out to answer the following question. How should one design a network protocol, that is intended to be implemented as a userspace application running on a general-purpose operating system and hardware, so that it comes as close to perfect security as possible. Our answer to this question is Lynxtun, and is the subject of this thesis. We take a holistic approach, covering all aspects of the system including the protocol specification, implementation and system configuration.

1.2 Proposed Solution

In this thesis we present Lynxtun, which is a VPN solution that is designed to run as a userspace application on a general-purpose operating system and hardware while striving to achieve the highest level of security possible. It allows the creation of a secure point-to-point tunnel between two network hosts over an insecure network. It does this through the use of strong cryptography and the regulation of all observable features of network traffic in order to achieve traffic flow confidentiality. For this purpose, it adopts the constant bit-rate (CBR) approach.

The Lynxtun Protocol defines the protocol between the two endpoints of a Lynxtun tunnel. At its core, it is an encrypted layer-3 encapsulation protocol that encapsulates IP datagrams in fully encrypted UDP datagrams. We develop our own cryptographic protocol that is based on the AES-256 block cipher and a modified version of the Galois Counter Mode (GCM). Authentication is due to the use of GCM's authenticated encryption. The Lynxtun Protocol also includes countermeasures against replay attacks, and is designed to be resilient against denial-of-service attacks.

Encrypted Lynxtun datagrams have a fixed size. Their payload includes encapsulated IPv4 packets and padding. These datagrams are dispatched at fixed intervals. The size of the datagrams and the interval at which they are dispatched are configuration parameters for the tunnel.

The Lynxtun Protocol is not connection oriented and does not provide reliable delivery. There is a logical separation of the processing of incoming data and outgoing data. There is no concept of request and response. A Lynxtun endpoint will continue to send datagrams whether or not it receives any data in return. It might even be the case that the tunnel peer goes offline. This will not effect the dispatch process of the other. The initial dispatch is functionally identical to all subsequent ones. There is no initial negotiation or connection-establishment phase. The only shared state that exists in the system is the shared configuration parameters that the user must provide when starting the endpoint. This includes a shared AES-256 secret key in addition to the dispatch interval and size parameters mentioned above. This information has to be shared between the two hosts in a secure way prior to the establishment of the tunnel. This issue is outside of the scope of the Lynxtun Protocol.

A Lynxtun implementation interfaces with the network stack of the host operating system using a TUN virtual network interface driver, which is available for all major operating systems. Since the TUN device represents a logical network interface, it is configurable through standard routing and net device configuration methods. This makes it very easy to adapt Lynxtun to different usecases, such as using the tunnel peer as the default network gateway or creating a secure network with multiple hosts, where each link consists of a separate Lynxtun tunnel.

We implement the Lynxtun Protocol for the GNU/Linux operating system using the C programming language. Our implementation has no external dependencies besides the `glibc` library and the POSIX multi-threading library (`pthreads`).

The security of Lynxtun depends on regulating all observable features of generated network traffic so that these cannot be exploited by an attacker to make inferences regarding the underlying communication. This requires fully encrypting Lynxtun datagrams and regulating the dispatch process in order to provide TFC. The first issue is fully addressed by the Lynxtun Protocol specification. The latter is ultimately depends on the runtime behavior of the Lynxtun implementation.

In the sense that the Lynxtun protocol requires fixed-sized datagrams to be dispatched at fixed intervals, it is theoretically secure in terms of TFC. However, achieving this in practice is a difficult engineering problem; particularly since we are implementing Lynxtun as a userspace application. The interarrival times of datagrams as observed by an attacker will *always* be variable due to numerous factors. These include factors due to both the Lynxtun implementation itself, its complex interaction with the operating system and the underlying hardware, the statistical properties of the underlying communication, factors due to physical hardware devices, and other processes running on the same system. One should not assume that the observed variability does not depend on tunnel activity. Our goal is to understand the mechanisms through which such a relationship can manifest and identify ways of hiding such a relationship. We want, to the extent possible, that the observed variability remains invariant to tunnel activity. If the mean interarrival packet time increases when the tunnel is active (in the sense that it is being used to carry actual data), security is compromised. If the mean instead decreases, security is compromised. If the maximum deviation from the mean, when measured over a duration of five seconds, is consistently different depending on whether the tunnel is active or not, security is compromised. An ideal solution would ensure that the observed dispatch process is *statistically indistinguishable* when the tunnel is active and when it is idle.

This problem is similar to that of designing a cryptosystem, where ciphertext should be statistically indistinguishable from randomly generated data. One of the main goals cryptanalysis is to devise *distinguishers* that are able to identify ciphertext as such. It is not possible to design a cipher such that it is guaranteed that such a distinguisher can never be found. Similarly, it is not possible to assert that tunnel activity can never be detected. Actually, the problem is more difficult in our case. Cipher definitions are static, whereas Lynxtun's dispatch process is extremely dynamic. The environment in which Lynxtun runs is neither stationary nor ergodic.

As we said earlier, absolute security is neither attainable nor required in practice. There is some amount of tolerance when it comes to how successfully we are able to regulate the dispatch process. Consider the data observed by an attacker on the network. The network itself is not pristine, and therefore there will always be some random noise. There will also be noise due to unrelated work taking place on the same operating system and hardware. The smaller the signal-to-noise ratio is, the more difficult it will be for an attacker to exploit it. Deviations less than a microsecond will be extremely difficult to detect in the presence of noise on the order of milliseconds.

But how do we differentiate noise? The guiding principle that we use, which we refer to as the *burden of knowledge principle*, is as follows:

Definition 1.2.1 (Burden of Knowledge Principle)**.** Let C be a datum. Any given entity E will either have access to C, or it will not. Any information that is generated either intentionally or inadvertently by E can have been influenced by C if and only if E has access to C. Therefore, if E has access to C, which we can also describe by saying that E knows C, then it is possible for information being propagated by E to be related to C. If C is confidential information, then E carries the burden of knowledge, and must ensure that the information it generates does not accidentally divulge information related to C. Conversely, any information generated by an entity that does not know C has no bearing on its confidentiality.

All variability due to components that have access to the confidential communication is suspect. First and foremost, we have the actual Lynxtun implementation. We also have the kernel that does work on behalf of it through system calls, and the hardware on which these are being executed. We also have the actual data sources that are responsible for the IP packets that are encapsulated. These can be userspace processes running on the same host, or network interfaces. Then, there is activity belonging to tasks that have no particular reason to access the confidential information, but are nevertheless capable of doing so. These include other userspace processes running under the same user, all processes running with superuser privileges, and kernel threads. According to the burden of knowledge principle, one cannot inadvertently divulge what one does not know. therefore, we regard variability due to unrelated sources as noise.

The organization of this thesis is as follows. In Chapter 2, we provide an overview of related work. In Chapter 3, we present the Lynxtun Protocol specification. In Chapter 4, we discuss issues related to the Linux kernel and underlying hardware that are pertinent to achieving deterministic runtime behavior in a userspace process. This discussion serves two purposes. First, establishes the groundwork for our implementation of Lynxtun for the GNU/Linux operating system, and it provides insight into how a system should be configured in order to improve security. In Chapter 5, we discuss our Lynxtun implementation for the GNU/Linux operating system. In Chapter 6, we analyze empirical data collected using our Lynxtun implementation in relation to our research objectives. Finally, we present our concluding remarks in Chapter 7. Here, we discuss our contributions and identify future work to be done.

CHAPTER 2

RELATED WORK

2.1 Statistical Traffic Analysis

In the introduction, we stated that Lynxtun must regulate the dispatch process in order to prevent attackers from using publicly observable information due to the packet interarrival times and packet sizes in order to undermine the confidentiality of the communication. Attacks that target this vector are known as *statistical traffic analysis* attacks. The term for the associated security requirement is *Traffic Flow Confidentiality (TFC)*, and is identified as such in the ISO/OSI Security Architecture (ISO 7498-2) standard [33]. The standard states that TFC mechanisms can be implemented at the physical, network or application layer.

TFC mechanisms are based on traffic shaping that can generate additional *dummy* data, introduce artificial delays, or make use of fragmentation to control the statistical characteristics of the resulting traffic flow. The aim can either be to hide the traffic flow signature, or to masquerade the traffic by imitating the signature of a different type of network activity. The simplest approach to hiding traffic flow is the constant bit-rate (CBR) approach.

There is extensive literature on both statistical traffic analysis attacks and TFC countermeasures.

Traffic classification, application fingerprinting, protocol fingerprinting and version detection are related issues. These seek to identify the type of network activity taking place based on observed network traffic. The features used include but are not limited to traffic flow features. Other features include using known fixed ports and plaintext meta-data fields. Information can be collected passively by recording network traffic, or actively by sending packets in order to prompt the generation of additional network traffic. Ferreira et al. present a recent meta-analysis of feature selection approaches for traffic analysis [27]. The nmap network scanner [4] includes functionality for detecting the type and version of the operating system and software being used. Bernaille et al. [12] show that it is possible to identify which application is being used based on observing the size and direction of the first few packets of a TCP connection. Roughan et al. [47] present a framework for using statistical signatures that take into account features such as interactive use or the presence of bulk-data transport to classify network traffic. Williams et al. [56] evaluate various machine learning algorithms for automated network application identification. Another related study is due to Nguyen and Armitage, who do a survey of machine learning methods used in traffic classification [40]. Sen et al. [49] propose methods for detecting Peer-to-Peer (P2P) application signatures.

A particular type of traffic analysis attempts to identify which network hosts are communicating with each other. This is important in terms of privacy. If an attacker is able to observe the traffic close to two hosts, if they are able to match a message that is being sent to a message that is being received, then they can infer that the two hosts are communicating. Raymond presents an overview of this problem by presenting various attacks and countermeasures [45].

An important field within statistical traffic analysis is *website fingerprinting*. This aims to detect which website a user is visiting. Wang et al. present various attacks and countermeasures against website fingerprinting [55]. They use ML classifiers and claim to be able to reliably identify which of the monitored 100 websites has been visited. The countermeasures they propose are intended to be efficient in terms of generated overhead. They make the claim that they are able to provide provable defense. Other countermeasures are due to Liberatore and Levine [37] who use a Pad-to-MTU strategy and Wright et al. [57] who propose *Traffic Morphing* that seeks to reduce overhead by optimally modifying packets in real-time to match a different, but behaviorally compatible type of network activity. A paper by Dyer et al. [26] analyzes various types of so-called *efficient* website fingerprinting countermeasures and conclude that they are ineffective. They go on to claim that an effective website fingerprinting countermeasure may not be viable. They present a CBR-type countermeasure that they have named `BuFLO` that, similar to our approach, transmits fixed-sized data at fixed intervals. However, data transmission stops when a website is fully loaded. The authors are able to use this fact to compromise `BuFLO` too. An improved implementation called `CS-BuFLO` is due to Cai et al. [16].

An early notable attack is due to Song et al. [52], who observed that the SSH protocol pads data to an eight-byte boundary and the dispatch events coincide with keystrokes. They show how this can be used to reveal sensitive information including passwords. Canvel et al. [17] show how the TLS padding scheme can be exploited to intercept passwords.

The Linux kernel provides traffic control mechanisms that can be used to perform traffic shaping [15]. These are more suitable for Quality-of-Service (QoS) purposes, but can also be used to mitigate traffic analysis attacks.

There are a class of proposed solutions, including SkypeMorph, StegoTorus and CensorSpoofer that aim to achieve unobservable communication through *imitation*. That is, they try to shape the traffic to make it appear to be consistent with the signature associated by a target protocol. SkypeMorph attempts to imitate Skype, StegoTorus imitates VoIP and CensorSpoofer imitates Skype or HTTP. The important work due to Houmansadr et al.[32] shows that each of these fail in practise. This paper is especially relevant to our research. The authors provide compelling arguments for why unobservability through imitation is *fundamentally flawed*. They argue that, in order to effectively imitate a target protocol, it is not sufficient to follow the protocol standard, but rather to mimic every and all idiosyncrasies of a specific version of a specific implementation of that protocol. This is an insurmountable challenge. A single discrepancy is sufficient to detect the impersonator. They point to the work of Gianvecchio and Wang [30] which uses an entropy-based approach for detecting covert channels and suggest that such an approach would always be able to detect slight differences in entropy between traffic generated by a genuine application and an imitator.

Secure network protocols tend to have features that enable TFC mechanisms to be built upon them. For example, in TLS 1.1 [22] it is stated that padding up to 255 bytes can be added, provided it is a multiple of the block cipher's block length. IPsec has more explicit support, in the form of *TFC Padding*. RFC-4303 (IPsec ESP) [28] includes the following discussion:

> "Dummy packets can be inserted at random intervals to mask the absence of actual traffic. One can also "shape" the actual traffic to match some distribution to which dummy traffic is added as dictated by the distribution parameters. As with the packet length padding facility for Traffic Flow Security (TFS), the most secure approach would be to generate dummy packets at whatever rate is needed to maintain a constant rate on an SA. If packets are all the same size, then the SA presents the appearance of a constant bit rate data stream, analogous to what a link crypto would offer at layer 1 or 2. However, this is unlikely to be practical in many contexts, e.g., when there are multiple SAs active, because it would imply reduc- ing the allowed bandwidth for a site, based on the number of SAs, and that would undermine the benefits of packet switching. Implementations SHOULD provide controls to enable local administrators to manage the generation of dummy packets for TFC purposes."

2.2 VPN Solutions

In a phyical network, the network topology is determined by actual network interfaces installed on hosts, and the physical connections between them. A Virtual Private Network (VPN) allows using this physical topology to define alternative logical topologies. For example hosts that are located far away, connected to different networks with many hops in between and form a virtual network that, from the point of view of higher network layers, appear to be neighbours in a single local area network.

Historically, there has been a distinction between public networks and private networks, and VPNs are sometimes described as a way to simulate a private network over a public network [48]. For example, *intranets* belonging to a company are considered to be private. A VPN can be used to create a logical connection between a host on a separate network, through a so-called public network, so that the user would be able to access private resources available on the private network.

The technical foundation of building a VPN is encapsulation of network layer packets or link layer frames. This is also referred to as tunneling. Following the OSI model [60], these are referred to as Layer 3 and Layer 2 VPNs respectively.

The carrier protocol can be anything. It can be a network layer protocol, a transport layer protocol like UDP or TCP, or even an application layer protocol like SSH or even DNS [41] or HTTP [20].

A pair of Layer 2 VPN endpoints act like a switch, and a pair of Layer 3 VPN endpoints act like a router. We can think of the situation like this. Both switches and routers have multiple ports on them. The device itself is a closed box, and there is some internal wiring that somehow connects these ports inside the device, so that data going in one port can come out of the other. What we do is cut the device in half. One of the ports is attached to one network, and the other is attached to some other network which can be far away. Then, we replace the internal wiring that used to connect the two sides of the device with the internet. The entire internet is now the internal wiring of a single logical network device that just happens to be exceptionally large.

IP packets or ethernet frames picked up by one tunnel endpoint are encapsulated using the carrier protocol. The carrier protocol datagram is then sent to the other endpoint using the underlying network infrastructure. These datagrams are routed and delivered as usual. There is nothing special about them. The receiving side unpacks the datagram, extracts the packets or frames inside, and injects them into the local network.

So far, our discussion has been focused on the situation with only two tunnel endpoints, corresponding to two ports of a network device. However, the number of connections can be more than this. The basic operating principles remain the same.

In general, VPN tunnels can *funnel* simultaneous traffic flows into a single traffic flow observable on the network. This makes is a beneficial property when guarding against traffic analysis attacks. Even if the VPN protocol does not employ explicit TFC mechanisms, the signatures of the individual tunneled flows will become multiplexed and therefore harder for an attacker to analyze in isolation. This is not applicable if there is a single traffic flow, however.

OpenVPN [5] is a widely used userspace, open-source VPN solution that relies on TLS for encryption. It can be used to establish Layer-2 and Layer-3 VPNs using a virtual TAP and TUN device driver respectively. It can be configured to use both UDP and TCP. It is has a large number of configuration parameters. This leads to high complexity of the code base. The OpenVPN codebase has more than a hundred thousand lines of code. This does not include the OpenSSL library, which itself is a very large codebase with over 400,000 lines of code, and which OpenVPN relies heavily. Due to being implemented in userspace, it is less performant than IPSec, since network buffers have to be copied in between the kernel and userspace several times. Since its security is based on TLS, it has high cipher and algorithm agility. It supports the use of pre-installed symmetric keys in addition to public key cryptography for authenticating users and generating symmetric encryption keys. It can be regarded as a comprehensive general-purpose VPN solution that can adapt to meet a wide variety of operational requirements. As such, it is a userspace alternative to IPsec. Implementations are widely available for all major operating systems.

IPsec is a comprehensive VPN solution that is implemented directly within the network stack of the kernel. It is included as part of the IPv4 protocol suite. The original IPv4 specification does not include security functionality. IPsec was introduced as an enhancement of the IPv4 specification that brought security functionality, hence its name. Unlike IPv4, security is a primary design requirement of the IPv6 specification, which adopts the IPsec specification. On Linux, the IPsec implementation is based on the `xfrm` (transform) layer. It is highly configurable. Users are able to select from a range of ciphers and compression algorithms. Its design is based on clearly separated layers of abstraction. Since it is implemented in the kernel, it offers greater performance when compared to userspace solutions. When correctly configured, IPsec is a reliable and powerful VPN solution. However, configuring it correctly is known to be challenging and requires a good understanding of how it operates. This difficulty is in large part due to the existence of several layers of abstraction, each of which are highly configurable. IPsec defines *Traffic Flow Confidentiality (TFC)* padding as a countermeasure against traffic flow analysis attacks. This allows dummy packets to be inserted to mask the absence of actual traffic, which is along the same lines as the security requirements for Lynxtun. Kiraly et al. [35] propose an TFC sublayer for IPsec. IPsec is Layer-3 by definition.

SoftEther [8] is another popular VPN solution. It is also open-source and runs on Windows, Linux, Mac, FreeBSD and Solaris. SoftEther is an alternative to OpenVPN. Its encryption is also based on SSL/TLS. It has features such as tunneling over HTTPS, DNS and ICP to make it harder to detect. SoftEther also has a large codebase, which is over 300,000 lines of code. It is understood that the main differentiator of SoftEther in comparison to alternate VPN solutions is its support for firewall evasion.

WireGuard [24] is a new VPN solution that is under active development. It is primarily designed for the Linux kernel, and has a prototype that has been implemented as a kernel module. WireGuard is different from the other VPN solutions discussed in that it has made simplicity and audability one of its main design objectives. The WireGuard prototype comes in at only 4000 lines of code. The simplicity of WireGuard is due to the fact that it is highly opinionated. Unlike IPsec, OpenVPN and SoftEther, it does not provide cipher agility. It relies on pre-shared static Curve25519 keys for mutual authentication. It only uses UDP as the transport layer protocol. It uses ChaCha20Poly1305 for authenticated encryption. The author holds that forgoing cipher agility is necessary for simplicity, and simplicity is crucial for security. He points out that OpenVPN has been effected by numerous security vulnerabilities discovered in OpenSSL, which were hard to detect due to the complicated codebase. We agree with the author in thinking that this is the correct approach for developing a VPN protocol, and is the approach that we have adopted for Lynxtun.

There are also other VPN solutions such as L2TP, PPTP and PPP. Berger [11] presents a comparison of these against IPsec.

2.3 Anonymity Networks

Anonimity Networks or alternatively *Anonymous Routing Networks* are systems that are used to randomize the route each IP packet takes through the network in order to make users anonymous.

Anonymity networks are conceptually related to VPNs, as they define logical network topologies. Specific to anonymity networks, the logical network topology is highly dynamic. Moreover, anonymity networks generally operate as a network proxy. This is similar to a VPN tunnel being used as a gateway.

Anonymity networks present a possible solution to our research problem. This is due to two main reasons. First, the role of an anonymity network is to mask which hosts are communicating. Second, as stated by [59], a anonymity network router "will not relay the received packet immediately (and rather) it collects several packets and sends them in a *batch*." This batch will likely combine packets from different sources going to different destinations, which changes the timing signature.

The concept of an anonymity network dates back to the seminal paper due to Chaum in 1981 [18], who proposed the idea of using anonymous remailers, also known as *mixes*, as a solution to sending email anonymously. According to Chaum, "the purpose of a mix is to hide the correspondences between the items in its input and those in its output."

This concept later evolved into low-latency anonymous networks, that utilize similar principles to relay network packets rather than email messages. These are also called *mix-networks*. Examples include Onion routing (Tor) [53], Freedom [13], Tarzan [29] and Crowds [46]. Tarzan supports TFC mechanisms based on the CBR approach by establishing bidirectional streams between network nodes that transmit data at a fixed rate.

Johnson compares various efficient anonymous network systems [34]. Le Blond et al. evaluates anonymity networks in terms of traffic analysis resistance [36]. Zhu et al. [59] investigates flow correlation attacks, which is a type of traffic analysis attack, in mix networks. Panchenko et al. [42] describe support vector machine based website fingerprinting attacks in the context of onion routing based networks. Murdoch et al. [39] describe timing signature based attacks against the Tor network. They observe that the distortion to the timing signature is low due to the low-latency feature of the network.

2.4 Evaluation of Existing Solutions

The objective of Lynxtun is to protect communication against statistical analysis attacks and provide Traffic Flow Confidentiality. This makes it a suitable countermeasure against traffic classification and website fingerprinting attacks.

Lynxtun uses the CBR approach to TFC. The results due to Houmansadr et al. [32] indicate that an imitation-based approach is not suitable. OpenVPN, SoftEther and WireGuard have no built-in mechanisms for CBR. IPsec provides the necessary framework on top of which such a system could be built, but this requires integrating the solution on top of the highly complex IPsec infrastructure. Additionally, our aim is to achieve highly deterministic real-time behavior on the order of microseconds. Having a custom, opinionated protocol that has been specifically designed with this goal in mind is beneficial towards this goal.

When we consider the use of mix-networks as an alternative to Lynxtun, one has to differentiate between two alternate scenarios, depending on whether or not the communicating hosts are nodes of the mix network, or are external to it and using the mix network as a proxy. In the latter case, the system is open to traffic correlation attacks. An attacker who is capturing traffic on all the end nodes of the mix network might be able to match related traffic. This is best exemplified by looking at an edge case. Suppose that the two hosts communicate *exclusively* using the Tor network. If an attacker observes that there is traffic between the first host and the Tor network only when there is also traffic between the second host and the Tor network, and furthermore the amount the two traffic flows is consistent, they can reliably infer that the two hosts are communicating. In the second scenario, the endpoint itself is a mix. Using Tarzan with CBR TFC is then similar to how Lynxtun operates. However, Lynxtun focuses specifically on highly deterministic behavior, which is more difficult to achieve in a more complex system.

We have mentioned that the ISO 7498-2 [33] identifies the physical layer as a implementation target for TFC. Dedicated network hardware that supports encryption and CBR TFC would ultimately be the most secure solution, because it can utilize specially designed hardware that built specifically to exhibit highly deterministic behavior without being subject to the numerous complexities and level of uncertainty inherent in a software-based solution running on general-purpose hardware. However, this would be much more expensive. Our aim is to solve the problem in software.

CHAPTER 3

THE LYNXTUN PROTOCOL

In this section, we present a complete specification of the Lynxtun Protocol. The naming conventions that we use when specifying directions and actions are presented in Appendix A. We begin with an introductory example intended to develop a basic understanding of how the Lynxtun Protocol operates. We then present a formal specification of the protocol. We specify the interface of a Lynxtun endpoint has with the local network stack and with its tunnel peer. We describe the structure of the Lynxtun Datagrams that are exchanged between tunnel endpoints. We describe how processing of incoming and outgoing data is performed. We define the our custom cryptographic protocol that is used to encrypt and authenticate Lynxtun datagrams. Finally, we present a technical evaluation of various aspects of the protocol design and the present the rationale of certain design decisions that we have made.

3.1 Introductory Example

Before providing a formal specification of the protocol, it is illustrative to present a simple introductory example of how the Lynxtun protocol is expected to work. This examples is meant to give a sense of what our solution entails without going into technicalities or striving for robustness.

Alice and Bob are both connected to the internet, and communicate over it using IP. Alice's public IP address is `192.0.2.101` and Bob's is `192.0.113.102`. Alice and Bob communicate using an instant messaging application. The client on each side sends and receives data through network sockets. This data gets placed inside IP packets and routed through the network.

Alice and Bob decide to use Lynxtun to secure their communication. They agree upon the following shared configuration:

Example Lynxtun Configuration

	Alice	Bob
AES-256 Key	K	K
Local IP Address	192.0.2.101	192.0.113.102
Remote IP Address	192.0.113.102	192.0.2.101
Local UDP Port	6060	6060
Remote UDP Port	6060	6060
Local Tunnel IP Address	10.0.0.1	10.0.0.2
Remote Tunnel IP Address	10.0.0.2	10.0.0.1
Dispatch Interval	25 ms	25 ms
Dispatch Size	5000 B	5000 B

Note that this is a simplified version of Lynxtun configuration, and does not include all configuration parameters defined by the protocol.

Besides switching local and remote values on the two sides, the two configurations are identical. This configuration includes the shared AES-256 secret key. Therefore, this configuration should be shared between the two systems through a secure channel.

Alice and Bob each run the Lynxtun endpoint application on their machines. Lynxtun will create a virtual TUN interface called `lynx0` on each host, and assign it the host's local tunnel IP address. The routing table will be configured so that packets being sent to the peer's tunnel IP will get routed to the local `lynx0` device.

Each endpoint each binds to UDP port 6060, and sends fully encrypted Lynxtun datagrams that are 5000 bytes each at 25 millisecond intervals to the corresponding socket on the tunnel peer.

The operation of Lynxtun is strictly periodic. We call each period a round. In each round, the Lynxtun endpoint running on Alice's computer will attempt to read IP packets from the `lynx0` device, encapsulate these in a staged Lynxtun datagram, encrypt the datagram and send it to Bob at the scheduled dispatch time.

Incoming data will be read from the UDP/IP socket. This data will be authenticated, decrypted and (assuming it was a valid Lynxtun datagram sent by Bob) unpacked. The contained IP datagrams will be delivered to Alice's local network stack through the `lynx0` device.

3.2 Definitions

N	Lynxtun datagram size (bytes). Configuration parameter.
δ	Dispatch interval. Configuration parameter.
σ	Freeze window duration. Configuration parameter.
P	Total payload size (bytes). $P = N - 32$.
n	Used payload size (bytes). The total size of the IP packets encapsulated inside the payload.
τ	Datagram timestamp.
ω	Timestamp tolerance. Configuration parameter.
γ	Size of the timestamp list. Configuration parameter.
K	Shared AES-256 Secret Key. Configuration parameter.

3.3 Protocol Design Objectives

The Lynxtun Protocol is a secure IPv4 [1] encapsulation protocol, that can be used to establish a secure point-to-point tunnel between two network hosts over an insecure network.

In Section 1.1.4, we stated that our interpretation of security has three parts. The communication should be confidential, unobservable and undisturbed. We further stated that ensuring confidentiality requires addressing two issues. First, the content of each individual packet should be encrypted. Second, traffic flow confidentiality should be achieved, meaning that patterns that may exist in the underlying communication should not be observable in the network traffic generated by Lynxtun endpoints. This requires regulating all observable features.

Our approach to securing the content of individual IP packets is to send fully encrypted datagrams with no plaintext segments. We refer to these as Lynxtun Datagrams. A Lynxtun Datagram is sent as the payload of a UDP datagram, which itself becomes the payload of an IP datagram. Since the Lynxtun Datagram is fully encrypted, the only plaintext segments of network packets sent by Lynxtun are IP and UDP headers. Our requirement is to that, if Lynxtun Datagrams are recorded and concatenated, the resulting data should be statistically indistinguishable from randomly generated data.

Requirement 3.3.1 (Fully Encrypted Datagrams)**.** Lynxtun Datagrams should be fully encrypted. The concatenation of captured Lynxtun Datagrams should be statistically indistinguishable from randomly generated data.

Due to the fact that UDP does not provide reliable delivery, we have the following additional requirement:

[1] IP is used to refer to IPv4 throughout this thesis.

Requirement 3.3.2 (Independent Datagram Decryption)**.** An encrypted Lynxtun Datagram should be fully self-contained in the sense that it includes all necessary information to fully decrypt and authenticate it.

In order to achieve traffic flow confidentiality, the Lynxtun Protocol uses the Constant Bitrate (CBR) approach. That is, each tunnel endpoint should transmit data at a fixed rate. We define the **Dispatch Interval** and **Dispatch Size** configuration parameters. The dispatch interval specifies the amount of time between consecutive dispatch operations, which we also refer to as the interarrival time of dispatch operations. The dispatch size, also referred to as the datagram size, determines the size of the Lynxtun datagram that is sent.

The dispatch size does not depend on how much data is generated by the underlying communication. The payload of any given Lynxtun datagram can include encapsulated IP packets in addition to padding. The size of the padding is adjusted to ensure that the total size of the datagram remains fixed. Since the entire datagram is encrypted, the data and padding segments of the payload are indistinguishable.

A Lynxtun tunnel consists of two Lynxtun endpoints that share a common configuration. Each endpoint in a tunnel sends and receives data from its tunnel peer. However, the dispatch process of each endpoint should be independent. Each endpoint should regularly transmit datagrams, independently from the times at which its peer sends them. It is not necessary to synchronize the times at which dispatch operations are performed between the two endpoints. As long as each endpoint continues to make dispatch operations at fixed intervals, datagram timestamps observable on the network cannot be used to undermine traffic flow confidentiality.

A related objective is to achieve ergodicity of the dispatch process. This implies that the first dispatch operation should be functionally equivalent to all subsequent ones. There should not be distinct operational phases that result in different dispatch behavior. Ergodicity implies stationarity. The statistical properties of the dispatch process should not change throughout the lifetime of the tunnel.

Realizing this approach successfully also fulfills the requirement for unobservable communication. Since the rate at which data is sent by a Lynxtun endpoint will remain the same even if there is no underlying communication, an attacker cannot detect whether or not actual communication is taking place. The attacker can at most determine the maximum amount of information that could have been exchanged over a period of time.

Ensuring that communication is not disturbed by an attacker requires being able to identify Lynxtun Datagrams that were in fact sent by the tunnel peer and were not modified. Only datagrams that meet this condition should be accepted. There are two aspects of authenticating datagrams. First, the datagram should exhibit proof that it was generated by someone with access to the shared secret, which takes the form of an AES-256 secret key. Second, the Lynxtun Protocol requires that each dispatched datagram is unique. This follows from Requirement 3.3.1. If a given datagram is received more than once, this property implies that only the first is authentic. Therefore, it is necessary to design mechanisms that identify and discard replayed datagrams. These mechanisms can be referred to as replay-attack mitigation countermeasures.

Requirement 3.3.3 (Datagram Authentication)**.** A Lynxtun Datagram D is authentic if and only if it demonstrates proof that it was generated by a party with access to a shared secret, and is not identical to any previously authenticated datagrams.

A hypothetical implementation that is able to adhere to the aforementioned requirements perfectly will be secure. Such an implementation would need to be perfectly deterministic, in that it is always able to execute dispatch operations at precisely the right time. This is not possible to achieve in practice. A successful implementation should be able to limit the deviations in actual dispatch times with a tolerance on the order of several microseconds, as these will be extremely difficult to detect on a network where latencies are on the order of milliseconds.

How successful an implementation is able to regulate its dispatch process depends on numerous factors, which will be discussed at length in later chapters. However, the viability of a successful implementation depends on a protocol design that is conducive to achieving this objective. This is one of the most important design objectives for the Lynxtun Protocol.

For this purpose, it is important that the Lynxtun Protocol is as simple as possible. Most importantly, it is necessary to ensure that all rounds can be treated in the same way. It is imperative that we are able to clearly define the work that has to be done within a single dispatch round, to be able to reason about whether or not this work can be completed in the available amount of time, and for this not to depend on past activity.

It is necessary to acknowledge that the actual dispatch process will neither be ergodic nor stationary, as it will be influenced by external factors. We will discuss ways in which such external factors can be mitigated in later chapters. However, from the standpoint of protocol design, it is necessary to design the protocol such that these requirements are not violated either intentionally or inadvertently by implementations themselves.

One final point to consider is that of efficiency. It should be possible to implement the protocol efficiently. That is, it should be possible to carry out the necessary work as quickly and with the least amount of instructions as possible. There are two reasons for this. First, the smaller the ratio of the time that is necessary on average to complete the work that has to be done in a round is to the time available to do the said work, the less likely it is that the dispatch deadline is missed. This is important for traffic flow confidentiality. Second, an attacker can attempt a denial-of-service attack by flooding an endpoint with a large number of unrelated datagrams. Unless we are able to detect and discard these efficiently, this can lead to the situation where actual communication can no longer take place. Therefore, efficiency is also necessary for undisturbed communication.

3.4 Interface Specifications

A Lynxtun endpoint will have two interfaces, that we refer to as its ingress and egress.

The ingress is a virtual TUN network interface that is integrated into the network stack of the local host. IP packets that are received through the ingress are encapsulated in outgoing Lynxtun datagrams. IP packets that are unpacked from received Lynxtun datagrams are delivered to the ingress.

The fact that the TUN device is subject to standard routing configuration allows Lynxtun to be readily used in different ways. For instance, the tunnel can serve as a secure network proxy by setting it as the default network gateway. Also, it is possible to create secure networks with multiple hosts, where each point-to-point link is a separate Lynxtun tunnel.

The egress is a UDP/IP socket that is used to send and receive encrypted Lynxtun datagrams from the tunnel peer.

The Lynxtun configuration specifies a pair of local and remote **Tunnel IP Addresses**. The local tunnel IP address is assigned to the ingress TUN device. An entry is added to the routing configuration so that IP packets that are sent to the remote tunnel IP address are routed to the TUN device.

The Lynxtun configuration also specifies a pair of local and remote **Public IP Addresses**, and a pair of corresponding **UDP Port Numbers**. The Lynxtun endpoint binds to the local UDP/IP socket specified by the local IP address and port number. It sends encrypted Lynxtun datagrams using the remote IP address and UDP port number. There are no restrictions on which UDP port numbers can be used, and can be different for each host.

3.5 The Lynxtun Datagram

Lynxtun tunnel endpoints exchange encrypted Lynxtun datagrams. In this section we define the structure of the encrypted and unencrypted forms of the Lynxtun datagram. The cryptographic protocol used to encrypt and decrypt Lynxtun datagrams is given in Section 3.9. The limits of the datagram size is covered in Section 3.10.3. Reasons why large datagram sizes might be desirable are discussed in Section 3.10.13.

Since we are defining a network protocol, we follow the network byte ordering convention. All fields are represented in network byte order, which is **big-endian**.

Requirement 3.5.1 (Byte Order)**.** All fields in a transmitted Lynxtun datagram must be stored in network byte order (big-endian).

3.5.1 The Encrypted Lynxtun Datagram

The size of the encrypted Lynxtun datagram is N. It is made up of a 16-byte encrypted header, a 16-byte GCM authentication tag field, and an encrypted payload of size P.

All fields are the output of AES-256 block encryption. As such, the encrypted Lynxtun datagram contains no plaintext fields.

The structure of the encrypted Lynxtun datagram is given in Figure 3.1.

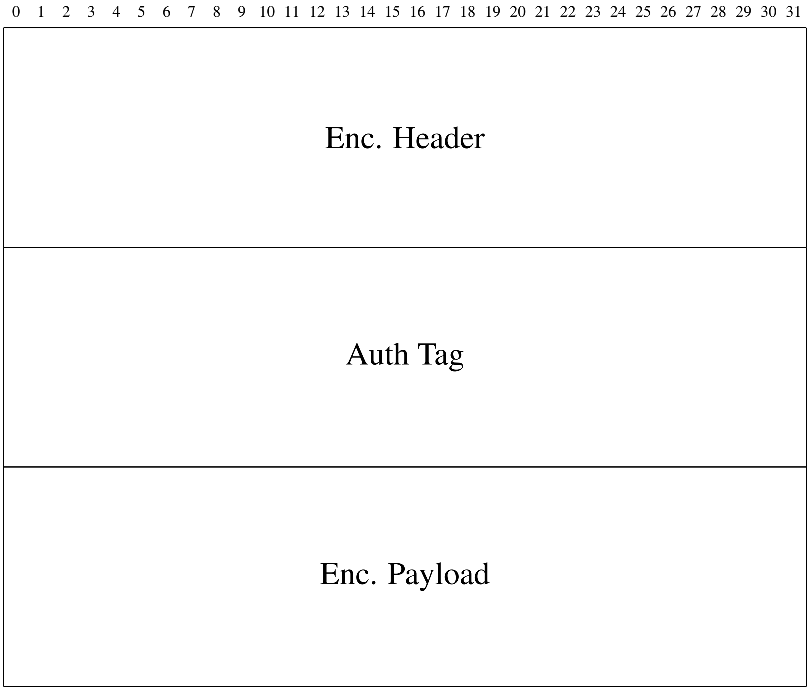


Figure 3.1: The Encrypted Lynxtun Datagram

3.5.2 The Unencrypted Lynxtun Datagram

The structure of the unencrypted Lynxtun datagram is similar to that of its encrypted counterpart, but does not include the GCM authentication tag field. Figure 3.2 shows the structure of the unencrypted Lynxtun datagram.

3.5.2.1 Header Fields

The **Host ID** field identifies the tunnel endpoint. It is defined to be 0 for the tunnel endpoint that has the smaller tunnel IP address, as interpreted as an unsigned 32-bit integer and 1 for the other endpoint.

`ts_sec` and `ts_msec` together constitute the **datagram timestamp**, indicating the time at which the datagram was generated with millisecond resolution. `ts_sec` is the number of seconds since the UNIX epoch (1970-01-01 00:00:00 UTC) and `ts_msec` is the remainder in milliseconds.

Together, the Host ID and datagram timestamp make up a 96-bit nonce that is used as the GCM initialization vector for authenticated encryption, as described in section 3.9.4.

The 16 bits after `ts_msec` is set to 1, represented as an unsigned 16-bit integer. The is necessary for the security of the cryptographic protocol, as discussed in section

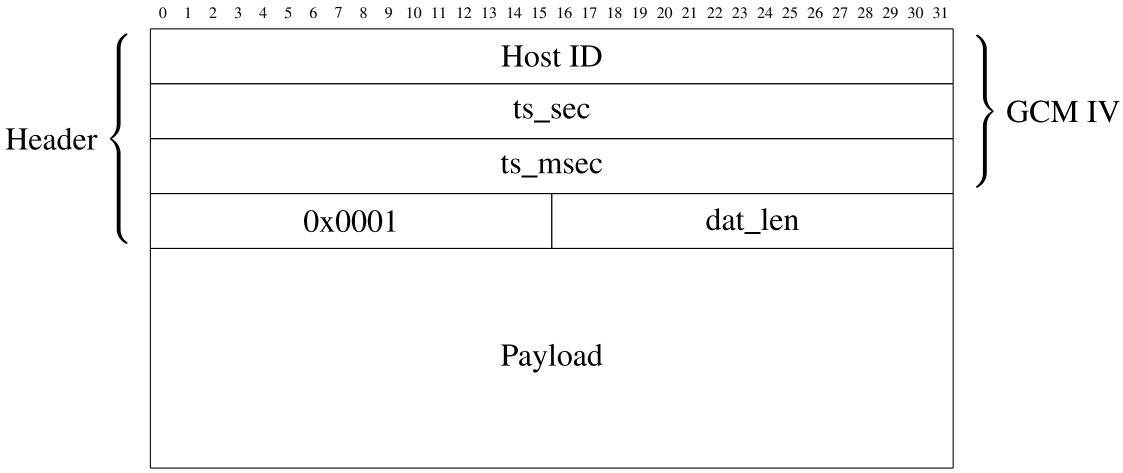


Figure 3.2: The Unencrypted Lynxtun Datagram

3.10.7.

The final 16 bits of the header, `dat_len`, stores the size of the actual data that the datagram holds in bytes, which is n. This is the sum of the sizes of all encapsulated IP packets in the payload.

Note that the total datagram size is not explicitly stated in the datagram header. Since Lynxtun transmits fixed-sized datagrams and the same configuration is shared on both sides, this is known implicitly. Moreover, reading a datagram from a UDP socket will reveal its size. Therefore such a field is redundant.

There are two main reasons why we choose to use millisecond resolution rather than a finer resolution. First, the clock resolution is not the same on all systems, but we can safely assume that millisecond resolution will be available[2]. Secondly, using millisecond resolution rather than microsecond or nanosecond resolution fixes more bits of the datagram header to be 0. This is beneficial for header authentication, that is explained in Section 3.9.4.

Each Lynxtun Datagram is required to be unique. This follows from the security requirements defined in Section 3.3. Uniqueness of IVs are also a requirement of GCM. The implication of the aforementioned design decision is the following requirement:

Requirement 3.5.2 (Minimum Theoretical Dispatch Interval)**.** The dispatch interval must not be less than 1 millisecond. More specifically, the amount of time between the generation of two separate datagram headers should never be less than 1 millisecond. The timestamps of all datagrams should be unique.

In practice, there is no benefit of a setting the dispatch interval below one millisecond. Even a dispatch of one millisecond is unlikely to provide sufficient time to sustain a regular dispatch process. It is more appropriate to set the dispatch interval to be at least tens of milliseconds.

[2] In practice, implementing Lynxtun such that deviations in dispatch times remain on the order of microseconds requires high-precision timers to be available on the system. Without this, TFC will be reduced.

3.5.2.2 Payload

The payload of a Lynxtun datagram contains encapsulated IP packets and padding. IP packets cannot be divided across multiple datagrams.

Requirement 3.5.3 (Fully-Formed Encapsulated IP Packets)**.** Encapsulated IP packets in the payload of a Lynxtun Datagram have to be fully-formed. They cannot be divided across multiple datagrams.

The size of the padding is $P - n$. The Lynxtun Protocol does not impose any requirements regarding the content of the padding. It is allowed to contain arbitrary data. The payload is encrypted, and the cryptographic protocol uses GCM mode of operation such that identical padding plaintext will be mapped to different ciphertext blocks. The padding is discarded by the receiving endpoint. However, depending on how an implementation allocates memory, it is possible that the payload to include sensitive data that should not be sent to the tunnel peer. Therefore, an implementation can choose to zero out the padding of a datagram before it is encrypted.

3.6 Specification for Processing Outgoing Data

The algorithm for processing outgoing data is strictly periodic. Each iteration is identical, and is called a round. Each round ends with a dispatch operation. That is, an encrypted Lynxtun datagram being sent over the network to the tunnel peer.

At the start of each round, there is an empty *staged datagram*. Outbound IP packets received through the TUN device are encapsulated in the payload of the staged datagram.

The structure of a round of outgoing processing is shown in Figure 3.3.

A round can be divided into five distinct stages. These are:

Initialization The staged datagram is cleared. The target dispatch time and the target freeze time are set.

Encapsulation IP packets are added to the payload of the staged datagram.

Preparation The staged datagram is frozen, meaning that no more packets can be encapsulated. The datagram header is generated, and the datagram is encrypted.

Waiting Waiting for the scheduled dispatch time.

Dispatch Execution of the dispatch operation by writing the encrypted Lynxtun datagram to the UDP/IP socket.

Requirement 3.5.3 states that encapsulated IP packets must be fully-formed. If an IP packet that is read from the TUN device file descriptor does not fit into the staged datagram in the current round, then the datagram is said to have reached capacity and is immediately frozen. The client should not attempt to read any more IP packets, even if there is still some unused space in the payload. The IP packet that was read and could not be encapsulated should be cached by the client until a future round. If there is such a cached IP packet at the start of the encapsulation stage, then this must be the first packet encapsulated in the empty payload before any packets are read from the TUN device, after which the cache is cleared. The TUN device MTU must be set to be no greater than the fixed payload size, although it is allowed to be less. This ensures that a cached IP packet will fit into an empty staged datagram.

For the i th round, let R_i be the current time at the start of the round, D_i be the target dispatch time, F_i be the target freeze time, D_i' be the realized dispatch time, δ be the dispatch interval and σ be the freeze window duration.

For the first round, we set $D_0 = R_0 + \delta$ and $F_0 = D_0 - \sigma$. For all subsequent rounds, we have $D_i' > D_i$. That is, the client must not execute the dispatch operation before the scheduled dispatch time. Therefore, the actual dispatch will take place after the scheduled dispatch time. The client implementation should ensure that the difference $D_i' - D_i$ is as small as possible, and does not depend on tunnel activity.

At the start of the i th round for $i > 0$, we tentatively set $D_i = D_{i-1} + \delta$ and $F_i = D_i - \sigma$. Noting that $R_i > D_{i-1}'$, there are three different cases that have to be considered:

1. $R_i < F_i$
2. $F_i \leq R_i < D_i$
3. $D_i \leq R_i$

The first situation describes standard operating procedure. The tentatively set dispatch and freeze times are accepted. The amount of time $F_i - R_i$ specifies the amount of time in which to perform encapsulation. The client repeatedly performs encapsulation attempts until either the staged datagram reaches capacity and a IP packet is cached, or the freeze time is reached. Each encapsulation attempt begins by trying to obtain an IP packet to encapsulate. If there exists a cached IP packet from a previous round, then this is used. Otherwise, the client will attempt to read an IP packet from the TUN device file descriptor. In either case, the client is required to check the current time and only initiate a new encapsulation attempt if the freeze time has not been reached. If data is not available to be read from the Tun device, then the client can wait for data to become available. However, the client has to stop waiting if the freeze time is reached, in which case the encapsulation attempt times out. To summarize, the encapsulation phase ends when one of the following conditions are met:

- An obtained IP packet does not fit inside the staged datagram payload. The staged datagram is said to have reached capacity and the IP packet is cached for a future round.
- Once an obtained IP packet has been added to the staged datagram, the client discovers that the freeze time has passed and does not attempt to obtain a new datagram.

- The client attempts to obtain a new packet, but one does not become available until the freeze deadline is reached and the encapsulation attempt times out.

Note that if an IP packet is obtained and processing begins shortly before the freeze time, then the processing is *not* aborted once the freeze time is reached. The exact amount of time it takes once an IP packet has been obtained until the encapsulation process is complete is variable. Furthermore, once the datagram is frozen, it is encrypted. Encryption also takes some variable amount of time. The freeze window duration should be set by taking this into consideration. That is, it should be sufficiently longer than the time it takes to perform encapsulation and encryption, so that the staged datagram is ready in advance of the scheduled dispatch time. Once encryption has been completed, the client enters the waiting phase. As mentioned above, the waiting phase ends no sooner than the dispatch deadline, and the client should ensure that the time delay between the dispatch deadline and the end of the waiting period is minimal. Once the waiting period ends, the dispatch operation is executed.

If the start of a round was considerably delayed, then it is possible that either the tentative freeze time or dispatch time was already missed. This respectively corresponds to the second and third cases mentioned above. In either case, there is not enough time to encapsulate IP packets and prepare the staged datagram. Instead, a *dummy datagram* that contains no actual data will be dispatched.

How the dummy datagram is generated is up to the implementation, but the requirement is that it is not distinguishable from an encrypted Lynxtun datagram. That is, it should appear to statistically indistinguishable from randomly generated data. A dummy datagram can only be used once. An initial dummy datagram should be generated at startup. Afterwards, the dispatch of a dummy datagram should immediately be followed by the generation of a new one, to be used in the future. This generation takes place *after* the dispatch has been made, and therefore does not induce further delays.

If the start of a round was delayed, this could either be because the actual dispatch operation was delayed, or the dispatch operation was performed at the correct time but there was a delay before control was given back to the Lynxtun process. Therefore, the fact that the round start was delayed does not necessarily imply that an observable change in the dispatch process has occurred. However, this might be the case.

If the freeze deadline was missed but not the dispatch deadline, then it is still possible to dispatch the dummy datagram at the next intended dispatch deadline. If the delay was not due to the previous dispatch, then it is still possible that no externally observable disruption has taken place. The endpoint waits for the dispatch deadline as usual, with the only difference being that the dummy datagram is transmitted instead of an actual datagram.

If the tentative dispatch deadline was missed, then an externally observable disruption in the dispatch process is no longer avoidable. The client should dispatch the dummy datagram as soon as possible. The tentative dispatch deadline is rejected and the client sets D_i to the current time. To explain why this is necessary, consider the following example. Let the dispatch interval be 20 milliseconds. The scheduled dispatch to take place at t milliseconds was missed by 100 milliseconds, and was therefore executed at

$t + 100$ milliseconds. If The next round sets its dispatch deadline based on the missed dispatch deadline as usual, then the new deadline will be $t + 20$, and will have also been missed. This would result in several dispatches being made in rapid succession for at least five rounds, further disrupting the externally observable dispatch process.

If the dispatch interval is not sufficiently long, then this will lead to *thrashing* which is characterized by only dummy datagrams being sent. In such a case, communication will not be possible.

3.7 Specification for Processing Incoming Data

The algorithm for processing incoming data is also iterative. At the start of each iteration, the client attempts to read a UDP datagram payload from the UDP/IP socket. This may or may not be a Lynxtun datagram. The client should assume that it will receive unrelated or corrupt datagrams.

The general procedure for processing incoming datagrams is to perform validation and authentication of the datagram in several steps, decrypt an valid datagram, unpack encapsulated IP packets, and write these to the TUN device file descriptor, thus completing their delivery to the local network stack.

The initial validation checks compare the source address to the tunnel peer, and the size of the datagram to the dispatch size. A datagram is immediately discarded in the event of a mismatch.

The authenticated decryption algorithm is described in 3.9.

Upon being accepted, any packets that are encapsulated in the datagram payload are unpacked. Due to Requirement 3.5.3, we know that IP packets can only be encapsulated as a whole. Therefore, the payload will be the concatenation of zero or more IP packets, followed by padding. The total size of the IP packets is available in the datagram header. The size of each IP packet is available in the encapsulated IP header. Unless the payload is empty, then the start of the payload will coincide with the IP header of the first encapsulated IP packet. This packet is written to the TUN device file descriptor. The size obtained from the IP header points to the IP header belonging to the next encapsulated packet. IP packets are extracted in this way, until the total size of processed packets is equal to the data length specified in the Lynxtun datagram header.

Once a datagram has finished processing, then the iteration is complete and the client waits for the next datagram to be available.

Note that, since Lynxtun uses UDP/IP sockets, Lynxtun datagrams (i.e. UDP payloads) are received as a whole and not partially as in the case of TCP sockets.

3.8 Independence of Incoming and Outgoing Data Processing

The algorithms for processing incoming and outgoing data operate independently.

A single-threaded client can partition the available work time between processing incoming and outgoing data.

A multi-threaded client can run each loop simultaneously on a separate thread. However, a client *must* ensure that the thread responsible for processing incoming data does not contend for CPU time with the dispatch thread, thereby having the possibility to delay a dispatch operation. How this is implemented depends on the client. Possible approaches include thread synchronization, running the threads on isolated CPUs, or running the dispatch thread with greater priority vis-a-vis the system scheduler.

3.9 Cryptographic Protocol

In this section, we present our cryptographic protocol that is used to encrypt and decrypt Lynxtun datagrams. The cryptographic protocol also defines mechanisms for authenticating incoming datagrams.

The cryptographic protocol relies on the security of the AES-256 block cipher. We use AES-256 in a combination of ECB and a modified version of GCM mode. We discuss the reasons why we have chosen these cryptographic primitives in Section 3.10.8

3.9.1 Key Establishment

The Lynxtun Protocol requires a shared AES-256 secret key to be installed on both tunnel endpoints prior to the operation of the tunnel. The protocol does not define mechanisms for generating or sharing the keys.

3.9.2 Construction of the GCM IV

We construct a unique GCM IV that is used for the encryption of a single Lynxtun datagram. The length of the GCM IV is 96 bits, and corresponds to the first 96 bits of the unencrypted Lynxtun datagram. This is the recommended IV length as it can be processed more efficiently and is the most secure [38].

The method that we use to construct the GCM IV corresponds to the deterministic construction method described in the NIST recommendations for GCM [25]. Accordingly, the IV is the concatenation of a 32-bit fixed field and a 64-bit invocation field.

The fixed field is the Host ID. This ensures that no two IVs generated by different endpoints can be the same.

The invocation field is the 64-bit timestamp, which is the concatenation of the ts_sec and ts_msec fields, and represents the timestamp at which the IV was generated with millisecond resolution. IVs generated by a single endpoint will be unique provided that the time between the generation of two IVs is *strictly greater* than one millisecond, and the system clock is not adjusted backwards. Requirement 3.5.2 specifies that the time between two timestamp generation events should always be greater than one millisecond. The implications of adjusting the system clock backwards is discussed in 3.10.5.

3.9.3 Authenticated Encryption

3.9.3.1 Generation of the Datagram Timestamp

The first step in encrypting the staged Lynxtun datagram is to generate the datagram timestamp, based on the current time. This is subject to Requirement 3.5.2. The remaining fields of the datagram header are assumed to have set prior to the start of the encryption operation.

3.9.3.2 Encryption of Datagram Payload

The first 96-bits of the datagram header constitute the IV used for GCM authenticated encryption based on the AES-256 block cipher, using the shared secret key K.

We use a modified version of the GCM algorithm, which we refer to as the early-stopping modification. Normally, GCM calculates an authentication tag over all the ciphertext. Our modification changes this behavior. 16-byte blocks (AES block size) of the payload that contain only padding are not included in this calculation. The payload is nevertheless fully encrypted. If the data length is zero, then the authentication tag will be equal to $E_K(C_0)$, where C_0 is the initial counter value, which is $IV||0^{31}1$. The modification can be described as follows. Pure GCM mode is used to encrypt up to the point where blocks no longer contain actual data. The remainder of the payload is encrypted in AES-256 CTR mode. The IV used by CTR is the last counter value used by GCM incremented by one.

As a result of this modification, the part of the encrypted payload that contains only padding is not authenticated. This is not a problem, since this data will be discarded. The modification allows the receiver to perform authentication and decryption more efficiently, stopping when the end of actual data is reached. This is why we refer to the modification as early-stopping.

The authentication tag produced by GCM authenticated encryption is placed in the encrypted datagram.

The additional authenticated data argument of GCM-mode encryption is not used.

3.9.3.3 Encryption of Datagram Header

The last step is to encrypt the datagram header. The datagram header is a single AES block. It is encrypted using ECB AES-256 block encryption, using the shared secret key K.

Using ECB mode to encrypt the header allows the header to be decrypted independently using only the shared key K. This reveals the IV required to decrypt the payload is available in the header. Therefore, Requirement 3.3.2 is satisfied, and the decryption of a datagram is entirely self-contained.

ECB mode is generally considered to be insecure, since identical plaintext blocks get mapped to identical ciphertext blocks. In our case, it is ensured that the datagram header is unique, so ECB mode encryption is never performed on the same plaintext block more than once.

3.9.4 Authenticated Decryption

Received datagrams are subjected to authenticated decryption. The authentication takes place in several steps. The aim is to detect and discard invalid datagrams as early as possible, and as efficiently as possible.

Datagrams are received through the egress UDP/IP socket. There is no guarantee that a received datagram is an authentic Lynxtun datagram that was sent by the tunnel peer. As specified by Requirement 3.3.3, there are two conditions that have to be met in order for a datagram to be accepted as authentic. First, it has to be demonstrable that it was generated by someone that has access to the shared secret key. Second, it has to be different from all previously accepted datagrams.

The definitive check as to whether the datagram was encrypted by the tunnel peer is full GCM authenticated decryption. This has to be done before a datagram is accepted. However, full GCM authenticated decryption is a relatively expensive operation. There are a number of checks that can be done efficiently before this is done. While these checks do not provide conclusive evidence of authenticity, datagrams that are discarded as a result of these checks are guaranteed to be unauthentic.

The second requirement, which states that an identical copy of an authentic datagram that is received at a later time is not authentic, requires an additional mechanism. The Lynxtun protocol defines the **Timestamp Tolerance** ω and **Timestamp List Size** γ configuration parameters for this purpose. The implementation maintains a list of timestamps belonging to recently accepted datagrams. The size of this list is γ. The list is initially empty.

3.9.4.1 Preliminary Checks

Reading a datagram from the UDP/IP socket will reveal its origin. If the IP address and port are not consistent with values defined for the tunnel peer in the configuration, the datagram is discarded without further processing.

Similarly, a datagram is immediately discarded if its size is not equal to the dispatch size N.

3.9.4.2 Authentication of Datagram Header

We perform AES-256 decryption on a single block using the shared AES-256 key K. Unless the data was encrypted using the same key, then the result of the decryption operation will be random.

Decrypting the datagram header reveals the fields it contains. This contains all the information necessary to fully decrypt a datagram, in accordance with Requirement 3.3.2.

The receiving endpoint checks the current time after decrypting the header. Call this t. The timestamp in the decrypted header, τ is compared against this value. The datagram is dropped unless $|\tau - t| < \omega$, where ω is the timestamp tolerance. If this condition is met, then τ is compared to the oldest timestamp in the list of recently accepted timestamps. If τ is older, then the datagram is dropped. Next, τ is compared against all recently accepted datagrams in the list. If it matches any of them, the datagram is dropped. This ensures that a duplicate of a previously accepted datagram will be rejected.

The checks that are made to authenticate the datagram header are as follows:

1. Does the Host ID match the Host ID of the tunnel peer?
2. Is the `ts_sec` value in an acceptable range, based on the timestamp tolerance.
3. Is the `ts_msec` value smaller than 1000?
4. Is the 16-bit value before the data length equal to 1?
5. Is the 16-bit data length field smaller than the payload size?

Unless the header was generated using the secret key K, the probability that each of these conditions will be true are given below.

Condition	Probability
1	2^{-32}
2	$2\omega \cdot 2^{-32}$
3	$1000 \cdot 2^{-32}$
4	2^{-16}
5	$P \cdot 2^{-16}$

The probability that a datagram header that was not generated by the tunnel peer will meet all of these conditions is $125\ \omega P \cdot 2^{-124}$. We are therefore able to reliably identify and discard datagrams after performing AES-256 decryption on a single block.

Note that, in order to reach this stage, the attacker should have set the datagram size correctly and also spoofed the IP address of the tunnel peer.

3.9.4.3 Decryption and Definitive Authentication

The next step is to perform GCM authenticated decryption with our early-stopping modification. The GCM IV to be used is available in the decrypted header. The early-stopping modification causes the authentication tag to only be calculated over AES blocks that contain actual data, and not over blocks that contain only padding. The length of the actual data is available in the decrypted header. This increases efficiency of the authenticated decryption. Decryption of padding blocks is not done, as doing so would be redundant.

After performing decryption, we compare the calculated authentication tag against the received authentication tag. The datagram is dropped if the two values do not match. This situation would imply that an attacker has captured an authentic datagram and sent it after modifying its payload. Moreover, the modified datagram would have to be received before the original datagram. Otherwise, it would be discarded since it contains an invalidated timestamp.

If the calculated and received authentication tag values match, then the authentication is complete and the datagram is accepted. At this point, the datagram timestamp is added to the list of recently accepted timestamps. If the size of the list is smaller than γ (which will be true when the tunnel is first created), it is appended to the end of the list. Otherwise, it replaces the oldest timestamp currently in the list.

3.10 Evaluation of Protocol Design

3.10.1 CBR Dispatch Process

The Lynxtun protocol defines a constant bitrate (CBR) dispatch process. In this section we explain why we believe this is the most suitable approach.

We have mentioned in the Related Work section the results due to Houmansadr et al. [32] that show why imitation based approaches are likely to be unsuccessful. Using an imitation-based approach to regulating the dispatch process would therefore likely lead to an attacker easily identifying that Lynxtun is being used. While this alone does not necessarily mean that security has been compromised, it would mean that the additional complexity required is not justified.

Alternatively, we could have randomly changed the effective dispatch interval and dispatch size for each round. While this *could* make it slightly more difficult to detect that Lynxtun is being used, it would also make it much more difficult to ensure consistent real-time behavior.

Ultimately, Lynxtun is secure as long as the distribution of the dispatch process does not depend on that of the underlying communication. While it is theoretically possible to shape this distribution freely without violating this requirement, the CBR approach is most suitable for ensuring that it is not violated in practice.

3.10.2 Regarding Rekeying

Since we exclusively use 96-bit IVs that are the result of deterministic construction, the constraints on the number of invocations stated in [25] do not apply. However, suppose that these constraints did apply. Since the invocation field is 64 bits, then we should not perform more than 2^{64} invocations. We perform one invocation per datagram, and we limit the rate at which datagrams can be generated to 1 datagram per millisecond (see discussion regarding IV construction above). This means less than 2^{10} invocations are performed per second. This corresponds to less than 2^{35} invocations per year of continuous operation. Meaning that it would take more than 2^{29} years of continuous operation to reach the maximum number of invocations allowed. Under these circumstances, we can say that there are no rekeying requirements and that the shared key can be used directly for the lifetime of the tunnel, provided that the clock that is used to generate the timestamps is not adjusted backwards, in which case the key should be invalidated.

3.10.3 Datagram Size Limits

The size of the payload is set by the sender. The `dat_len` cannot be larger than the size of the payload. When the dispatch process is regulated to fix the size of the dispatched datagrams, the payload size is specified as a configuration parameter. The difference between the payload size and the actual data size is the size of the padding.

The total size of an encrypted datagram is its payload size plus 32 bytes for the header and the authentication tag.

The datagram size or payload size is not explicitly specified in the datagram. Lynx datagrams are sent using UDP/IP. Receiving a UDP datagram through a socket will yield the UDP payload (which is the Lynxtun datagram) in addition to the size of the UDP payload (which is the size of the Lynxtun datagram). This is because UDP datagrams are read from the socket as a whole and not partially. Therefore, we do not need to explicitly specify the total length.

The data length field stores the length of the encapsulated data as bytes as a 16-bit unsigned integer. Therefore, the maximum theoretical data length is $2^{16} - 1 = 65535$ bytes.

The GCM counter value that is used to generate the first keystream block used to encrypt the first plaintext block is $IV \mid\mid 2$, where IV is the first 96-bits of the unencrypted header. This value is incremented once for each plaintext block encrypted. If x is the data length of a datagram, then the datagram header is $IV \mid\mid 2^{16} + x$. Therefore, we limit the largest counter value that could be used to $2^{16} - 1$, corresponding to block $2^{16} - 2$. Since each block is 16 bytes, the theoretical upper bound for the payload size is $2^{20} - 32 = 1048544$ bytes, and the upper bound for the padding size is $1048544 - 65535 = 983009$ bytes.

In Lynxtun, we send datagrams using UDP/IP. The maximum size of an IP datagram is 65535 bytes. The longest IP header is 60 bytes, and the UDP header is 8 bytes. This means the maximum size of an encrypted lynx datagram is 65467 bytes. Since 16 bytes are used for the datagram header and another 16 bytes are used for the authentication tag, this leaves 65435 bytes to use for the datagram payload. As a result, for the purposes of our implementation, the maximum size of the lynx datagram payload is 65435 bytes that can hold up to 65435 bytes of actual data and the remainder will be padding.

3.10.4 Detecting Data Modification

Suppose an attacker captures a datagram being sent from Alice to Bob. If Bob modifies the encrypted header, then this modification will be detected as part of the procedure for authenticating the header, and the datagram will be dropped.

If Bob leaves the header intact and modifies the payload, there are two possibilities. If the original datagram was received and accepted before the modified datagram, then the latter will be rejected because it contains an invalidated timestamp. Suppose that the modified datagram is received earlier. Again, there are two possibilities. If the modification was made to a part of the encrypted payload that consists of padding, then the datagram will be accepted; but the modified part was done to a part that would be discarded anyway. Therefore, security is not affected. If the modification was done to a part that coincides with actual data, then the authentication tag calculated over the ciphertext will not match the value specified in the header. Therefore, the datagram will be rejected.

The fact that the GCM IV is included in the datagram header, and that this datagram is encrypted means that it is not necessary to specify the datagram header as *additional authenticated data* input to the GCM operation.

3.10.5 Time Synchronization

The Lynxtun Protocol requires that the system time on each tunnel endpoint to be synchronized. How this synchronization should be achieved is not defined by the protocol. However, there are certain operational and security implications that should be mentioned.

Checking the timestamp of a received datagram to the current time as described in Section 3.9.4 involves comparing time values generated on two separate machines. Even if the two clocks are perfectly synchronized, there will be a difference in the timestamps, primarily due to network latency. Small differences in the two clocks are accommodated by the timestamp tolerance parameter. This is the reason why timestamps that are greater than the current time when the check is made are also allowed. However, if the difference in the two clocks is significant, then datagram authentication will consistently fail.

Adjusting the clock backwards poses a security problem. This is due to two reasons. First, it invalidates the uniqueness of GCM IVs, as it becomes possible for the same timestamp to be used to generate a new datagram timestamp. Second, it undermines the replay attack prevention mechanism, as old timestamps become valid again.

In general, the system clock should *not* be adjusted backwards while the tunnel is in use. More specifically, even if the tunnel is currently not being used, the time should not be set backwards beyond the timestamp of the most recently sent datagram. If this is necessary, then the AES-256 key should be invalidated and a new key should be used.

3.10.6 Replay Attack Mitigation Mechanisms

We discuss the replay attack mitigation mechanisms specified by the Lynxtun Protocol in Section 3.9.4. Here, we explain the rationale behind our approach along with some implications.

First, consider the following simple mechanism. The endpoint records the timestamp of the most recently accepted datagram. When a datagram is received, its timestamp is compared to this value and the datagram is dropped unless it is more recent. If the datagram is accepted, then its timestamp replaces the recorded value, so the recorded value always represents the timestamp of the most recently accepted datagram. This completely eliminates the possibility of a replay attack. However, it is unsuitable for the Lynxtun Protocol. Suppose we did not persist the timestamp information on disk. Restarting a Lynxtun process would cause this state to be lost. At this point, it becomes possible for an attacker to replay a captured datagram. Since there is no timestamp value to compare it to, it will be accepted. Moreover, if the tunnel endpoint does not send any new datagrams, then the attacker can keep replaying previously captured datagrams in the same order that they were captured. This will likely result in many of them being accepted, as they will have increasing timestamps. We could avoid this problem by persisting the state, but this would require I/O operations to be performed. A primary design goal for the Lynxtun Protocol is to make it conducive for implementations to attain deterministic runtime behavior. Requiring state to be persisted is incompatible with this goal. This is the reason for the *timestamp tolerance* parameter. It imposes a limit to how old a timestamp can be. Specifically, if we shutdown the Lynxtun process and wait 2ω seconds before restarting it, then it is guaranteed that any datagrams that might have been captured by an attacker now have invalid timestamps.

The reason why we keep a list rather than a single timestamp is because Lynxtun is an unreliable network protocol. This means datagrams can be received in an order different from which they were generated and sent. If we set $\gamma = 1$, then any reordering would necessarily lead to discarding an otherwise legitimate datagram. Allowing for larger values of γ allows us to accommodate reordering. However, since datagrams are dispatched at regular intervals and not immediately after one another, reordering is not very likely. Therefore γ can be small. Since the list is initially empty, the probability that a reordered datagram will be dropped is higher when the process first starts, but will quickly drop as the list becomes populated. We should also note that, since the dispatch interval is fixed, the number of datagrams that could be sent within the timestamp tolerance is known. Setting γ beyond this will not have any effect, other than decreased performance, since checking against the oldest timestamp in the list becomes redundant.

In order for this mechanism to function, it is necessary for the system clocks on the two tunnel endpoints to be synchronized, within the margin allowed by the timestamp tolerance. Implications of time synchronization are discussed in Section 3.10.5.

3.10.7 Security of the Datagram Header

Let x be a 16-bit unsigned integer representing the length of the data in the payload in bytes.

The datagram header can be represented as $IV||0^{15}1||x$. There is an important reason for setting the bit before x to be 1.

Consider the case where x is 2. Then the first keystream block that is XORed with the first plaintext block is $E_K(IV||0^{31}2)$. If we did not set the bit before the x to 1, then this would be identical to the encrypted datagram header, which is seen by the attacker. In such a case, an attacker could simply XOR the first ciphertext block with the datagram header to recover two bytes of data.

Similarly, suppose that x is 1. Then the datagram header would be identical to $E_K(IV||C_0)$, where C_0 is the initial GCM counter value. This is the value that gets XORed in the last step to produce the Auth Tag, so the last XOR operation can be undone by an attacker.

Since the maximum payload size corresponds to 4090 blocks, the maximum counter value will be $IV||4091$, where 4091 is represented as a 32-bit integer. Due to the fact that the last 32 bits of the header are a concatenation of 1 represented as a 16-bit integer and the data length, the range of values that the header can take start from $IV||64536$. This prevents a collision from occurring.

Another point is that the values in the datagram header are predictable. A lynx datagram that is seen on the network is likely to have a timestamp value that is close to the time that the observer sees the datagram. Suppose there is a 10 second window. The millisecond field can be considered random. Suppose the payload size is 10000 bytes. The Host ID field has two possible values. Then there are 2×10^8 possible values for the datagram header. So the probability that an attacker can guess the plaintext datagram header is not negligible. However, none of the data contained in the datagram header is secret. Also, without knowing the key, there is no way for the attacker to verify whether or not a plaintext matches the observed ciphertext. Finally, even if the attacker knew which plaintext headers corresponded to which ciphertext headers, AES-256 is secure against known plaintext attacks.

3.10.8 Choice of Cryptographic Primitives

The two cryptographic primitives that we use in Lynxtun are the AES-256 block cipher, and the GCM mode of operation based on AES-256. As such, there is only a single shared secret in the system, which is the symmetric AES-256 key.

AES [21, 44] is a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The original name of the AES cipher is the Rijndael block cipher. It was submitted to the AES competition held by NIST and was selected as the winner, after which point it took on the name AES. AES is included in the ISO/IEC 18033-3 standard. It has been widely studied by the cryptographic community and is secure against all known cryptanalytic attacks.

GCM [38] is a mode of operation that can be used with AES-256, that provides authenticated encryption. [38] presents a discussion on the security of GCM. It is also a NIST recommendation [25]. GCM is essentially HMAC that does encrypt-then-MAC. There is no need for a separate secret key. Both encryption and authentication depend on the same key, which in our case is the AES-256 key. Since this is among the primary design goals of GCM, its security has been widely studied in this context. This allows us to use a single AES-256 key that has to be exchanged prior to establishing the tunnel, and that does not have to be rekeyed.

AES supports key sizes that are 128, 192 and 256 bits. Longer key sizes offer greater security at the cost of performance [10]. However, we find that our AES-256 is sufficiently performant. Using a longer key is particularly desirable due to the fact that rekeying is not done.

One of the primary benefits of using GCM as opposed to a separate cryptographically secure hash function, such as SHA256, in order to implement HMAC is performance. The time it takes to process a block of data using SHA256 is on the same order of magnitude as performing block encryption on the same amount of data with AES-256, when software implementations are used. We need to perform decryption in any case. GCM incorporates the decryption into its authentication procedure, and calculating the authentication key comes down to providing a field multiplication operation for each block of data. In Appendix D, we show how this can be efficiently implemented in software. Therefore, using GCM has performance benefits.

Another advantage due to the use of GCM is that its recommended IV length is 96-bits. This fits nicely into the datagram header. Furthermore, our method of GCM IV construction (see Section 3.9.2) allows us to incorporate the datagram timestamp into the IV. This timestamp is also the basis of replay-attack prevention mechanisms. As such, using GCM with AES-256 presents us with an elegant approach.

Unlike SSL/TLS, IPsec or SSH, the Lynxtun protocol does not support the use of different ciphers. Its cryptographic protocol is fixed. In this sense, Lynxtun is *cryptographically opinionated*. The Wireguard technical whitepaper [24] argues that trying to have cipher agility, which means supporting different ciphers, is detrimental to security. Like Lynxtun, Wireguard is also cryptographically opinionated. In general, our approach to the problem is as follows. A cipher is either cryptographically secure, or it is not. In certain cases, such as embedded programming where hardware resources might not be enough to run AES-256, lightweight crypto algorithms become important. However, in the context of a general-purpose computer, AES-256 is usable. Furthermore, AES-256 is widely accepted by the cryptographic community as being secure. If we are able to use AES-256 to secure our communication, then adding support for another cipher is redundant. Furthermore, the additional complexity required to add support for other ciphers increases complexity, which is itself detrimental to security. The implication of this approach is that, if AES-256 or GCM is ever found to be insecure, then the Lynxtun Protocol would have to be redesigned. This is a risk that we are willing to accept.

3.10.9 Regarding the GCM Early-Stopping Modification

The modification to GCM that we describe above allows the receiver to perform authentication and decryption more quickly, by not wasting time authenticating or decrypting blocks that only contain padding, which will be discarded.

This would have caused a problem if an attacker was able to check segments of varying length against the authentication tag to learn the length of the data segment. This is not the case since calculating the authentication tag requires knowledge of the secret key.

The main reason for this modification is to increase the rate at which datagrams are dequeued from the RX queue of the UDP/IP socket, making the system more resilient against DoS attacks. The modification is especially relevant to the case where large Lynxtun datagrams are being sent, that contain mostly padding.

There is an important implication with regards to implementations that use a single thread. More specifically, any implementation where incoming data is processed on the same thread that dispatches are made. The duration of the decryption operation *will* depend on the amount of actual data in the payload. Such an implementation is required to ensure that this will not lead to the dispatch deadline being more likely to be missed if the incoming data rate is high. On the other hand, a multi-thread implementation should ensure that the dispatch thread is shielded from the workload of the thread processing incoming data.

3.10.10 Reasons for a Connectionless Protocol

The Lynxtun protocol is *not* a connection-oriented protocol (COP). As such, there are no mechanisms for the establishment, management, and closure of connections. However, there is an implicit connection between the two tunnel endpoints. In this section, we discuss this design decision with regards to its security implications.

A COP depends on the concept of a session that is associated with shared persistent state that defines the context in which incoming network messages should be processed. This state should be consistent between the two endpoints. This is not trivial, as the state exists on separate machines. Mechanisms have to be implemented to ensure consistency. Implementing such mechanisms should be justified by having certain benefits associated with using a COP. In the case of Lynxtun, we have decided that no just justification exists. To the contrary, defining Lynxtun as a COP would be detrimental in terms of security.

COPs are often represented as *state machines*. This requires clearly identifying and labelling all possible states that the system can be in at any given time, which transitions are allowed between these states, under what conditions they are executed, and how they are performed. This model has to be robust and the implementation must strictly adhere to it.

There are two primary benefits of a COP. The first is that it allows defining a particular context in which related IP packets are to be processed. The second is that this context is not fixed, but adaptable. That is, the rules governing the communication can adapt to changing circumstances.

In Lynxtun, there exists an implicit logical connection between the two endpoints. The shared state associated with this connection is the shared configuration parameters that are installed on each of the clients. However, this shared configuration is shared prior to tunnel operation and remains fixed throughout the lifetime of the tunnel.

If there was no implicit connection, then a connection would have to be established during startup. This would require one endpoint to initiate a connection request, and the other to respond to it. Together, they would transition from an initial state where there is no connection to one where there is. This is not desirable with regards to our requirement that the tunnel behavior should be stationary. There would have to be at least two different modes of operation, one for establishing a connection, and the other for normal tunnel operation.

This sort of coordination between two hosts is quintessential to COPs. Control messages have to be exchanged in order to coordinate transitioning from one state to another. Since Lynxtun is not a COP, there is no reason for the two endpoints to directly interact with one another. Due to our specification, the operation of an endpoint does not change even when the peer becomes offline. There is no inherent difference between the cases where there is no tunnel peer. Operation will resume as usual. Dispatches will continue to be made regularly. The only salient change will be that there will never be any incoming datagrams to process.

When we consider whether there is any reason to alter state while in operation, we see that this would be detrimental to security. By definition, the dispatch process should be unrelated to tunnel activity. Therefore, the parameters should remain fixed. It might be necessary that the requirements change and a greater bandwidth is required. This will result in an observable shift of the dispatch process. If this is necessary, then the user should restart the tunnel with modified configuration parameters, being aware of its possible implications. However, since this is discouraged unless absolutely necessary, there is no benefit of doing this while the system is in operation.

Unless the state machine is limited to two states, then additional mechanisms would be required to detect if the system has entered an inconsistent state and apply corrective actions if this is found to be the case. The fact that the system has become inconsistent indicates that something unexpected happened, and therefore handling this eventuality is a complicated process.

A related issue is whether or not to use a connection-oriented transport layer protocol, such as TCP. We discuss this issue in the next section.

3.10.11 Unreliable Delivery and UDP

Reliable delivery implies that data can be treated as a stream, and is received in the same order that it is sent. The Lynxtun protocol is based on the unreliable UDP transport layer protocol, and does not implement reliable delivery itself. As such, it is an unreliable network protocol.

Implementing reliable delivery requires a connection-oriented protocol. While Lynxtun is not a COP, the implicit connection could still be used to implement reliable delivery. However, this would increase the complexity of the protocol. More importantly, there is a fundamental reason why it would be wrong to design Lynxtun as a reliable protocol, which we will explain below.

Lynxtun is a Layer-3 encapsulation protocol. The IP protocol itself is unreliable. It relies on a best-effort delivery principle, without providing any guarantees. Packets are routed independently through the network. They can be dropped, reordered, or duplicated.

As a Layer-3 encapsulation protocol, Lynxtun represents a segment of an IP network that is fundamentally unreliable. Therefore it is not possible to achieve end-to-end reliability. Making Lynxtun reliable would be limited to the communication between its endpoints, which would correspond to having a reliable segment within an unreliable network, to which there is no benefit in terms of the IP communication.

The Lynxtun cryptographic protocol is defined such that each datagram can be encrypted independently. This is due to Lynxtun being unreliable. Making Lynxtun reliable would mean that this does not have to be the case, but there is no benefit from doing so. The encrypted Lynxtun header is 16 bytes. The TCP header that is primarily used to implement reliability is 20 bytes. Furthermore, the Lynxtun header also contains the timestamp which is necessary for replay-attack prevention, and therefore cannot be eliminated. A reliable protocol could implement alternative mechanisms for replay-attack prevention, but overall it is not likely that any gains in terms of overhead reduction would be possible.

The only actual benefit from having reliable delivery would be the possibility to divide IP packets across multiple datagrams, thereby increasing average datagram utilization. However, as we explain in Section 3.10.13, it is possible to use leverage IP fragmentation in order to optimize datagram utilization even with an unreliable protocol.

There is one other conceptual reason why reliable delivery is unsuitable. The encapsulated IP packets will either be carrying datagrams belonging to a reliable transport layer protocol such as TCP, or an unreliable transport layer protocol such as UDP. In the case of UDP, then the sender has already accepted the possibility for this packet to be lost. However, if it is TCP, then the sender will be in charge of retransmission if an acknowledgement is not received. If we also independently retransmit a Lynxtun datagram containing such a packet, then we would be wasting network resources. In principle, encapsulating a reliable communication stream through another reliable communication stream reduces efficiency [54, 58].

The same arguments are also valid if we do not implement reliability within Lynxtun, but instead have it depenend on TCP, which is reliable. This has the additional downside of making Lynxtun datagrams subject to TCP congestion control mechanisms, which would disrupt the regularity of the Lynxtun dispatch process.

Therefore, Lynxtun is designed to be used only with UDP/IP. This is the approach also taken by WireGuard [24]. OpenVPN supports UDP *and* TCP, but UDP is preferable [58].

3.10.12 Time Overflow

UNIX time, which is the amount of seconds since the epoch 1970-01-01 00:00:00 UTC represented as an signed 32-bit integer value will overflow in 2038. We represent this value as an unsigned 32-bit integer value, so this will not overflow until 2106. Being able to cleanly separate the `ts_sec` and `ts_msec` fields into two 32-bit words facilitates implementation. In all likelihood, this protocol will have become utterly irrelevant by 2106. If for some reason it does not, then unused bits from the `ts_msec` field can be used to increase the number of bits used to store `ts_sec`.

3.10.13 Regarding Large Datagrams

The Lynxtun datagram is allowed to be larger than the network MTU. IP fragmentation and reassembly will be handled by the network stack. This has certain benefits.

In general, it gives us more control over the target data rate. The bandwidth can be kept constant by balancing longer dispatch intervals with larger datagrams.

In Section 3.10.11 we mentioned that large datagram sizes can be used to optimize datagram utilization. We freeze the staged datagram when a IP packet read from the Tun device does not fit. In the worst case, this will lead to leaving $N - 1$ bytes unused in the datagram even though there was data available to be sent, where N is the maximum size of the IP packet, which is bounded by the Tun device MTU. We can also configure the Tun device MTU. By increasing the ratio of the datagram size to the Tun device MTU, datagram utilization improves, provided that datagrams are consistently reaching capacity in most rounds. On the other hand, if datagrams are not usually filled by the time of the freeze deadline, then increasing the datagram size will decrease datagram utilization. The optimal case will depend on the distribution of the underlying communication.

There is an important benefit of relegating the fragmentation to the network stack rather than handling it within Lynxtun. This is due to the burden of knowledge principle. The reassembly code of the network stack *does* have access to the underlying communication, but is overall far more isolated from it than the Lynxtun client itself.

By transferring complexity further away from the sensitive information, we reduce the risk of tainting the dispatch process.

If fragmentation takes place, then an attacker will not see entire Lynxtun datagrams but IP fragments in rapid succession. However, this has no implication on security.

One final point to make on this issue is that increasing datagram size when communicating over networks that are prone to high data loss will be detrimental to tunnel operation. This is because the loss of a single IP fragment would result in the loss of an entire Lynxtun datagram, regardless of how many fragments do arrive. By using datagram sizes that are smaller than the network MTU, then the loss of an IP packet does not affect the delivery of other packets. This should be taken into consideration when deciding on the correct dispatch size.

3.10.14 Detectability of Lynxtun

Detectability means being able to classify Lynxtun datagrams as such. Whether or not we wish to hide the fact that we are using Lynxtun depends on the context in which we are using it. The design of Lynxtun makes it relatively difficult to detect. While this is a desirable quality, undetectability is not in itself a design objective. Our objective is to hide the traffic flowing through Lynxtun and not Lynxtun itself.

One specific area where detectability becomes important is censorship circumvention. A censor that wants to block Lynxtun will have to reliably classify network traffic as Lynxtun datagrams. The Lynxtun datagrams are fully encrypted. The only unencrypted parts of Lynxtun traffic are the IP and UDP headers, which contain no information that can be linked to Lynxtun. Therefore deep packet inspection cannot reliably detect Lynxtun datagrams. The UDP ports that are used is not fixed, so port-based identification is not possible. The format of *all* datagrams are the same. There is no connection-establishment stage where unencrypted data is sent.

Contrast this with OpenVPN. OpenVPN also uses encryption and does not use fixed ports. However, it is possible to identify OpenVPN traffic reliably based on datagram content [43]. Connection initialization messages such as "Client Hello" and "Server Hello" can be easily matched. Even after the session has been established, the unencrypted opcode field can be used to reliably identify OpenVPN communication.

Many protocols, like OpenVPN, include some plaintext segments in their datagrams. While these can be used to identify those protocols, observing that there are not plaintext fields whatsoever can suggest that Lynxtun is being used. A censor might decide to treat datagrams that appear to be entirely encrypted (or entirely random) as suspicious and block them altogether.

Another reason why it is hard to detect Lynxtun is because the endpoints do not respond to incoming requests. An OpenVPN server can be detected by trying to connect to it and seeing how it responds. In Lynxtun, there is no concept of a request. There is nothing that an attacker can send, even if they know the secret key, that would make the Lynxtun endpoint send data in response, confirming that it is in fact a Lynxtun endpoint.

The most viable approach to classifying Lynxtun traffic is by doing statistical analysis of the observed dispatch process. Since the regulated dispatch process transmits the same amount of data at fixed intervals, such an approach is likely to succeed. Again, the result would not be definitive. Provided that we are able to regulate successfully, then the result should be indifferent from sending entirely random data at fixed intervals. However, since that sort of behavior is not expected, the attacker can simply assume that traffic that matches this description should be blocked.

3.10.15 Denial of Service Attacks

First of all, we assume that an attacker cannot disrupt communications between Alice and Bob, meaning preventing packets from being delivered. If they are able to do this, then there is nothing that the protocol can do to avoid it. Therefore, we focus our attention on an attacker sending malformed datagrams to keep the protocol client busy so that legitimate datagrams will be dropped.

Ultimately, this is a matter of how fast an attacker can fill up the RX queue attached to the UDP/IP socket through which the tunnel client receives datagrams, and how fast the tunnel client can drain that queue. If the queue overflows, then incoming legitimate datagrams will be dropped. Optimizing the rate at which we drain the queue is mainly the responsibility of the client implementation. However, the protocol specification has been designed to make it easier for the client to do this. This is the primary reason for the multi-layered authentication strategy and the GCM modification that makes early stopping possible. The aim is to make it possible to recognize corrupt datagrams as soon as possible, having done as little work as possible, in order to avoid having to do any more work than is strictly necessary. Without this consideration, many of the things that we have discussed above become unnecessary. We could simply rely on full GCM authenticated decryption to make our decision on whether or not to accept a received datagram. But this would make it much easier for an attacker to be able to mount a successful denial of service attack.

Of course, we are talking about denial of service attacks that result from the inability of the tunnel client to drain the RX queue of the socket. It is also possible that the bottleneck will turn out to be a network component along the way that cannot handle the amount of traffic flowing in. In this case, there isn't anything that we can do.

3.10.16 Regarding IPv6

IPv4 and IPv6 are the two preeminent network layer protocols that are responsible for most of the internet infrastructure. IPv6 is the successor of IPv4, although IPv6 adoption is still limited and IPv4 is still widely used. While we have focused our attention on IPv4 traffic, the Lynxtun Protocol can easily be adapted to support IPv6 packets too.

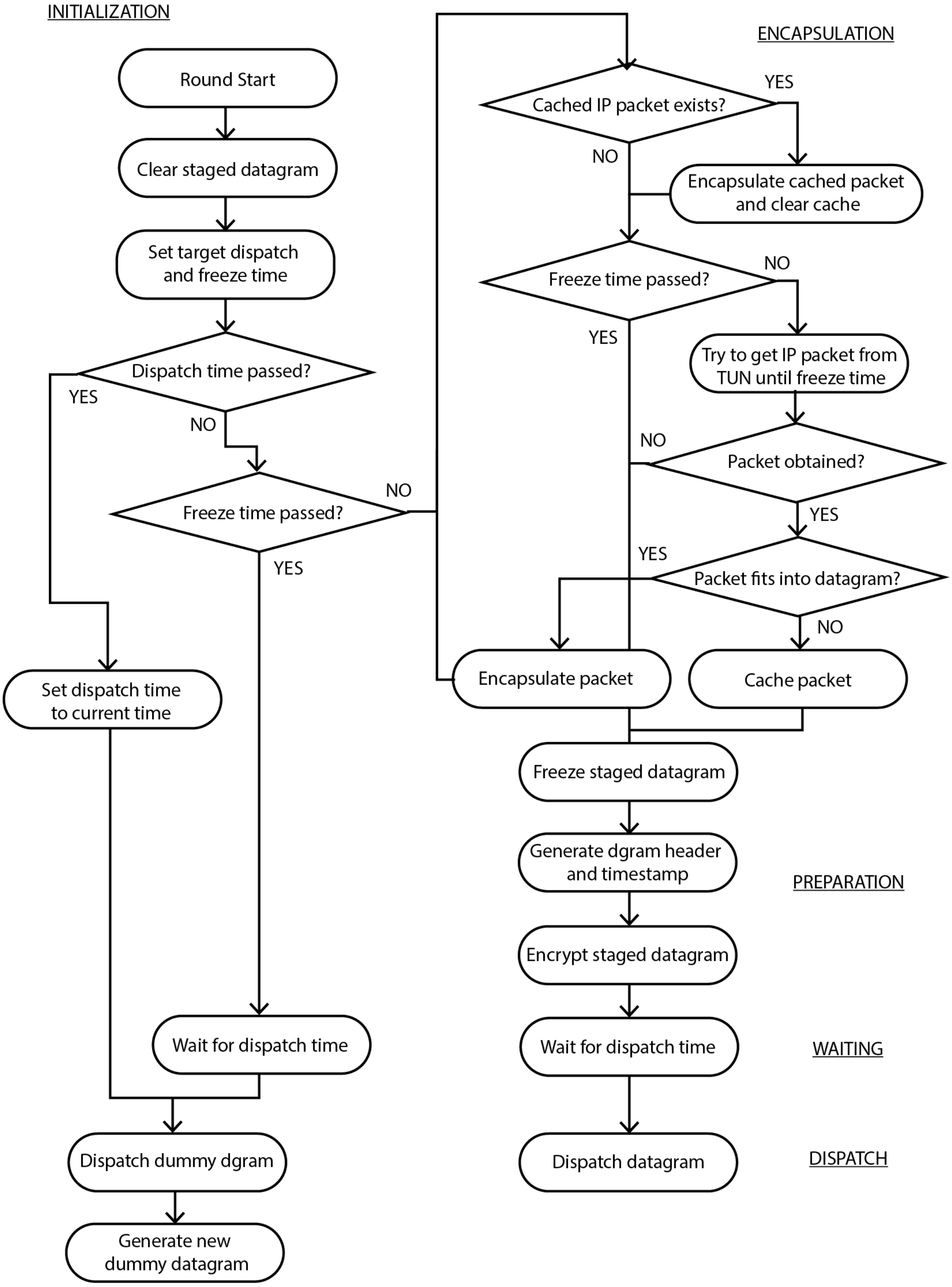


Figure 3.3: Outgoing Processing Round

CHAPTER 4

LYNXTUN AND ITS ENVIRONMENT

4.1 Realtime Requirements

An algorithm is the expression of the steps required in order to perform a task. In the field of computation theory, the task is to transform input data into output. Therefore, an algorithm is analogous to a mathematical function. Unlike a mathematical function, however, there is an inherent notion of the passage of time. An algorithm is fully deterministic. It will always produce the same output for a given input. However, the number of steps that are taken before an output is produced can vary. It can even tend to infinity, in which case no output is ever generated. This is similar to a function being undefined for a certain input. The problem is that an algorithm may require arbitrarily many steps to produce an output. There is no fixed point that allows us to say with certainty that, if an algorithm has not produced an output thus far, it never will. This is known as the halting problem.

A Turing machine is an ideal model of a general-purpose computer that has infinite memory [51]. A problem is computable if it is possible to express it as an algorithm and associated input that combined will produce an output. If a problem is computable, then a Turing machine can compute it. This is a very powerful concept, as it implies that there exists a finite set of logical operations that constitute the building blocks of all computations. If an instruction set or programming language is Turing complete, in that it provides the necessary functionality to simulate a Turing machine, then that instruction set or language can be used to solve all computational problems. Memory limitations aside, all computers are equal in their capacity to *eventually* solve a computable problem.

This brings us to an important point. How long it takes to produce the output is *always* important. Consider the case of decrypting data that has been encrypted using the AES-256 cipher. There are 2^{256} different keys to try. If one were to try each and every one of them, then they are sure to find the correct key, eventually. We can safely say that by the time the brute-force algorithm returns, the person who started it will have long gone. For any problem, there is an upper bound to the amount of time that we can accept to wait before the problem becomes computationally infeasible.

Even within the limits of feasibility, producing output sooner than later is generally preferable. This is true of programs that are run in order to produce the output. For example, a student who is using Latex to produce PDF documents from input text files will appreciate if the output would be produced in one second rather than ten.

Now consider an interactive 3D computer game, which is essentially an interactive simulation. The work includes getting user input, updating the world model, and rendering a frame. Frames are displayed in rapid succession to the user, similar to how frames of still images played back one after the other make a movie. When we consider the frame to be the output of a computation, the rendering operation, then all computers (provided they have sufficient memory) will be able to render the same frame equally well. However, rendering a single frame every hour is infeasible, as the game can no longer be played. If we are able to render frames at a rate of 30 Hz, then we have a game. Moreover, it is also bad for a frame to be displayed too soon, as variable frame rate can result in jittery animation. The refresh rate of the display also has a fixed frequency, and there are benefits to matching this in how often we present new frames.

We see that the problem has now changed. We are no longer only concerned with the result of a computation. We are also concerned with *when* these results are computed. An algorithm specifies the sequence of instructions that will be performed. Now, we want control over when these steps are performed. In other words, we want control over *runtime execution*. We can refer to a program that has such requirements as a *realtime* program.

The logic expressed by the algorithms that make up a program do not depend on the hardware that is used to execute them. The benefit of using high-level programming languages like C is that it enables expressing these algorithms in a platform-independent way. That being said, a program that is unable to interact with the user is of little use. User interaction involves receiving input or providing output, and is done using I/O devices. Displays, keyboards, hard disks and network interfaces are all examples of I/O devices. Interacting with them requires producing what is known as side-effects in programming. It causes something to change in external state.

One of the primary goals of an operating system is to present an abstraction layer to userspace processes for interacting with hardware devices. A process does not interact with I/O devices directly. Rather, it makes system calls. This is a request that the process makes to the kernel. Upon receiving the request, the kernel interacts with the designated hardware device on behalf of the process that made the request. The result is later transferred from the kernel back to the process. It is the kernel's responsibility to know how to interact with each device, being aware of its idiosyncrasies. This complexity is hidden from userspace processes, which only see the abstract system calls API. When developing userspace software, we only have to know about the kernel API and not about specific hardware devices.

As we have said, the core logic of a program is essentially platform-independent. The parts that are platform dependent are developed targeting a specific operating system platform, rather than a specific set of hardware. Furthermore, it is possible to keep the core part of a program intact and only change the platform-dependant parts in order to adapt the program to run under different operating systems. This is known as porting.

For a program that does not have realtime requirements, its interaction with the kernel and underlying hardware may not be particularly important. If a program performs up to specifications on a particular platform, then porting it to a different platform will likely result in an acceptable solution.

In the case of a realtime application, porting becomes more involved. The goal of the program can no longer be reduced to computing a result. While the way that results are computed remains the same, the means in which we ensure that work gets done at the right time depends entirely on platform-specific mechanisms. This requires a more intimate understanding of the kernel: how it works, what its interaction with the process is like, and what its interaction with the hardware is like. We also need to have some idea about the underlying hardware too. Designing a realtime application needs taking all of this information into account. This information also determines how a system should be configured, in order to run a realtime application properly.

Lynxtun is a quintessential realtime application. There are two specific realtime requirements that we have. First, a dispatch operation needs to be performed at the right time. That is, the network interface should start transmitting the Lynxtun datagram as soon as possible, once a scheduled dispatch operation is due. Secondly, a Lynxtun datagram has to be prepared before it can be dispatched. IP packets have to be encapsulated in it, and it has to be encrypted. Other than reading IP packets from the TUN device, the preparation of the encrypted Lynxtun datagram is a computational problem. The goal here is that, the computational output (the encrypted datagram) should be ready some time before the next dispatch operation is due.

4.2 Achieving Deterministic Runtime Execution

In light of the discussion above, we now identify numerous factors that can contribute to non-deterministic runtime behavior, and propose methods of mitigating them. The information presented here has implications on various design decisions necessary for our implementation of Lynxtun for Linux, which is discussed in the following chapter. It also provides insight with regards to how a system should be configured in order to achieve optimal security. However, we should mention that the correct configuration for any particular Lynxtun deployment will depend on both the actual hardware specifications, as well as operational and security requirements. The discussion below is based on Linux 4.16.

4.2.1 CPU Related Factors

Each machine instruction requires a certain amount of processor cycles to perform. How long it takes to execute a given instruction will depend on the CPU frequency, design and architecture. If instructions can be executed in parallel, then the number of cores (physical or logical) is also important. In general, modern CPUs are highly complex. They cannot be assumed to have a fixed clock rate. Technologies like Intel SpeedStep and Turbo Boost are used to dynamically adjust the CPU frequency. The frequency is dialed back when there is no load in order to conserve power. This is especially important for battery powered devices like laptops. A CPU core can be throttled if its temperature exceeds a given threshold. The CPU can be overclocked for brief periods of time when there is high load, as long as the power, current and temperature values are within acceptable ranges. If there are multiple cores available, and a single highly intensive thread of execution, this can switch from one core to another. The active core will be overclocked. When it becomes too hot, execution will continue running on another core working at full speed while the previous core is allowed to cool. Dynamic adjustment of the CPU clock rate is referred to as *CPU frequency scaling*.

A CPU can only continue execution if all computational resources necessary for an instruction to be executed are available. Otherwise the CPU will stall. The time it takes to access memory is significantly greater than the duration of a CPU cycle. Waiting for disk access is much worse. CPUs use several layers of cache that take advantage of the concepts of spacial and temporal locality, to store data that is likely to be needed in the short term by the CPU in a location that can be accessed much more quickly. Cache management is inherently a difficult problem. Furthermore, the content of the cache is affected by all processes that are executing on a given CPU core. The fact that the core on which a particular task is executed can change makes the issue even more complicated.

To add to this complexity, CPUs perform out-of-order execution. Suppose that the CPU stalls because resources needed for the next instruction are not available. It might be the case that another instruction further down the line can be executed straight away. In this case, the CPU will execute this instruction *out-of-order* instead of stalling. Modern CPUs go one step further and perform out-of-order in a *speculative* fashion. That is, the processor is allowed to execute instructions that have not been actually issued yet, but are *statistically likely* to be issued in the near future, based on the current state and previously observed behavior. If the speculative instruction does in fact realized, then the result has already been computed. Otherwise, the CPU has to roll back the speculative instruction. The mechanisms that are responsible for this are immensely complex. Speculative execution is the cause of a recently discovered class of security vulnerabilities, including Spectre and Meltdown. The complexities of speculative out-of-order execution and its implications on security are still being understood, and constitute an important field of active research.

The fundamental issue here is that general-purpose CPUs are designed to maximize utilization and achieve high levels of throughput and responsiveness, while consuming less power and generating less heat. Deterministic runtime behavior in the sense that it is possible to predict the exact time that a specific instruction will get executed, or how long it will take to carry out a given sequence of instructions, is not a priority.

The Linux kernel provides a mechanism for controlling CPU frequency scaling. This is done through the use of a CPUFreq governor. Setting the `CPU_FREQ_DEFAULT_GOV_PERFORMANCE` kernel configuration option enables statically setting the frequency to the highest value supported by the CPU. However, this is still subject to throttling that can occur if the CPU exceeds a temperature threshold, in order to prevent hardware damage. Alternatively, a *userspace* CPUFreq governor enables setting the frequency manually while the system is being used. This offers greater flexibility, as the user can alter the configuration depending on current needs. However, this is not available on all systems. For example, Intel Core processors use the `X86_INTEL_PSTATE` driver, which does not support the userspace CPUfreq governor. It only supports the performance and powersave governors.

The duration of a single CPU cycle represents the smallest unit of time in the system.

Modern CPUs generally run at GHz frequencies. 1 GHz corresponds to a clock cycle that lasts one nanosecond. We define our tolerance for deviations in dispatch times in absolute terms. We say that, the deviations should be on the order of several microseconds at most. It is unlikely to be able to achieve this requirement if we were using a CPU running at a frequency of 1 MHz, as a single CPU instruction would introduce a delay above our fixed threshold. In general, the higher the CPU frequency is, the more likely we are to be able to minimize the deviations.

CPU frequency scaling issues have no bearing on how to design the Lynxtun implementation. However, it can influence runtime behavior. In principle, we can assume that fixing the frequency at the highest possible value that is sustainable even during period of heavy load is the optimal strategy in terms of a deterministic dispatch process.

4.2.2 Memory Related Factors

Modern operating systems, including Linux, use virtual memory management. Virtual memory management requires support from both the kernel and the hardware. The memory space allocated to a userspace process physically exists across several layers of CPU cache, main memory and swap space. Memory access times vary greatly depending on where the requested data is actually located at the time of the request. The number, sizes and access times of CPU cache layers depend on the processor. The access time from main memory depends on the type and frequency of the memory modules installed. For a given type of memory module, access times tend to increase with memory size. Memory access time for pages that were swapped out onto disk are significantly longer, though SSD disks offer significantly higher performance in comparison to magnetic hard disks. Memory access time is also greatly influenced by overall system load, which is highly non-deterministic. Different processes contend for CPU cache and main memory. Memory access requires information being sent over a bus, and will take longer if contention for bus utilization is high.

When implementing Lynxtun for Linux, there are several ways in which we can mitigate these issues to a certain extent.

The first is by using *memory locking*, which is done using the `mlock()` system call. This allows a specific range of memory addresses to be *locked* into main memory, such that they are not allowed to be swapped out to disk until either it is unlocked explicitly through the `munlock()` system call, or the process terminates. This prevents significant delays that can arise from having to retrieve data from disk. Our implementation should take advantage of this capability. However, there is an implication. Locking memory will significantly increase the memory pressure on the system, as it effectively decreases the available space on main memory for everything else. We need to be conservative in our use of memory. The general design principle should be to allocate all memory buffers that will be required throughout the lifetime of the process at startup, keep the sizes of these to a minimum and lock all of them into main memory. The allocation can either be done statically or dynamically. However, it is very important that dynamic memory allocation (`malloc()`) should not be done after the start of the first round. Dynamic memory allocation is inherently unpredictable in how long it takes to allocate the requested amount of memory.

The second approach, that can be used to complement the first, is to make efficient use of the CPU cache. By optimizing the structure of our code and particularly memory access operations, it is possible to decrease the likelihood of cache misses. Of course, writing perfectly optimized code would require us to know how large the cache actually is, but this depends on the CPU. In general, performing less memory access operations, keeping the size of the accessed values small, and accessing memory that is stored adjacently is beneficial. While the CPU cache is subject to external influences, it is also possible to mitigate this through the use of CPU isolation, as discussed below.

As was the case in the previous section, using hardware that is more performant is beneficial.

4.2.3 Scheduling Related Factors

Earlier in this chapter, we mentioned that one of the objectives of a kernel is to present an abstract API to interact with the hardware to userspace processes. In the case of a *timesharing* operating system, there is an additional objective: being able to present the same interface to multiple processes running at the same time. Unix, Linux, Windows and OS X are all examples of timesharing operating systems.

We have to make a clarification. A single CPU can only execute a single instruction at a time. Therefore, it can only be working on a single task at any given moment. On a uniprocessor system, two processes can never be said to be running at the exact same time. Once again we invoke the fact that computer design should not be considered independently from how a user — that is, a human — interacts with it. There are limits to the human perception of time. By partitioning available CPU time into short *timeslices* (on the order of milliseconds or less), and allocating each timeslice to a different process creates the illusion, from the perspective of the user, that the processes are running at the same time despite the fact that the CPU executes a single instruction belonging to a single process at a time. This is the foundation of the interactive computer experience that we are accustomed to. It is then the operating system's job to make it appear to the process that, while it is running, it has full control over the hardware; as if it were the only process that is running.

However, there is an important limitation to this, that is related to our earlier discussion on program logic and runtime execution. The process is isolated in the sense that which instructions are executed are not affected by other processes (unless such a relationship is implemented explicitly through inter-process communication mechanisms). However, the runtime execution certainly is. Therefore, there are ramifications due to timesharing in the context of a realtime application.

The key component of a timesharing kernel is the *scheduler*. The scheduler manages the allocation of CPU timeslices among processes. A process is said to be running while its instructions are being executed on a CPU. Otherwise, the process is said to be sleeping. In order for a sleeping process to start running again, the scheduler has to select it.

The transition between running and sleeping is transcendental to the logic of the userspace program. The internal state of the process is not changed. There is no lapse in the process's *stream of consciousness*, so to speak. As far as the process is concerned, it never stopped running. In order to do this, the entire internal state of a process should be captured as it is being put to sleep, and this state should be restored before the process starts running again. A context switch is the process through which execution switches from one thread of execution to another. Performing a context switch requires saving the state of the former, and restoring the previously saved state of the latter. This incurs time and computational overhead. It may even be the case that context switches happen so frequently that all processor time is spent on saving and restoring state, and no time is left to perform any actual work. This is called *thrashing*, and it is the scheduler's responsibility to avoid it.

While the concept of sleeping is not related to the computational logic of a program, as we have noted, not all parts of an actual software program are purely computational, and there are times when it is necessary to produce external side-effects. This might be done in order to interact with an external hardware device through the kernel, or it might be done to interact with the kernel itself. In either case, this requires a system call being made from the process to the kernel. When a system call is made, then execution switches from the process code to kernel code, so that the kernel can handle the request. This puts the process to sleep. There are two reasons why this is necessary. First, the kernel itself is not exempt from the fact that only a single task can be running on a CPU at any given time. If there is a single CPU on the system, control necessarily has to switch to the kernel so that it can handle the request, thereby putting the process to sleep. Even if there are multiple CPUs available, the process cannot continue its execution before the kernel at least acknowledges that the request has been received. A system call is similar to a function call in this sense. In fact, the `glibc` library implements library wrappers for most system calls. Execution is blocked on the calling thread until this function returns.

If the system call was hardware-related, then the kernel has to send instructions to the device. I/O operations in particular take a long time to perform. For this reason, the Linux kernel uses asynchronous I/O. Instructions that are sent to the device are executed on a hardware controller on the actual device. The kernel does not wait for the operation to finish, and instead moves onto working on other things. This can include performing a context switch to another application. At some point in the future, when the result is ready, the device will generate a hardware interrupt request (IRQ) in order to notifying the kernel. The interrupt is received on a CPU. Upon receiving an interrupt, the CPU stops any work that it was doing, and starts executing the kernel's interrupt handler code that exists at a fixed memory address. At this point, the result to be returned to the original userspace process is available within the kernel. The kernel will flag the process as *runnable*, so that it becomes eligible to start running as the result of a scheduling decision. However, this might not happen immediately. It is possible that the scheduler selects a different process to run in the next available timeslice. When the original process wakes up, the result of the system call is transferred into the memory space of the process, and the system call returns. Note that there are delays associated with each step in this process that depend on the scheduler load, context switches, and latency due to hardware devices. Therefore, it is difficult to predict how long it will take for a system call to return.

The implication of this on implementing Lynxtun is that we have to be careful about when and where we make system calls. In particular, system calls that involve I/O operations. Whenever we make a system call, we do not know when control will be returned to us. Then again, there are I/O system calls that we nevertheless have to make. At the very least, we have to read and write IP packets to the TUN device, and read and write Lynxtun datagrams to the UDP/IP socket.

While making a system call voluntarily yields control, it is also possible for a context switch to take place without being triggered by the process. We have already touched upon the way this happens, when we mentioned that a CPU that receives a hardware interrupt will stop working on what it was doing. This could very well be a userspace process, putting it to sleep.

Linux is a *preemptive* kernel. This means that a running process can be suspended even if it does not yield control voluntarily. Such a process is said to have been preempted. This is a central requirement for a timesharing operating system. A userspace process is not allowed to run indefinitely even if it does not make any system calls. As we have mentioned, CPU time is partitioned into timeslices. Each timeslice has a fixed duration, and control will return to the kernel at the end of each timeslice so that a new scheduling decision can be made. This can be followed by performing a context switch to a different runnable process. In general, every time control switches to the kernel, the scheduler has to decide what to run next. Even in the absence of IRQs that result from various system calls and device activity, the a timer interrupt will be generated at regular intervals called *ticks* to ensure that scheduling decisions are made regularly. The kernel can also explicitly preempt a process if there is another runnable process with a higher priority.

Returning to our runtime requirements for Lynxtun, there are two challenges related to scheduling. First, if Lynxtun is not given sufficient processor time necessary to complete the work required to prepare an encrypted datagram by the time that the dispatch operation is due, then the dispatch deadline will be missed. Secondly, if Lynxtun is not running at the exact time that the dispatch operation should be made, it can only do so once it regains control. Again, this leads to the dispatch deadline being missed.

The most straightforward approach to reducing variability due to scheduling is to limit the number of other processes that are running on the same system. However, there is an important point to consider. The reason we want deterministic runtime execution is in order to ensure that dispatch time variability is not correlated with the underlying communication. *Random* variability due to unrelated processes is not necessarily a problem in and of itself in terms of traffic flow confidentiality. If the IP packets that are encapsulated by the tunnel are generated by another userspace process running on the same host, and this is the only other process running besides Lynxtun, then variability due to scheduling becomes more dangerous. While running unrelated processes make it more difficult to achieve deterministic runtime execution, it also means that failing to achieve this is less dangerous, as the resulting variability will also include noise.

4.2.3.1 Realtime Scheduling

The Linux kernel allows the scheduling policy of a process in runtime. This presents us with an important tool in order to mitigate issues related to scheduling.

The first concept to be aware of is that each process can be assigned a *scheduling priority*. Processes can be preempted in favor of a runnable process with a higher priority. Similarly, a routine scheduling decision will favor processes with higher priority.

The default scheduler policy of Linux is `SCHED_OTHER`, which is also *Completely Fair Scheduler (CFS)*. CFS operates in what is essentially a round-robin fashion by aiming to give each process an equal amount of processor time. A process that is subject to this policy will have a *nice value*, which is 0 by default. The negative of the nice value is the process' scheduling priority. Therefore, increasing the nice value *decreases* the priority of the process.

Linux also supports realtime scheduling policies specified by POSIX realtime extensions [6]. These are `SCHED_FIFO` and `SCHED_RR`. Within the set of all runnable tasks at a given priority, the former will select the processes in the order that they became runnable, and the latter will use a round-robin approach.

The primary system call used to configure the scheduler is `sched_setscheduler()`. We can use this to configure Lynxtun to run with a high realtime priority in order to improve deterministic runtime behavior.

4.2.3.2 Kernel Preemption

While preemption is an integral part of the design of Linux, there is an important caveat. A low priority process that is working in user mode can always be preempted by the kernel to pass control to a higher priority runnable process. However, the situation is different when the low priority process made a system call and the kernel is in control and doing work on behalf of the low priority process. The instructions are being executed here belong to the kernel. Whether or not kernel code itself can be preempted is a separate matter. If it is, then we call this a *preemptible* kernel. The degree at which this is possible depends on how the kernel was compiled. If the kernel is not preemptible, then a high priority task might be delayed while waiting for the kernel to finish working, even if this work is being done on behalf of a low priority task.

According to the kernel configuration documentation, `CONFIG_PREEMPT_NONE` is the traditional Linux preemption model. Here, kernel tasks cannot be preempted. It increases throughput and is recommended for servers and scientific-computation systems.

`CONFIG_PREEMPT_VOLUNTARY` and `CONFIG_PREEMPT__LL`, known as *Voluntary Kernel Preemption* and *Preemptible Kernel* configurations, are geared more towards desktop use. The introduces explicit preemption points into the kernel. The latter makes allows kernel code that is not in critical sections to be preempted. This reduces throughput, but allows interactive applications that require latencies in the milliseconds range to be more responsive when the system is under load. Applications for which this is particularly important are media applications like listening to music. This makes these suitable choices for desktop machines.

Next, we discuss the PREEMPT_RT kernel patch, which is maintained by the *Real Time Linux* collaborative project [7]. This is not a part of the mainline Linux kernel. Rather, it is a kernel patch that has to be applied separately. It converts Linux into a *fully preemptible kernel* and allows it to function as a realtime operating system (RTOS) capable of attaining latencies on the order of microseconds. Having said this, the use of RT_PREEMPT is not sufficient to make this happen. The RT Linux Wiki contains useful information concerning realtime application design and system configuration issues and mistakes to avoid.

The RT Linux wiki provides an interesting example that illustrates how tricky this subject can be. A user reported that they have observed latencies greater than 500 microseconds when the computer is booted with a USB flash stick plugged in. However, if the same USB flash stick is connected *after* system boot, then latencies become significantly lower.

The following passage from the RT Linux Wiki [7] neatly sums up the level of difficulty in achieving determinism.

> "An RT-application is only able to operate correctly if the underlying OS and hardware are able to provide the needed determinism. That means a higher priority task can preempt a lower priority task. If for example a BIOS decides to use all CPU cycles for a very long time, no operating system or application can provide any latency guarantees. The hole system needs to be tuned and configured correctly."

Having a preemptible kernel is desirable for Lynxtun. The kernel should either be configured with PREEMPT__LL or preferably with PREEMPT_RT. There are two issues to consider with the latter. First, it means that a patched kernel should be used, which is not very convenient for the user. Second, a fully preemptible kernel is not suitable unless a dedicated machine is being used for Lynxtun. In Section 4.3, we compare the realtime behavior of the two alternatives. Additionally, in Chapter 6, we compare results obtained from running Lynxtun on each configuration.

4.2.3.3 Reducing Scheduling Clock Ticks

We have mentioned that timer interrupts that are generated at regular intervals ensure that scheduling decisions are made even in the absence of other IRQs. The Linux kernel source tree includes a documentation file on this subject[1].

IBM-compatible PCs have a hardware time-measuring device called the Programmable Interval Timer (PIT) that can be used to generate a timer interrupt on IRQ line 0 at

[1] /Documentation/timers/NO_HZ.txt under the Linux kernel source tree.

a fixed frequency that is set by the kernel [14]. Each timer interrupt is called a *tick*. These play an important role in scheduling, which is why they are referred to as scheduling ticks. Even if a running process does not make a system call and no other IRQs are generated, the timer interrupt will cause control to switch to the kernel and lead to a scheduling decision being made. Different frequencies for the clock range from about 100 Hz to about 1000 Hz.

While the timer interrupts are important from the point of view of timesharing if there are multiple runnable processes waiting on a single CPU. However, as stated in the Linux documentation (NO_HZ.txt), "if a CPU is idle, there is little point in sending it a scheduling-clock interrupt (because) the primary purpose of a scheduling-clock interrupt is to force a busy CPU to shift its attention among multiple duties, and an idle CPU has no duties to shift its attention among." This becomes an important concern when considering battery-powered devices, because handling IRQs increases power consumption. We see that these concerns were important in the decision to make a shift towards a *dynamic ticking system (dynticks)*, where scheduling clock ticks can be omitted depending on the situation, such as when the CPU is idle.

The `CONFIG_HZ_PERIODIC` kernel configuration option can be used to have the old behavior, where timer interrupts are generated at a constant rate. This rate can be 100 Hz, 250 Hz, 300 Hz or 1000 Hz. Linux kernel configuration documentation suggests that 1000 Hz is typical for desktops and 100 Hz for servers.

The configuration documentation suggests that the *idle dynticks system (tickless idle)* is the usually desired behavior, with energy saving being cited as the reason.

The most interesting alternative for us is `CONFIG_NO_HZ_FULL`. This option causes scheduling timer interrupts to be omitted when a CPU has only a single runnable task. NO_HZ.txt identifies timer interrupts are a source of timing jitter, and presents `CONFIG_NO_HZ_FULL` as a useful means of reducing this for real-time applications, particularly those that have short and regular iteration times. This is the case for Lynxtun.

Even if the kernel is compiled with `CONFIG_NO_HZ_FULL=y`, it is not activated by default and has to be explicitly enabled for each CPU using the boot parameter `nohz_full=`. A recent addition is the `CONFIG_NO_HZ_FULL_ALL` configuration option that enables the full dynticks system for all CPUs apart from CPU 0.

We evaluate the various possible alternatives in section 4.3.

4.2.4 Timing Events and the vDSO

Our realtime requirements dictate that we have control over when particular actions are taken. Specifically, we need to control the time that dispatch operations are executed. There are two basic approaches that we can take in order to do this. Either we delegate the task to the kernel and request it to notify us at a given time, or we repeatedly check the current time until we find that a set deadline has been reached.

In both cases, the precision with which we can time events depends on the resolution of the timekeeping system available on the system. As stated by the Linux documentation[2], before Linux 2.6.21, the accuracy of the timekeeping device was limited to a *jiffy*. The duration of a *jiffy* is precisely the period of the timer interrupt tick. This meant that timekeeping could only be done at a resolution of milliseconds. In such a case, it is not possible to limit dispatch time variability to microseconds.

To solve this problem, the Linux kernel features the high-resolution timers API, that allows timers up to nanosecond resolution. The Linux kernel source tree includes documentation on the subject[3]. There is also a comprehensive LWN article on the subject [2]. We should point out that high-resolution timer availability depends on the hardware to support it.

Userspace applications can make use of the high-resolution timers API through several means. `nanosleep()` system call puts the process to sleep, to be waken up after the specified time. The itimers interface instructs the kernel to send a signal to the process at specified intervals. The `clock_gettime()` system call can be used to obtain the current time with high precision.

The problem with `nanosleep()` is that it puts the process to sleep. As we observe in Section 4.3, the wake-up times are subject to significant variability. Our assessment is that using itimers to generate signals at specific times is not the correct approach for Lynxtun. The procedure for processing incoming data and outgoing data can each be executed in a completely synchronous way, and the two can be processed independently. Having to handle asynchronous signals would introduce complexity that makes it more difficult to reason about the realtime behavior of the program. Furthermore, it is challenging to implement signal handling in the context of a multithreaded program. Being able to handle each of the two aforementioned tasks on separate threads has certain advantages.

In Section 4.3, we show that it is viable to implement a busy-loop that constantly checks the current time until a set deadline has been reached. However, for this to be viable, it should be possible to check the current time using the vDSO.

vDSO stands for the *Virtual Dynamic Shared Object* [1]. It was found that some userspace code makes certain system calls provided by the kernel so often that it leads to the resulting context-switching overhead to dominate the system [4]. In particular, these are calls related to getting the current time, such as `gettimeofday()` and `clock_gettime`. While these have to do with device interaction, unlike most other system calls, there is no obvious reason why these calls should not be allowed to be made directly from userspace. This led to the addition of the vDSO support to the Linux kernel.

The vDSO is a shared library that gets automatically mapped into the address space of all userspace applications. The `glibc` library takes advantage of the vDSO when available by calling a virtual system call exported into the vDSO rather than making a full system call that would result in a context switch. The virtual system call is executed directly in userspace, greatly reducing latency and context-switching overhead.

While being able to get the current time using the vDSO can be assumed to be the

[2] See man (7) time

[3] Under the /Documentation/timers directory of the Linux kernel source tree

[4] man (7) vdso

case on almost all physical systems, there are certain cases where getting the current time through vDSO is not possible. As it turns out, the implementation of certain hypervisors does not support doing this. Results published online [9] indicate a 77% performance decrease in performance due to code running on AWS EC2 because the virtualized clock source on the xen hypervisor does not support vDSO.

While Lynxtun will run regardless of vDSO support, its deterministic runtime behavior will be *severely* degraded if it is not available. This is because each `clock_gettime()` call made will result in a full system call being made, and this will be done shortly before a dispatch operation is made.

4.2.5 Multiprocessing and CPU Isolation

A Linux kernel compiled with `CONFIG_SMP=y` will be able to make use of multiple CPU cores. A kernel compiled with `CONFIG_SMP=n` will run faster on a uniprocessor machine, but will only be able to use a single core on a multi-processor machine.

Linux uses the Symmetric Multiprocessing Model (SMP) to handle multiple CPUs. What this means is that the kernel does not have a bias toward one CPU with respect to the others [14]. Linux configures the Advanced Programmable Interrupt Controller (APIC) to dynamically distribute IRQ signals among available CPUs, following what is essentially a round-robin approach.

On the other hand, Linux is not *purely* SMP, as it offers ways of treating cores separately. We have already seen one example of how this is possible, in the case of omitting scheduling clock ticks for certain CPUs.

The `CONFIG_CPU_ISOLATION=y` kernel configuration option is another such mechanism. When this is available, the `isolcpus=` kernel boot parameter can be used to isolate a number of CPUs so that no process will get added to the run queue of this processor by default, and no IRQs will be routed to this CPU.

The kernel configuration documentation states that this allows to "make sure that CPUs running critical tasks are not disturbed by any source of 'noise' such as unbound workqueues, timers, kthreads, etc." This is made possible by a feature known as the IRQ affinity of multi-APIC systems [14]. This feature allows IRQs to be routed to a specific CPU.

The `sched_setaffinity()` system all or the taskset utility can be used to set the *CPU affinity* of a process, which designates the set of CPUs that the scheduler can assign this task to run on. While the *isolcpus=* boot option is the preferable way of isolating a CPU, it is also possible to use taskset on all exiting processes to isolate a CPU while the system is already running.

The Linux kernel also provides the `cpuset` pseudo-filesystem [5] if compiled with `CONFIG_CPUSETS=y` that can be used to manage which processes get assigned to which CPUs.

[5] See man (7) cpuset

A related issue is Simultaneous Multithreading (SMT), which is also known as *hyperthreading*. It is related to the `CONFIG_SCHED_SMT` kernel configuration option. Its most notable use is in the Intel Pentium 4 CPU. SMT presents a single physical CPU core as two logical CPU cores to the operating system. Within the CPU, parts of it that store architectural state are duplicated so the scheduler can interact with two logical CPUs and schedule processes to them independently. However, this duplication does not extend to the main execution components, meaning that only one of the threads can be executed at any given time. This is an alternative approach to the problem solved by out-of-order execution, in that the CPU can quickly switch to working on a different task instead of stalling (e.g. due to a cache miss).

4.2.6 Hardware Devices

Hardware devices generate hardware interrupts that can be detrimental to deterministic runtime behavior. Therefore, limiting the number of connected devices is beneficial. That being said, similarly to what we have said about non-determinism due to unrelated userspace processes, if we have reason to believe that non-determinism introduced by a particular hardware device cannot be related to the underlying communication, then this might actually be beneficial to the overall security of Lynxtun.

We should also mention that there will be some variability due to the network stack on the host. The actual dispatch time is when the network packets appear on the network, as they are physically transmitted by the egress network adapter. However, we do not have precise control over when this happens. Within the context of a userspace implementation, the most that we can aim for is to make the system call that writes an encrypted Lynxtun datagram to the UDP/IP socket close to the scheduled dispatch time. There will be some variable amount of delay between this and when the first packet is actually sent. Here, we refer to the burden of knowledge principle. From the point of view of the network stack, encrypted Lynxtun datagrams should all appear to be equivalent, regardless of the data contained within them. Therefore, it is likely that the variability generated here will mostly be noise. If, however, the underlying communication is also arriving through the network stack and the same network adapter in particular, then it is possible for this variability to be correlated to underlying communication. There is not a lot we can do to prevent this. If the underlying communication arrives through a network, ensuring that it does so through a separate network adapter should help mitigate this issue.

4.3 Experimental Comparison of Alternative Configurations

4.3.1 Experimental Setup

We implemented the simple C program presented in Appendix F.

The waiting behavior is implemented in two separate ways.

1. Sleeping using `clock_nanosleep`.

2. Spinning in a busy-loop by checking the current time using `clock_gettime()`.

We note that we perform these experiments on a physical host that supports `clock_gettime()` through the vDSO. This is essential to this experiment.

All our experiments are performed using a dedicated physical host. The hardware specifications for this host are given in Appendix B.

The specifications for the three kernels configurations that we use in our experiments are given in Appendix C.

We can boot each kernel using `isolcpus=1` to isolate CPU 1. Alternatively, we do not isolate any CPU.

We use `sched_setaffinity()` to be able to constrict the process to run on a specific CPU. Alternatively, we set no CPU affinity.

We use `sched_setscheduler()` to make the process be subject to the realtime `SCHED_FIFO` scheduler with a high priority of 98, or alternatively leave it at the default which is `SCHED_OTHER` with the default nice value of 0.

We use the hackbench benchmarking utility to test the system when the scheduler is under heavy load. hackbench forks many processes which each use a large number of pipes between them to send messages to one another, in order to put artificial load on the system scheduler. We perform our experiments when the system is idle, and when it is under load.

We collect 250 samples. The target waiting duration is 20 milliseconds.

4.3.2 Results

4.3.2.1 Baseline

Parameters: lynx-2-desktop, no CPU isolation, default scheduler.

This represents a system without any special configuration.

We observe the following values when we compare sleeping vs. spinning when the system is idle and under load.

	Sleep (Idle)	Spin (Idle)	Sleep (Busy)	Spin (Busy)
Mean (nanosec)	292,406	404	4,819,182	146,862,816
Std Dev.	9,477	91	24,841,475	178,450,136

When we use spinning when the system is idle, then the average deviation is on the order of nanoseconds. This is the order of CPU cycles. On the other hand, when we sleep instead, the average deviation rises to around 300 microseconds. This is a very significant increase. While both values increase significantly when the scheduler is put under load, we see that the increase is much greater in the case of spinning. The average deviation is on the order of several milliseconds in the case of sleeping, but hundreds of milliseconds when spinning when the system is busy. This is an interesting observation. We see that by setting a wake-up timer in the kernel using `clock_nanosleep()` offers better performance that a program that constantly gets preempted as it is trying to check the current time using `clock_gettime()` in a busy-loop.

Noting that our Lynxtun implementation uses spinning while waiting for the dispatch process, we observe that this is much better than sleeping as long as the system is idle, performs particularly bad when the system is under load. This is unacceptable for Lynxtun.

4.3.2.2 Strictly Configured System

Parameters: lynx-0-nohz-rt, CPU isolation, realtime scheduler

We perform the same experiment with a strictly configured system. The system uses the `RT_PREEMPT` kernel patch, CPU isolation is enabled, and realtime scheduling is used.

	Sleep (Idle)	Spin (Idle)	Sleep (Busy)	Spin (Busy)
Mean (nanosec)	302,354	423	232,575	368
Std Dev.	31,348	98	24,554	86

We see that the situation under load has greatly improved. Most importantly, spinning yields values in the nanoseconds range regardless of whether the system is under load or not.

The second observation that is important to make at this point is that sleeping remains in the range of hundreds of thousands of microseconds. What this tells us is that the correct way to wait for the dispatch operation is in fact spinning, which is what we have done in our implementation. However, we see that this is particularly susceptible to poor performance on an unconfigured system. This shows how important configuring the system actually is.

4.3.2.3 Usability of a Stock Kernel

The next question to answer is whether a stock kernel that has not been patched and compiled specifically for Lynxtun can be used at all by relying on boot parameters, CPU affinity setting and using a realtime scheduler.

In Figure 4.1 we see the comparison of spinning data obtained from lynx-0-nohz-rt (Realtime) and lynx-2-desktop (Stock).

We see that the behavior is quite similar in all cases. There are some slight patterns in the behavior of the stock kernel when it is idle. Also, we can see that there is a slight

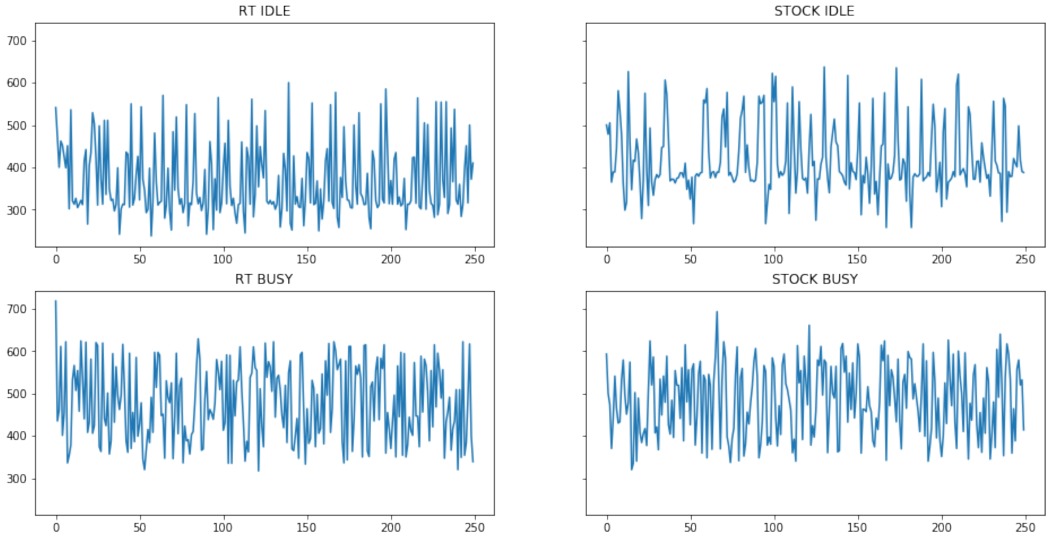


Figure 4.1: Realtime vs. Stock Kernel Spinning

difference in the means. In any case, all variations remain in the nanosecond range. This suggests that a stock kernel can be capable of providing reasonable security for Lynxtun.

We compare the two configurations's sleeping behavior in Figure 4.2.

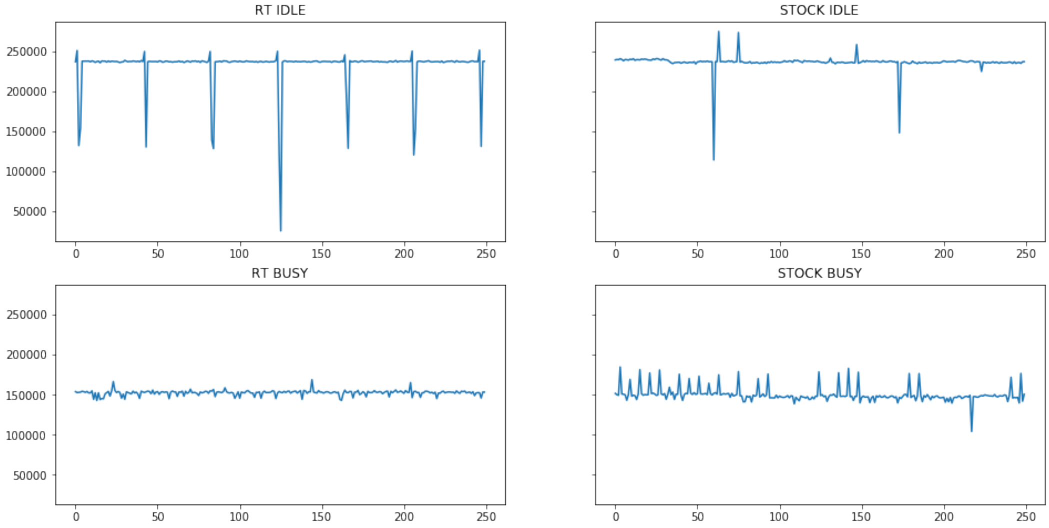


Figure 4.2: Realtime vs. Stock Kernel Sleeping

We make the following observations:

- The idle behavior has similar means in both cases. However, the stock kernel is susceptible to suprious spikes.
- The idle behavior of the RT kernel is very regular, but there are large troughs

that occur at every second. This was interesting. We found that the reason is probably related to NO_HZ_FULL. As it turns out, when timer interrupts are omitted for a CPU with a single runnable task (which is what happens in this case), an interrupt is nevertheless generated at a frequency of 1 Hz for synchronization. This corresponds to the troughs that we observe.

- When we look at the busy behavior, we see that the real-time kernel has less variability, but the means are similar.

- In both cases, the average deviation is significantly lower when the scheduler is busy. This result is rather counter-intuitive. In our first test, we saw that putting the scheduler under load resulted in significantly higher average deviations.

- In both cases, the deviations have larger spikes when idle. But if we remove these as outliers, then the remaining data exhibits less variability.

4.3.2.4 The Effect of Scheduler Load

As we have noted, it is interesting to see that the average deviation when sleeping becomes smaller when the scheduler is put under load, provided the process is running using SCHED_FIFO with high priority. We suspect that the reason is mainly related to the following. When a process makes a system call for an I/O operation, this will result in an I/O IRQ interrupt being generated at some point in the future. When there are a lot of these happening, these interrupts are generated at a greater frequency. Each time this happens, it triggers the scheduler. When the scheduler sees that there is a runnable high-priority process (that was sleeping), then it runs it. It appears that the frequent interrupts are causing the kernel to check whether our high-priority task is ready for wake-up earlier than it otherwise would. Basically, putting the system under load forces it to make scheduling decisions at a higher rate, and this leads it to discovering that a high-priority task is ready for wake-up sooner than it otherwise would. While we find this explanation to be reasonable, actually determining that it is correct would require careful analysis of kernel code. As we only observe this discrepancy in the sleeping behavior and not the spinning behavior, it is not crucial for our purposes. However, this result also goes towards supporting that the correct approach to wait for a dispatch process is spinning and not sleeping, and also the fact that vDSO support is important.

4.3.2.5 Realtime Scheduler vs. CPU Isolation

Using a fully preemptible kernel, using the SCHED_FIFO scheduler and setting a high priority are all related to scheduling. Scheduling, in the context of a timesharing operating system such as Linux, deals with running multiple tasks on a single CPU simultaneously. To be precise, the CPU can only ever execute a single instruction at any given time, but by interleaving instructions belonging to separate tasks, a time-sharing operating system creates the illusion that they are running at the same time. The scheduler is what decides which process gets run, and for how long.

Then, there is the case of CPU isolation. When we dedicate a CPU to a single task, then there is no inherent need to have a scheduler at all: there is nothing to be scheduled.

If we were able to *completely* dedicate a CPU to a process, then it would appear that CPU isolation and countermeasures targeting the scheduler are redundant. It is therefore a valid question to ask, is this actually the case? If we use CPU isolation, can we assume that scheduler related parameters will have no effect?

As we have said, the Linux kernel follows the Symmetrical Multi-Processing (SMP) approach. This is based on the assumption that the kernel treats all CPUs in the same way. However, as we have also said, Linux does offer some capabilities that work around this underlying design decision that allows some control for managing CPUs individually, such as the isolcpu boot parameter and the per-CPU configuration that allows omitting timer interrupts. Furthermore, the Linux kernel takes different execution paths depending on its chosen, preemption model which still influences real-time behavior even in the context of an isolated CPU.

We compare the sleeping behavior of kernels lynx-0-nohz-rt and lynx-2-desktop in three cases:

1. Only real-time scheduling is used with `SCHED_FIFO` 2. Only CPU isolation is used 3. Both are used

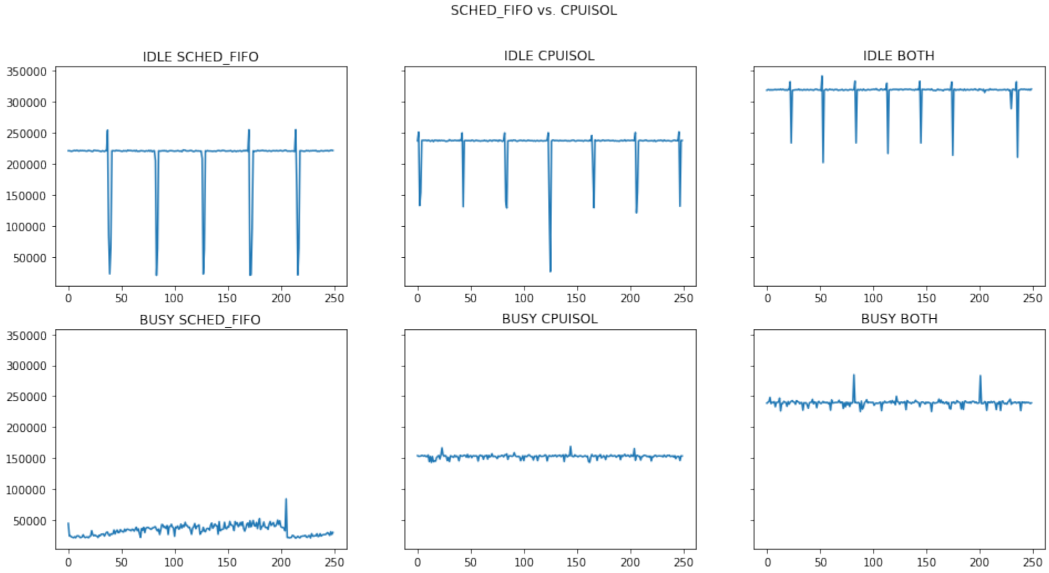


Figure 4.3: Realtime Kernel Sleeping Comparison CPU Isolation vs. Realtime Scheduling

The results shown in Figures 4.3 and 4.4 suggest the following:

- The average deviation is less when only real-time scheduling is used.
- The variability in deviations is less when CPU isolation is used, and its behavior

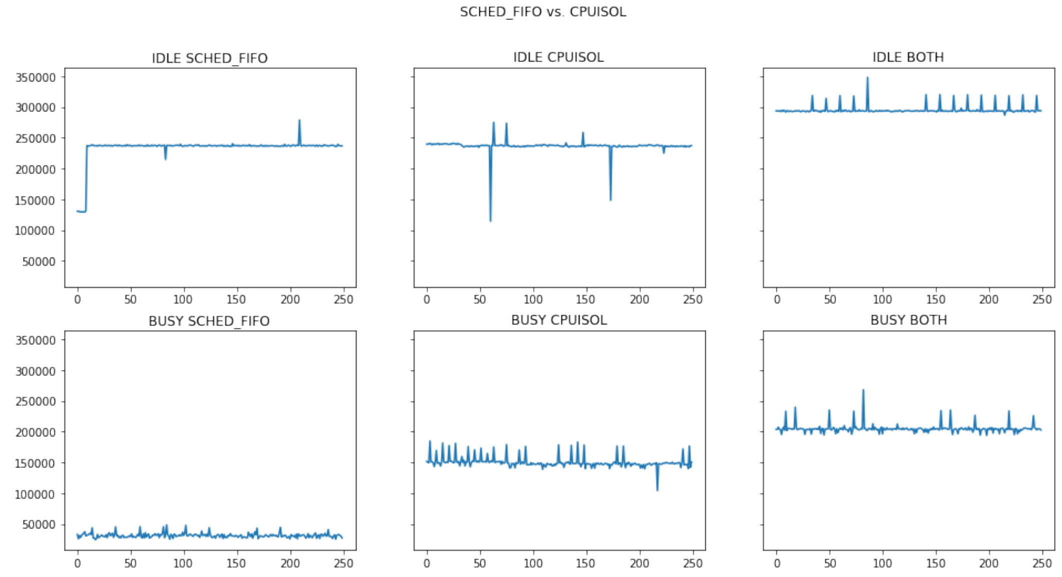


Figure 4.4: Desktop Kernel Sleeping Comparison CPU Isolation vs. Realtime Scheduling

is more stationary.

- Using real-time scheduling when CPU isolation is being used increases the average.

As expected, real-time scheduling does not improve the results if CPU isolation is used, since there is only a single runnable task. To the contrary, it increases the average deviation. While relying on real-time scheduling alone reduces the average deviation, the deviations become less consistent. These results suggest that CPU isolation is preferable, provided that there is a CPU available to dedicate. However, if this is not possible, then real-time scheduling is important to reduce interference due to other processes contending for the same CPU.

4.3.2.6 Periodic Timer Interrupts

As a final point, compare the sleep behavior of kernels lynx-0-nohz-rt and lynx-1-tick in order to observe the influence of having timer interrupts. We can see the fluctuations caused by the interrupt timer. However, the average deviations become smaller. The reason is likely to be similar to our earlier discussion on the influence of I/O IRQs. There is a distinct pattern caused by the timer interrupt, which exhibits significant periodicity. We see that the range of the deviations do not surpass those from lynx-0-nohz-rt. While the variability of the deviations is more, we can safely assume that the clock rate does not depend on the communication, and therefore can be considered to be noise. As such, this variability might actually be desirable. However, care should be taken when setting the dispatch interval to accommodate for these fluctuations.

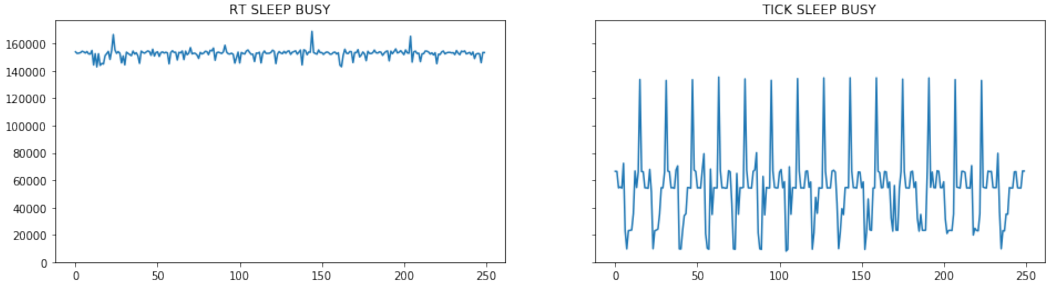


Figure 4.5: Full Dyntick vs. Periodic Tick Sleeping

CHAPTER 5

LYNXTUN IMPLEMENTATION

5.1 Important Data Structures

In this section, we provide an overview of the important data structures that are used in our implementation.

5.1.1 LynxTunnel

LynxTunnel is the most important data structure. Each lynxtun process has a single instance of LynxTunnel (which we usually denote as L) as a globally defined variable. This holds most of the state in the program, and a pointer to it is passed as an argument to many functions that implement parts of the Lynxtun Protocol. The annotated definition of the LynxTunnel structure is given below.

```
typedef struct {

    /* Protocol Configuration Parameters */
    Duration dispatch_interval;
    Duration freeze_window;
    uint16_t dgram_size;
    Duration timestamp_tolerance;
    int ts_list_size; /* Size of recent timestamps list */
    uint16_t payload_size; /* Related to dgram_size */

    int host_id; /* Host ID defined by the protocol */

    /* Time to spend spinning leading up to a
       dispatch operation. */
    Duration spin_window;

    /* The public and tunnel IP addresses for the
       local and remote endpoints. */
    struct {
        /* The virtual network address assigned
           to the TUN interface. */
        struct {
            char addr_str[INET_ADDRSTRLEN + 1];
            struct in_addr in_addr;
        } vlan;
        /* The internet address that the two
           hosts can use to reach one another. */
        struct {
            char addr_str[INET_ADDRSTRLEN + 1];
            uint16_t port;
            struct sockaddr_in sockaddr_in;
            struct in_addr in_addr;
            struct sockaddr *sockaddr;
            socklen_t socklen;
        } inet;
    } host, peer;

    /* Data related to the TUN device */
    struct {
        char name[IFNAMSIZ + 1]; /* Name. Default: lynx0 */
        uint32_t queue_size;     /* TX queue sz. in bytes */
        int mtu;                 /* MTU*/
        int fd;                  /* The TUN dev fd. */
    } dev;

    int sock_fd; /* UDP/IP socket file descriptor */

    uint8_t aes_key[AES_KEY_SIZE]; /* AES secret key */

    /* GCM Context (incl. lookup tables) */
    struct aes256gcm_ctx gcm_ctx;

    /* Buffers that hold the staged datagram. */
    struct {
        struct {
            uint16_t size;
            uint8_t buffer[IP_MAXPACKET];
        } tun_ip_packet;
        LynxDatagram dgram;
        LynxEncryptedDatagram enc_dgram;
    } staging_area;

    /* Buffers that hold the dummy datagram. */
    struct {
        LynxDatagram dgram;
        LynxEncryptedDatagram enc_dgram;
    } dummy_dgram;

    /* List of recently accepted timestamps */
    Time ts_list[TS_LIST_MAX_SIZE];
    int ts_list_count;

} LynxTunnel;
```

5.1.2 Lynxtun Datagram Structures

The structures LynxDatagramHeader, LynxDatagram and LynxEncryptedDatagram are used to implement the Lynxtun Datagram structure defined by the protocol specification.

```
typedef struct {
    struct {
        uint32_t host_id;
        struct {
            uint32_t sec;
            uint32_t msec;
        } timestamp;
    } iv;
    uint16_t one;
    uint16_t data_length;
} LynxDatagramHeader;

typedef struct {
    LynxDatagramHeader header;
    uint8_t payload[MAX_LYNX_PAYLOAD_SIZE];
} LynxDatagram;

typedef struct {
    uint8_t header[sizeof(LynxDatagramHeader)];
    uint8_t tag[LYNX_TAG_SIZE];
    uint8_t payload[MAX_LYNX_PAYLOAD_SIZE];
} LynxEncryptedDatagram;
```

5.2 Initialization

When the lynxtun process starts, it creates and initializes a global `LynxTunnel` instance based on parsed configuration parameters. This instance is locked into main memory using `mlock()`, which will prevent it from being swapped out. This is important for deterministic runtime behavior.

Note that the `LynxTunnel` statically allocates memory buffers that are used to hold the staged datagram and the dummy datagram. Dynamic memory allocations are not suitable in the context of a realtime application, as they can introduce substantial delays. A fundamental design decision is to lock all necessary memory at startup.

Besides parsing the input configuration, initializing the `LynxTunnel` instance requires the following to be done:

- Creating and initializing the TUN device and obtaining a file descriptor. IP packets read from this file descriptor will be encapsulated in outgoing Lynxtun datagrams. IP packets unpacked from incoming Lynxtun datagrams will be written to this file descriptor.
- Creating and binding a UDP/IP socket that will be used to send and receive Lynxtun datagrams.
- Initializing the GCM context. This is described in Section 5.6.
- Making a number of system calls to configure the system.

- Generating the initial dummy datagram.

5.2.1 Initializing the TUN Device

On Linux, creating a TUN device starts by opening a file descriptor to a special file called TUN_CLONE_DEV.

```
if ((dev_fd = open(TUN_CLONE_DEV, O_RDWR)) < 0)
    err_sys("Could not open tun device fd.");
```

This file descriptor is then used in a series of ioctl() requests that are used to configure the TUN device.

We begin by setting the name of the TUN device. If there is a single instance of lynxtun running on the system, then the name will be lynx0. If additional instances get started, the number at the end will be incremented. This device can be seen using standard configuration utilities such as ip. We bring the TUN device up, set it as a point-to-point device with the source and destination addresses matching the configuration. This ensures that a routing table entry is added, so that IP packets sent to the peer's tunnel address get routed to the TUN device.

We set the MTU of the TUN device. If a value is specified by the configuration, then this is used. Otherwise, it will be set to the size of the fixed datagram payload, as specified by the protocol. This ensures that the largest IP packet that can be read from the TUN device file descriptor is guaranteed to fit inside the payload of an empty staged datagram at the start of a round.

We allow setting the size of the TX queue of the TUN device. If there are periods of time when data enters the TUN device at a rate greater than Lynxtun collects them (which is determined by the configuration), then the TX queue will start to fill up. If it overflows, then packets will be dropped. While it is not possible to sustain prolonged periods where this is the case without dropping packets, increasing the size of the TUN device's TX queue can reduce the probability that the buffer overflows due to intermittent periods of high activity.

If lynxtun is started with the -gateway option, then we modify the routing table to use the TUN device as the default gateway. This is useful for operating the tunnel as a network proxy.

5.2.2 Runtime System Configuration

5.2.2.1 Scheduler Configuration

The rt-dispatch argument can be used to assign the realtime SCHED_FIFO scheduler provided by Linux to the main dispatch thread. The rt-dispatch-prio argument specifies the realtime priority to use. Similarly rt-aux and rt-aux-prio can be used to enable realtime scheduling for non-dispatch threads. The `sched_setscheduler()` system call is used for the dispatch thread and `pthread_setschedparam()` is used for other threads.

The default scheduler on Linux is `SCHED_OTHER`, which is the standard round-robin time-sharing policy. This will be used with a nice value of 0 unless one of the aforementioned arguments is used.

Timer slack[1] determines the amount of time by which the wake-up of a process can be deferred in order to coalesce multiple wake-up events that are scheduled to occur at around the same time. The default time slack is 50 microseconds. We make the following call during initialization to set the timer slack to 1 nanosecond. This will not have any effect if realtime scheduling is used.

```
if (prctl(PR_SET_TIMERSLACK, 1, 0, 0, 0) == -1)
    err_sys("Could not set timer slack.");
```

5.2.2.2 CPU Isolation

The `cpu` and `cpu-aux` arguments can be used to set the CPU affinity for the dispatch and non-dispatch threads respectively. The arguments specify a CPU ID. This value is used to make the `sched_setaffinity()` system call for the dispatch thread and `pthread_setaffinity_np()` for other threads.

It is most useful to combine the use of these arguments with the `cpuisol` kernel boot parameter, that is used to isolate a CPU so that it does not receive IRQs and processes are not scheduled unto it by default. To take advantage of CPU isolation, the Lynxtun threads can be assigned to such an isolated CPU.

5.2.3 Main Dispatch Loop

The code below shows implementation for processing outgoing data and performing dispatch operations as specified in the protocol. Each iteration corresponds to a round that ends with a dispatch operation.

```
/* To calculate the initial dispatch time. */
next_dispatch = get_current_time();

while (GlobalRunning) {

    /* Clear the staged datagram */
    L.staging_area.dgram.header.data_length = 0;

    /* Calculate the target dispatch time and freeze time */
    next_dispatch = time_add(next_dispatch, dispatch_interval);
    freeze_time = time_sub(next_dispatch, freeze_duration);
```

[1] see man (7) time and man (2) prctl

```
    /* Target dispatch time was already missed. */
    if (get_nsec_until_time(next_dispatch) < 0) {
        /* Dispatch the dummy datagram immediately. */
        dispatch_dummy_dgram(next_dispatch, &L);
        /* To be used for the next dispatch time. */
        next_dispatch = get_current_time();
    }

    /* Target freeze time was missed */
    else if (get_nsec_until_time(freeze_time) < 0) {
        /* Wait until the dispatch time, then dispatch the
           dummy datagram.*/
        dispatch_dummy_dgram(next_dispatch, &L);
    }

    /* NOTE: dispatch_dummy_dgram() will generate
       a new dummy datagram after the dispatch is done. */

    /* Encapsulate IP packets */
    else {
        do {
            /* Do not wait for more than time remaining until
               freeze time when waiting for an IP packet. */
            read_timeout = time_diff(freeze_time,
                                     get_current_time());
            /* Stop if freeze time is reached */
            if (duration_to_nsec(read_timeout) < 0)
                break;
            /* Encapsulate the packet */
            res = mvdat_local_to_staged(&L, &read_timeout);
        } while (res.packet_added_to_staged_dgram);
        /* res.packet_added_to_staged_dgram will be false if
           the IP packet did not fit into the datagram. */

        /* Freeze and encrypt datagram, wait until dispatch
           time and perform the dispatch operation. */
        dispatch_staged_dgram(next_dispatch, &L);
    }
}
```

5.2.3.1 Dispatching a Dummy Datagram

As specified by the protocol, it may be necessary to dispatch a dummy datagram if either the target dispatch time or target freeze time has already passed at the start of a round. This is done using the `dispatch_dummy_dgram()` call. This call will block until the scheduled dispatch time, and then send the dummy datagram.

A dummy datagram can only be used once. Therefore a new dummy datagram is generated immediately after the dispatch is made. This ensures that there is a dummy datagram available at all times. The initial dummy datagram is generated as part of the initialization.

5.2.3.2 Encapsulating IP Packets

```
EncapsulatePacketResult mvdat_local_to_staged
    (LynxTunnel *L, Duration *read_timeout);
```

The mvdat_local_to_staged() function represents a request to read a single IP packet from the TUN device file descriptor and encapsulate it in the staged datagram. The read_timeout argument is set to the target freeze time, so that the request is aborted if an IP packet does not become available for reading until then. Internally, pselect() is used to wait for an IP packet to be available.

As discussed in the protocol specification, it is possible that an IP packet that is read does not fit into the staged datagram. In this case, it is cached until the next round, and the result returned mvdat_local_to_staged() will indicate that the packet was not encapsulated. In this case, another call to the mvdat_local_to_staged() will not be made in the current round and dispatch_staged_dgram() will be called. The first call to mvdat_local_to_staged() in a future round will encapsulate the cached IP packet instead of trying to read one from the TUN device. Since the MTU of the TUN device is not allowed to be greater than the payload size, it is guaranteed that the cached IP packet will fit inside an empty payload.

5.2.3.3 The Dispatch Operation

The call to dispatch_staged_dgram() begins freezing the staged datagram. The datagram header is generated, using the timestamp of the current time. The datagram is then encrypted. Once this is done, then the thread will wait until the specified dispatch time, and finally send the encrypted datagram using sendto() by specifying the tunnel peer's UDP/IP address.

5.2.4 Implementing Waiting Behavior

The calls for dispatching a dummy datagram and the staged datagram both require waiting until the specified dispatch time, and then executing the dispatch. It is therefore necessary to implement waiting behavior. The waiting period should not end sooner than the target time, but once that happens, the waiting period should end as soon as possible.

As we have observed in 4.3, using nanosleep() on its own does not present a viable solution, as there is significant variability in the wake-up times. As we have also noted, using a busy-loop that takes advantage of vDSO clock_gettime() offers a superior solution. This enables us to implement *spinning* while waiting for the target time without yielding control to the kernel by making a system call. However, as is the case with spinning, this will cause the CPU to run at full speed. Simply from the viewpoint of making efficient usage of processor time, this is not preferable. However, there is a more important reason why spinning alone is unsuitable for Lynxtun. That is, if the CPU is used at maximum capacity, it will become hotter. This can lead to the system decreasing the CPU frequency in order to allow it to cool down, which can also lead to delays.

The correct solution is to use a combination of the two approaches.

```
void
wait_until (Time t, Duration spin_duration)
{
    if (get_nsec_until_time(t) < 0)
        return;

    Time spin_time = time_sub(t, spin_duration);

    if (get_nsec_until_time(spin_time) > 0)
        clock_nanosleep(CLOCK, TIMER_ABSTIME, &spin_time, NULL);

    /* Spin */
    while (get_nsec_until_time(t) > 0);
}
```

The `wait_until()` function is used to wait for dispatch operations. It will return immediately if the target time has already passed. Otherwise, it will use `clock_nanosleep()` to wait until a time that is earlier than the target time. Once this returns, the remaining time is spent spinning by constantly checking the current time until the target time has been reached.

The length of time that should be spent spinning is provided as a configuration argument. It should only be as long enough as to compensate for variability due to using `nanosleep()`.

Note that, as stated in Section 4.2.4, this implementation requires that `clock_gettime()` is available in the vDSO.

5.2.5 Dummy Datagram Generation

The Lynxtun Protocol specifies that a dummy datagram should be sent in the event that it is found at the start of a round that there the freeze deadline has already been missed, meaning that there is no time for encapsulating outgoing datagrams. It is up to the implementation to decide how to generate the dummy datagram, but it is required that it is indistinguishable from an encrypted Lynxtun datagram.

Here, we have two options. The first is that we encrypt a Lynxtun datagram with zero data length. The second is that we use a random data source. There are two random source devices implemented by the Linux kernel, which are `/dev/urandom` and `/dev/random`.

We argue that the best alternative is to use the first approach. `/dev/random` depends on accumulated entropy in the entropy pool of the system. The size of this pool decreases as entropy is consumed. Trying to read from `/dev/random` will block until there is enough entropy in the pool to generate the requested amount of data. This is unsuitable for our purposes. /dev/urandom does not block if the entropy pool is depleted. Instead, it falls back to a pseudo-random number generator. We cannot assume that the statistical characteristics of this method will match that of encrypted Lynxtun datagrams.

In order to solve these problems, we generate the dummy datagram using the same method that actual datagrams are encrypted. The data length field in the header is set to 0. The timestamp shows the time that the dummy datagram was generated.

5.3 Processing Incoming Data

Processing incoming data is performed on a separate thread. pselect() is used to wait for an incoming datagram to be available to be read from the UDP/IP socket file descriptor. When it is, this datagram is processed by verifying and performing authenticated decryption as specified by the protocol.

If a datagram is accepted, then we know from the header how much of the payload is actually used. IP packets can only be encapsulated as a whole according to the Lynxtun protocol. Therefore, the procedure to unpack the datagram is as follows:

```
uint8_t *skbuf = dgram.payload;
uint32_t total_size = dgram.header.data_length;
uint32_t processed_size = 0;
uint16_t packet_size;
struct iphdr *iphdr;

while (processed_size < total_size) {
    iphdr = (struct iphdr *) skbuf;
    packet_size = ntohs(iphdr->tot_len);

    /* Write the unpacked IP packet to the TUN dev. */
    write(L->dev.fd, skbuf, packet_size);

    processed_size += packet_size;
    skbuf += packet_size;
}
```

5.4 Regarding Byte Order

Endianness is a difficult topic to deal with [19, 31]. We have to pay special attention to issues related to endianness since we are implementing a network protocol. In this section, we share some of the lessons we have learned over the course of implementing Lynxtun, and present the approach that we use.

The standard that has to be followed when implementing a network protocol is to always use big-endian order. That is why big-endian order is also referred to as *network byte order*. Host order, as implied by its name, depends on the host. Host order and network order are different in the case of a little-endian machine. Conversion functions such as `ntohl()` and `htonl()` perform byte-swapping if necessary. They are noops on a big-endian machine.

Always using big-endian order when exchanging messages over the internet means little-endian machines incur conversion overhead in network code. This happens even if two little-endian machines communicate. While the conversion in such a case might appear to be redundant, it is a necessity. The lower layers of the internet are designed to be stateless. The concept of a stateful session only appears in higher layers of the OSI model. This design decision is crucial to making the internet viable. The overhead and complexity introduced by having to negotiate whether or not byte-swapping should be done prior to all exchanging IP packets would be debilitating. The overhead incurred by little-endian machines is fully justified, and having a convention is highly beneficial.

[31] observes that using such a convention in designing the external interface of an encryption function would enable various software components to interact without having to know about the internal representation being used in each case. They suggest that the convention should be to represent data as an array of bytes. The implication is that, if the internal representation makes use of multi-byte words, then these these should be converted to big-endian order when the data is represented as an array of bytes.

As we will discuss in more detail when we cover our AES-256 and GCM implementation (Sections 5.5 and 5.6), in both cases we use lookup tables that are 32-bit unsigned inteegers. Therefore, the internal data representation for plaintext, ciphertext and key buffers should also be the same.

Changing data representation from arrays of bytes to arrays of multi-byte words (and vice versa) requires byte-swapping to be done on little-endian machines.

Encryption and decryption is performed on Lynxtun Datagrams. The fields of the datagram header are represented in network byte order, and so are encapsulated IP packets. The content of the payload is irrelevant. This is in agreement with what [31] recommends, in that we require function signatures that accept data represented as an array of bytes. These functions would convert the data to the internal representation at the start, and back to the canonical representation at the end.

Within the AES-256 and GCM functions, work is performed using the internal representation, which is arrays of 32-bit unsigned integers. This leads to the following issue. GCM needs to call AES-256 directly. The internal representation of each is the same. However, if we only have an AES-256 function signature that expects data to be represented as an array of bytes, then this will require many redundant conversions. These additional conversions cannot be justified. Unlike the case of communicating network hosts, the two functions run on a single machine. Therefore we know that the endianness is the same in both cases. We further know that the internal representation of the two are compatible. Therefore, we implement additional function signatures that accept data to be provided in the native format for the function. Furthermore, the AES secret key argument is always expected to be provided according to the internal representation. Otherwise, we would perform redundant conversions with each call.

In general, byte order conversion is a requirement of data serialization. We serialize data when we send it over the network or write it to a file. We argue that, within the context of a single process, data conversion should take place close to points of serialization. Within the process, different components that use compatible internal data representations should be allowed to take advantage of this fact.

5.5 AES-256 Implementation

We have implemented AES-256 [21, 44] to use as the primary block encryption function in Lynxtun. Our implementation is based on the four lookup table approach described for 32-bit processors in Section 5.2.1 of [21]. We use the *equivalent inverse cipher* method for implementing decryption described in [44].

The internal representation represents an AES block as an array of 4 `uint32_t` values, and an AES key as an array of 8 `uint32_t` values. Following the discussion in Section 5.4, we implement the following functions:

```
void aes256_encrypt_native(uint32_t *plaintext,
   uint32_t *ciphertext,
   uint32_t *key);

void aes256_decrypt_native(uint32_t *plaintext,
              uint32_t *ciphertext,
           uint32_t *key);

void aes256_encrypt(uint8_t *plaintext,
            uint8_t *ciphertext,
                    uint32_t *key);

void aes256_decrypt(uint8_t *plaintext,
            uint8_t *ciphertext,
                    uint32_t *key);
```

5.6 GCM Implementation

We have implemented the modified version of the GCM mode of operation [38] using AES-256 block encryption. Our modifications are described in the protocol specification. The implementation uses byte-decomposition lookup tables in order to speed up the multiplication of 128-bit field elements with the hash key. The hash key is simply $E_K(0)$, where E is AES-256 block encryption and K is the shared secret key.

Aside from the early-stopping modification that we have made, which extends GCM functionality, we have implemented a full GCM implementation and have tested it with published GCM test vectors. That is, even though additional authenticated data (AAD) is not used in Lynxtun, and the tag and IV lengths are fixed at 128-bits and 96-bits respectively, our GCM implementation does not have these limitations. As such, the same implementation could easily be adapted to other usecases.

5.6.1 Interface

We implement the following functions.

```
void aes256gcm_encrypt (uint8_t *authdata, uint8_t *plaintext,
    uint8_t *ciphertext, uint8_t *iv, uint8_t *tag,
    uint32_t *authdata32, uint32_t *plaintext32,
    uint32_t *ciphertext32, uint32_t *iv32,
    uint32_t authdata_len, uint32_t ciphertext_len,
    struct aes256gcm_ctx *gcm_ctx);

int aes256gcm_decrypt (uint8_t *authdata, uint8_t *plaintext,
    uint8_t *ciphertext, uint8_t *iv, uint8_t *tag,
    uint32_t *authdata32, uint32_t *plaintext32,
    uint32_t *ciphertext32, uint32_t *iv32,
    uint32_t authdata_len, uint32_t ciphertext_len,
    struct aes256gcm_ctx *gcm_ctx);

void aes256gcm_encrypt_ext (uint8_t *authdata, uint8_t *plaintext,
    uint8_t *ciphertext, uint8_t *iv, uint8_t *tag,
    uint32_t *authdata32, uint32_t *plaintext32,
    uint32_t *ciphertext32, uint32_t *iv32,
    uint32_t authdata_len,
    uint32_t ciphertext_sig_len,
    uint32_t ciphertext_tot_len,
    struct aes256gcm_ctx *gcm_ctx);

int aes256gcm_decrypt_ext (uint8_t *authdata, uint8_t *plaintext,
    uint8_t *ciphertext, uint8_t *iv, uint8_t *tag,
    uint32_t *authdata32, uint32_t *plaintext32,
    uint32_t *ciphertext32, uint32_t *iv32,
    uint32_t authdata_len,
    uint32_t ciphertext_sig_len,
    uint32_t ciphertext_tot_len,
    struct aes256gcm_ctx *gcm_ctx);
```

The internal representation represents an AES block as an array of 4 `uint32_t` values, and an AES key as an array of 8 `uint32_t` values. Functions with `_ext` use byte array representation, and the others use internal representation.

The `ciphertext_sig_len` and `ciphertext_tot_len` arguments are related to our early-stopping modification. The first argument specifies the actual data that should be used to calculate the authentication tag (this is the used size of a datagram payload, excluding padding), and the second represents the total size of data to be encrypted (which is the size of the datagram payload).

We have discussed issues regarding data representation conversion in Section 5.4. There is an additional issue that we must address that is specific to GCM.

The sizes of the data buffers in AES-256 are all fixed. They are 16 bytes for plaintext and ciphertext buffers, and 32 bytes for the key buffer. This is not the case with GCM.

Although there are limits to the sizes of various data buffers like the additional authenticated data (AAD), plaintext and ciphertext buffers, these are large enough that, for the purposes of this discussion, we can say they are arbitrarily large. Secondly, GCM allows AAD, plaintext and ciphertext to end on byte boundaries, rather than a 16-byte block boundary. In other words, GCM allows encrypting data that is 7 bytes long, for instance.

This poses a problem when we have to convert from 8-bit arrays to 32-bit arrays with byte-swapping. We cannot convert in-place because it is possible that the data does not end on a 32-bit boundary. Otherwise we would be corrupting adjacent memory.

Since the fields can be arbitrarily large, we cannot allocate static memory to use as working buffers without introducing an arbitrary limit. Additionally, there would have to be separate buffers to allow calls to be made on multiple threads simultaneously. Workarounds could be found, but it is bad design.

Using dynamic memory allocation using `malloc()` would avoid this problem, but we have already discussed reasons why this is not appropriate in the case of Lynxtun.

Our solution was to make use of working buffers that are allocated on the stack: *stack buffers*. In Lynxtun, the size of the data that is encrypted/decrypted is fixed. Therefore, the caller knows how large the working buffers must be. The caller can allocate these on the stack (or statically), and pass a pointer to the GCM functions. The `uint32_t` pointers with names that end with 32 in the function declarations above are such working buffers.

We implement the `encrypt_payload()` and `decrypt_payload()` functions as wrappers that handle the allocation of working buffers on the stack.

5.6.2 The GCM Context and Lookup Tables

The GCM Context holds the AES-256 Key, hash key and lookup tables. It is a part of `LynxTunnel` and is initialized using the function `aes256gcm_init` during initialization.

```
struct aes256gcm_ctx {
    uint32_t K[AES_KEY_SIZE / 4];    /* AES-256 Key Data */
    uint32_t H[4];             /* Hash Key Data: E(K, 0) */
    /* Lookup tables (depends on key) */
    uint32_t M[AES256GCM_LOOKUP_TABLE_UINT32_VALS];
    uint32_t iv_len; /* Length of the IV in bits (96) */
    uint8_t tag_len; /* Auth tag length in bits (128) */
};

void aes256gcm_init (uint8_t *key, uint32_t iv_len,
                     uint8_t tag_len,
                     struct aes256gcm_ctx *gcm_ctx);
```

The lookup tables are used to implement the $GF(2^{128})$ field multiplication operation in an efficient way. Our approach is based on the software implementation section of [38], and is presented in Appendix D.

This method takes advantage of time-memory tradeoff. Implementing the multiplication of arbitrary field elements is expensive to implement in software because it requires many bitshift operations.

Instead, our implementation uses 16 lookup tables that combined use 64 KB. They are generated for a specific hash key. The hash key is derived from the AES-256 shared key, which remains fixed throughout the lifetime of the process. Therefore, the lookup table generation only has to be performed during initialization and therefore does not disturb the dispatch process. Even though this was not essential, we nevertheless optimized the generation of the lookup tables (as described in Appendix D) so that the number of field multiplication operations necessary to generate the tables is minimized.

CHAPTER 6

EMPIRICAL ANALYSIS

The security of Lynxtun depends on two things: the cryptographic protocol and the regulation of the dispatch process. The security of the cryptographic protocol follows from the assumption that the cryptographic primitives it relies on, and the manner in which these are combined are secure. It is therefore possible to discuss the security of the cryptographic protocol in abstract terms. If the cryptographic protocol is secure, then it will not depend on runtime behavior, and we do not need to observe runtime behavior in order to reason about its security. This is not the case when we consider the security due to the regulation of the dispatch process. As we have stated throughout this thesis, achieving sufficient determinism of runtime behavior poses a significant engineering challenge. We have discussed how the protocol should be designed, how the implementation should be carried out, and how the system should be configured in order to achieve this. Whether or not we have been successful in this objective can only be evaluated based on empirical observations of an actual Lynxtun deployment. That is the purpose of this section.

6.1 Data Collection Challenges

Our aim is to compare the properties of the communication flowing through the tunnel against the properties of the dispatch process. Therefore, we need to collect data on both of these.

We can observe the dispatch process of a given Lynxtun endpoint at three places. We can take measurements within the Lynxtun process itself, such as recording the time when the dispatch operation was made. We can record IP packets on the same host that is running Lynxtun. Finally, we can observe the network traffic somewhere along the network connecting the two Lynxtun endpoints. When we consider our threat model, the situation of an attacker is best described by the third alternative. This is what we are trying to protect against.

Similarly, when we consider how to observe the underlying communication, we again see that there are three alternatives. We can make observations within the Lynxtun process and we can capture IP packets on the same host. If the data arrives through a private network, then we can also record traffic on this network.

6.1.1 Capturing Network Traffic

If we take measurements on the network, then this will contain noise that is due to the network. This noise is difficult to predict or control, but we can assume that it will be on the order of milliseconds. The further away from the source the measurements are made, the greater the amount of noise there will be. In general, this noise is beneficial in terms of the security of Lynxtun, as it will hide residual patterns in the dispatch process. It is harder for an attacker to detect these patterns and use them to make inferences about the communication in the presence of noise. However, we want to be able to observe whether there are any residual patterns to begin with. One of the design objectives of Lynxtun is to limit the amount of variability in the dispatch process. The variability in the times between successive dispatch operations should be on the order of microseconds. We cannot assess this if our recordings contain noise on the order of milliseconds. Therefore, we need want to eliminate noise as much as possible when designing our data collection methodology.

This suggests that we should make measurements either within the Lynxtun process itself, or on the same host. This raises another problem, which is reminiscent of one of the key problems in Quantum Physics. Making an observation influences the observed phenomena. This is true in both cases.

The documentation of the pcap packet capturing library [3] contains the following warning:

> "If the time stamp is applied to the packet when the networking stack receives the packet, the networking stack might not see the packet until an interrupt is delivered for the packet or a timer event causes the networking device driver to poll for packets, and the time stamp might not be applied until the packet has had some processing done by other code in the networking stack, so there might be a significant delay between the time when the last bit of the packet is received by the capture device and when the networking stack time-stamps the packet; the timer used to generate the time stamps might have low resolution, for example, it might be a timer updated once per host operating system timer tick, with the host operating system timer ticking once every few milliseconds; a high-resolution timer might use a counter that runs at a rate dependent on the processor clock speed, and that clock speed might be adjusted upwards or downwards over time and the timer might not be able to compensate for all those adjustments; the host operating system's clock might be adjusted over time to match a time standard to which the host is being synchronized, which might be done by temporarily slowing down or speeding up the clock or by making a single adjustment; different CPU cores on a multi-core or multi-processor system might be running at different speeds, or might not have time counters all synchronized, so packets time-stamped by different cores might not have consistent

In order to have better deterministic runtime behavior, we run Lynxtun using high realtime priority. This means, the Lynxtun process can preempt other tasks on the system. If we use tcpdump (which uses the pcap library) in order to capture network packets on the same host, then it is likely that Lynxtun has a disruptive effect on tcpdump. It might be the case that Lynxtun is sending packets at perfectly regular intervals, but preemption of tcpdump can mean that this is not captured in the recorded timestamps. If the possibility that such disruption will occur depends on how much communication there is, then the output of tcpdump will suggest that the dispatch process is correlated with the underlying communication, even though this might not have been the case were the recording done somewhere else. In order to avoid this problem, we do not run tcpdump to capture packets on the same host that is running Lynxtun. Instead, we capture network traffic on a separate physical host, but this host is *close* to the source. It is on the same LAN.

Isolating both tunnel endpoints *and* the hosts that capture network traffic completely requires a large number of dedicated physical hosts to be used. Such a setup is shown in Figure 6.1.

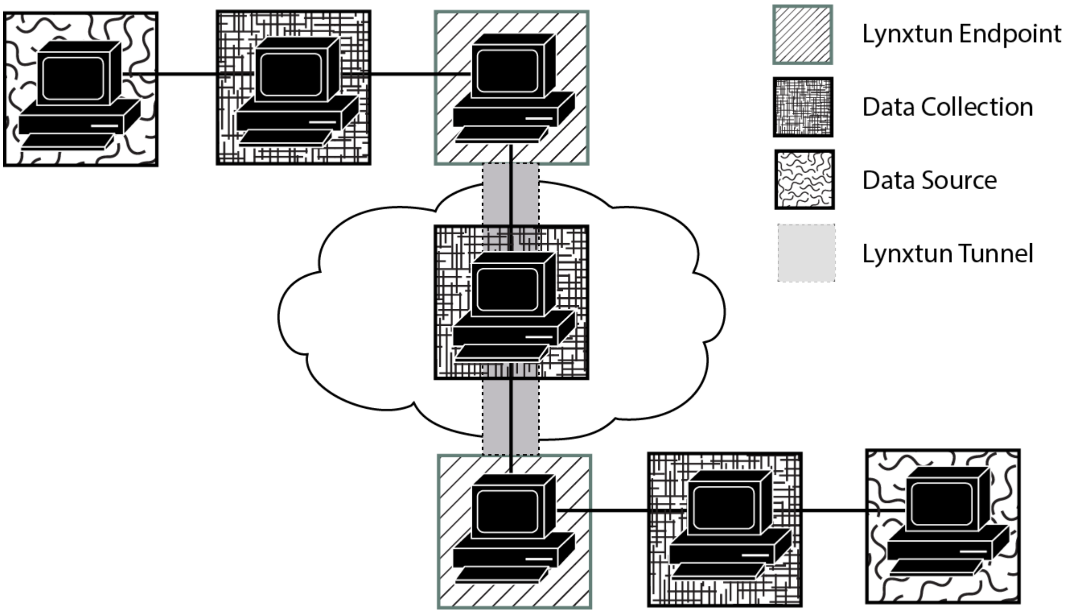


Figure 6.1: Ideal Full Data Collection Setup

6.1.2 Taking Measurements Within the Lynxtun Process

There is a similar problem when we consider taking measurements directly within Lynxtun. Since we are dealing with nanoseconds, this is the same order of magnitude that it takes to execute single machine instructions. Modifying the code of Lynxtun to add support for taking measurements is not possible to do without running the risk of affecting its realtime behavior. On the other hand, we have to do this in order to observe variability at such a high resolution. The most important problem to address is how to collect the results. In order for us to see the results, they have to be written to a file at some point. This can either be a pipe like standard output, or a file on the filesystem. I/O operations require system calls being made. These require control to be passed onto the kernel and the process put to sleep until the disk I/O has finished, which can take many milliseconds. This is especially detrimental to deterministic realtime behavior. The solution we have found is as follows. Global memory arrays are statically allocated and locked using `mlock()` during initialization. Collected samples are written to these memory locations. While memory access also takes time, it is much quicker than disk access. It also does not require relinquishing control flow. The implication is that the tunnel cannot continue running indefinitely. The tunnel main loop terminates when the specified number of samples has been collected. The samples are written to disk before the process terminates. This way, file I/O is not performed why dispatches are being made.

6.1.3 Dedicated Physical Hosts

It is imperative that the Lynxtun endpoint that is being analyzed is a dedicated physical host. That is, it is not appropriate to use virtual machines for this purpose. As we have noted in 4.2.4, there can be problems due to lack of vDSO support when running inside a VM. Even if this is not the case, the virtualization layer introduces significant complexity into the system making it much more difficult to reason about how and when instructions get executed on the baremetal hardware.

6.2 Data Collection Methodology

We now present our data collection methodology in light of the discussion of the previous section.

6.2.1 Experimental Setup

Figure 6.2 shows an overview of our experimental setup. It is composed of two physical hosts (ctrl and lynx) in addition to one virtual host (lynx-aux) that is running inside the VirtualBox hypervisor on ctrl. We set up a Lynxtun tunnel between lynx and lynx-aux. We analyze the dispatch process of lynx. The hardware specification of lynx is given in Appendix B. The kernel specifications are given in Appendix C.

The hosts ctrl and lynx are connected over a Gigabit ethernet LAN. lynx-aux is connected to the same LAN using a bridged network adapter.

Experiments produce the three sets of data described below.

1. The time at which a dispatch operation was made by the Lynxtun process running on lynx, and the amount of data that was in the staged datagram in bytes.
2. The time at which a Lynxtun datagram was received by the Lynxtun process running on lynx, and the amount of data that was in this datagram in bytes.
3. A pcap file generated using tcpdump containing encrypted Lynxtun datagrams that were sent from lynx to lynx-aux. This is recorded using tcpdump on ctrl.

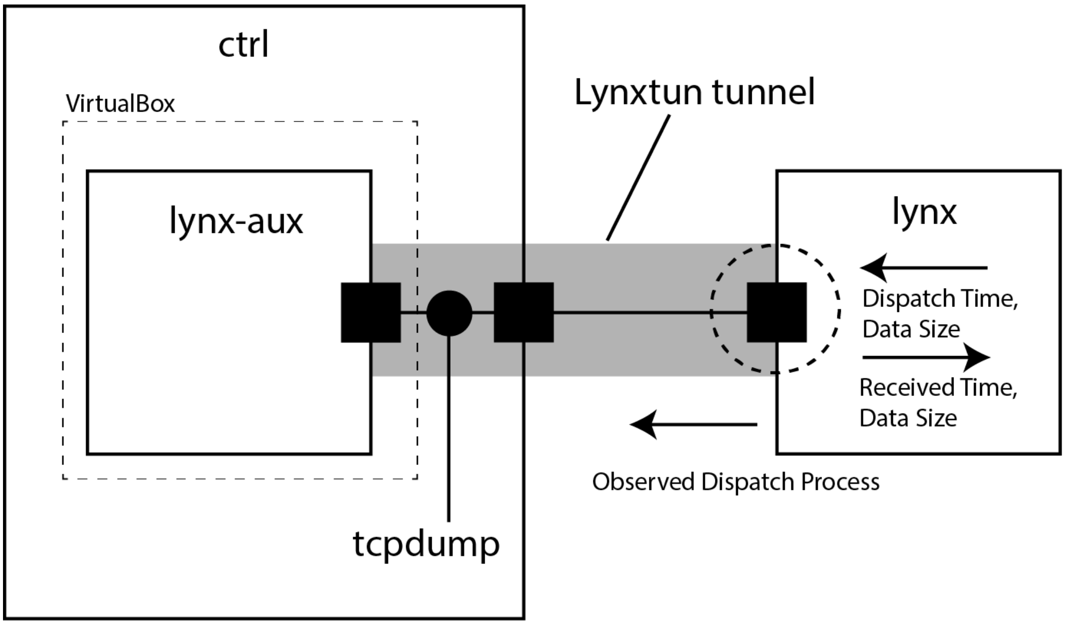


Figure 6.2: Data Collection Setup

> Note that we have to configure tcpdump only to capture IP packets without a fragmentation offset since Lynxtun datagrams can have been fragmented. We should only capture the initial fragment of each Lynxtun datagram.

By using the data provided by the first two items, we are able to determine the total amount of incoming and outgoing data that is processed by the Lynxtun process running on lynx per second.

By using the timestamps included in the first item, we are able to measure with high resolution how much variation there is in the times between successive dispatch operations. We calculate the difference between successive timestamps and subtract from this the target dispatch interval.

By using the timestamps in the pcap file, we also calculate the time between successive Lynxtun datagrams that were sent by lynx to lynx-aux. The difference is that these are recorded on a different physical host on the network.

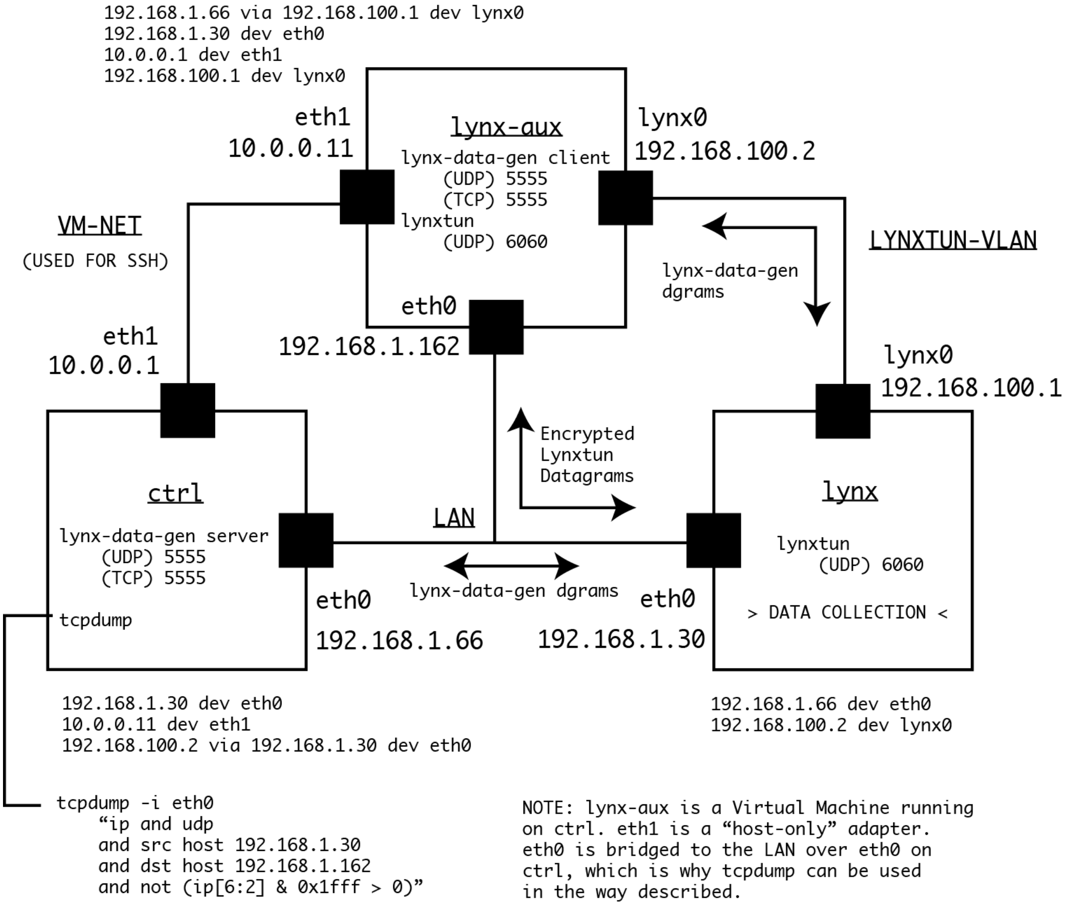
We analyze the dispatch process while controlling the amount of data that is flowing through the tunnel. The expectation is that changes in the underlying communication should not be observable in the dispatch process.

We implemented a simple C utility called lynx-data-generator. This is used to generate the IP packets that get encapsulated through the tunnel. lynx-data-generator can be started either as a server or a client. The server listens on a TCP socket. Data generation begins on both sides when the client establishes a connection to this socket. The operation is made up of rounds. Each round lasts 10 seconds. In any given round, lynx-data-generator either sends data to its peer at a fixed rate, or it is idle. The behavior for the i th round is different for the client and server, and is given below:

- IDLE if $i \mod (4) = 0$
- Client sends data if $i \mod (4) = 1$
- Server sends data if $i \mod (4) = 2$
- Both client and server sends data if $i \mod (4) = 3$

This corresponds to 10 second periods where either there is no communication, there is communication in one direction, or there is bidirectional communication. By tunneling this communication through Lynxtun, we are able to observe the dispatch process when the tunnel is idle, when there is only incoming data, when there is only outgoing data, and when there is both incoming and outgoing data.

When data is being sent, 1000 bytes of data is sent to the peer at 10 millisecond intervals using UDP.



```
192.168.1.66 via 192.168.100.1 dev lynx0
192.168.1.30 dev eth0
10.0.0.1 dev eth1
192.168.100.1 dev lynx0
```

```
192.168.1.30 dev eth0
10.0.0.11 dev eth1
192.168.100.2 via 192.168.1.30 dev eth0
```

```
192.168.1.66 dev eth0
192.168.100.2 dev lynx0
```

```
tcpdump -i eth0
    “ip and udp
    and src host 192.168.1.30
    and dst host 192.168.1.162
    and not (ip[6:2] & 0x1fff > 0)”
```

NOTE: lynx-aux is a Virtual Machine running on ctrl. eth1 is a “host-only” adapter. eth0 is bridged to the LAN over eth0 on ctrl, which is why tcpdump can be used in the way described.

Figure 6.3: Experimental Setup 1

The experimental setup is based on Figure 6.2 and is shown in Figure **??**.

Figure **??** also shows the routing table configuration on each of the hosts. This configuration is done such packets from ctrl to lynx-aux are routed over lynx, where they enter the tunnel and correspond to *outgoing* traffic. Similarly, packets from lynx-aux to ctrl are routed to the `lynx0` TUN device on lynx-aux, where they are read by the Lynxtun process on lynx-aux and get sent to lynx over the tunnel. This corresponds to *incoming* traffic. This setup ensures that the Lynxtun process being analyzed is running on a dedicated physical host that is isolated as much as possible.

The ctrl host is responsible for driving the experiment, and does so by executing remote commands on the other two hosts using SSH. We use a separate isolated host-only network defined in VirtualBox (corresponding to `eth1` on ctrl and lynx-aux) so that ctrl can access lynx-aux directly.

Data is generated using the lynx-data-generator program above. This program is run on ctrl and lynx-aux. ctrl runs as the server and lynx-aux runs as the client.

The procedure for running an experiment is as follows:

1. Start tcpdump capture on ctrl.
2. Create Lynxtun tunnel between lynx and lynx-aux.
3. Start lynx-data-generator server on ctrl.
4. Start lynx-data-generator client on lynx-aux.
5. lynx-data-generator connection is established and the two sides begin sending data.
6. All processes stop ones 10,000 dispatch samples have been collected on lynx.

We perform three trials for each experiment.

6.3 Analysis of Results

We present results collected using various configurations under Appendix E. The configurations are defined there. In this section, we analyze the results.

Configuration 1 uses a fully preemptible kernel that uses the RT_PREEMPT kernel patch. with both realtime scheduling and CPU isolation enabled for the Lynxtun process. We observe that the difference of the duration of each round and the target dispatch interval oscillates around zero, and remains under 1 microsecond. The oscillation is expected, and due to the fact that missing a dispatch deadline means there is less time from the start of the next round to the dispatch operation, so that the time between two dispatch operations is less than the target dispatch interval. This ensures that the mean round duration equals the target dispatch interval, which is observed in the data. We see that the variability of internally measured dispatch delays tends to increase when the tunnel is idle in Trial 1. This is not observed in the other two trials. This behavior is likely to be related to our observations in section 4.3, where we saw that increased scheduler load tends to improve deterministic behavior if real-time scheduling is used on a preemptible kernel. These fluctuations do not result in deviations greater than 1 microsecond, and is not reflected in the externally recorded dispatch delays. That being said, there are some irregularities that can be observed in the external dispatch delay data for Trial 1, that coincide with a period of increased tunnel activity. It is possible that this is due to an unrelated cause. In any case, the deviations here are under half a millisecond. Other than this, the externally recorded delays are fairly regular and well under a millisecond. These results indicate that, when used with a correctly configured system, Lynxtun is capable of providing a high level of Traffic Flow Confidentiality.

The difference between Configuration 1 and 2 is that, in Configuration 2 we do not use CPU isolation. We see that the internally recorded dispatch variability is extremely well regulated, with deviations that are on the order of a couple hundred nanoseconds. Similar to the results from Configuration 1, we see that the externally observed dispatch process is also fairly regular. While there are a few relatively greater fluctuations in Trial 3, similarly to Configuration 1, the deviations are well under 1 millisecond. These results suggest that, in this particular deployment, using a realtime scheduler without CPU isolation is capable of delivering highly deterministic runtime behavior.

This is not the case in Configuration 3, which uses CPU isolation but not realtime scheduling. We see peaks in the internally recorded dispatch variability, and these coincide with periods of increased variability in the externally observable data and changes in tunnel activity. The magnitudes of the fluctuations in internally recorded data is significantly higher, in the order of hundreds of microseconds. While these fluctuations are reflected in the externally recorded data, the fluctuations in externally observed delays remain under 1 millisecond.

In Configuration 4, we switch over to a regular preemptible mainline kernel rather than a fully preemptible kernel with RT_PREEMPT. The results clearly indicate that the internally recorded dispatch variability is highly correlated with tunnel activity. On the other hand, deviations from the target dispatch interval remain under 1 microsecond. That is, the fluctuations are evidently due to changes in tunnel activity, but the magnitude of these fluctuations are small. We see that the externally recorded data is highly regular. This result suggests that, while it is preferable to use a realtime kernel in order to minimize the level of correlation between the dispatch process and the underlying communication, it is still possible to limit the magnitude of the fluctuations to under a microsecond when running on a non-realtime kernel so that while the two processes are correlated, this cannot be observed through data captured on the network.

The internally recorded dispatch delays recorded for Configuration 2 are presented in 6.4 as a representative example. The values show the time difference between the actual dispatch time and the target dispatch time for each round. We see that the deviations are always less than 1 microsecond, and oscillate around zero.

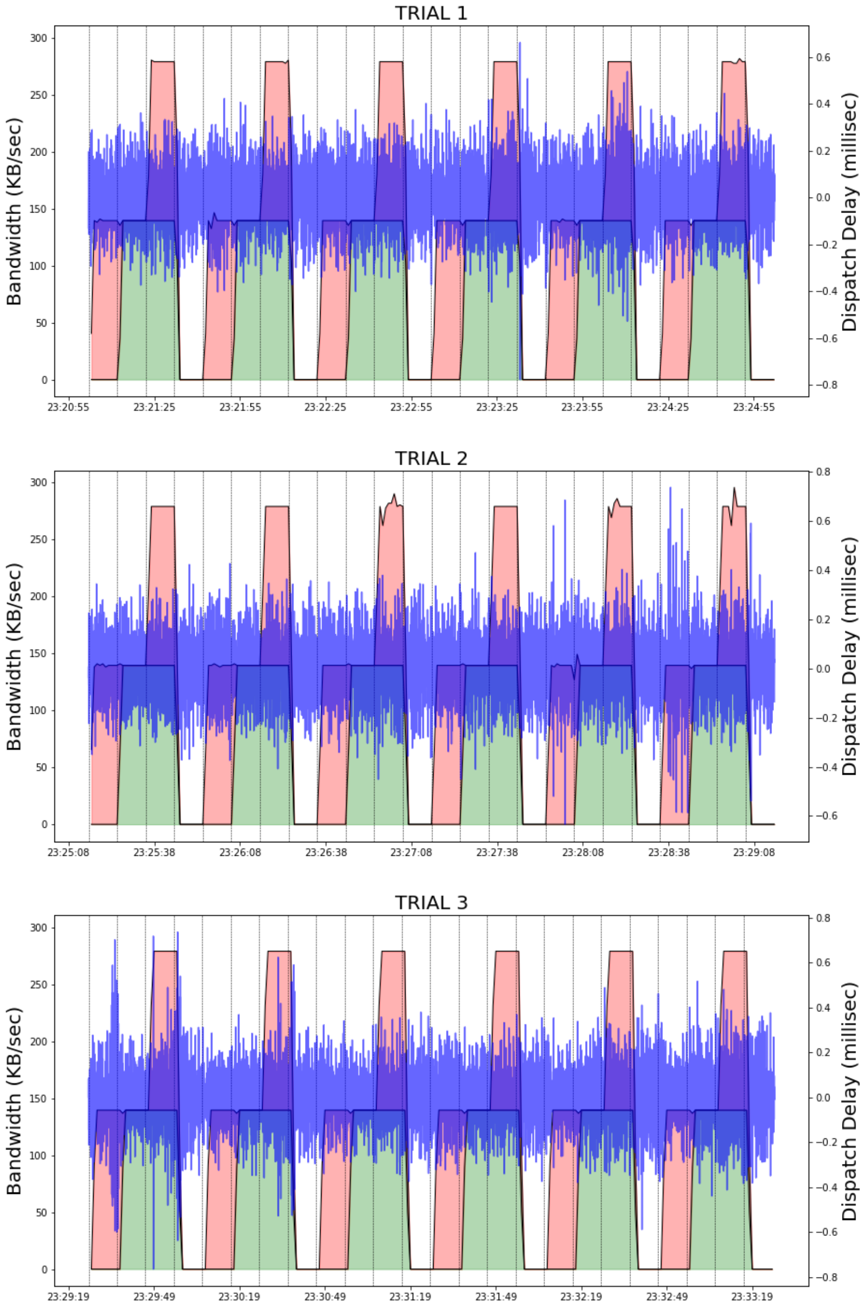


Figure 6.4: Dispatch Delays

CHAPTER 7

CONCLUSION

7.1 Our Contribution

In this thesis, we have successfully developed a point-to-point VPN solution that is capable of achieving high levels of security within the context of a userspace implementation. Our solution sends fully encrypted and authenticated datagrams that are encrypted and authenticated using a custom cryptographic protocol that we have designed, and is based on the security of the AES-256 block cipher. Our Lynxtun implementation is capable of achieving high levels of traffic flow confidentiality, such that fixed-sized encrypted datagrams are sent at regular intervals, and the deviation from these intervals remains is less than several microseconds regardless of how much communication is taking place. Furthermore, it is possible to achieve this level of security when running on a stock Linux kernel and general-purpose hardware. It is easy to configure a stock kernel through kernel boot parameters and runtime configuration without the need for custom compilation. Therefore, using Lynxtun is as simple as installing and running any userspace application, and it is possible to use Lynxtun on a wide variety of hardware configurations. Our design of Lynxtun does not require it to be running on dedicated hardware. On a correctly configured everyday workstation, Lynxtun can still be used to provide high levels of traffic flow confidentiality.

The standard that we set as our security requirement was extremely strict. That is, we wanted communication to be unobservable, meaning that the regularity of the dispatch process should be unaffected by whether or not any data is flowing through the tunnel. We have been successful in achieving this, provided that the system has been configured correctly. On the other hand, even on a system where no configuration whatsoever is done, the security provided by Lynxtun is nevertheless significant. Even if traffic flow confidentiality is not achieved in the sense that communication is unobservable, it will be extremely difficult for an attacker to gain enough information in order to learn about the content of the communication. Statistical traffic analysis attacks are difficult in general, as they require a lot of data to be collected. Furthermore, an attacker needs to have prior knowledge of signatures that result from various types of network activity. Any traffic shaping will make the task much more difficult for the attacker. In most usecases, ensuring that communication remains purely unobservable is not necessary. Therefore, using Lynxtun will provide very high levels of security, even if the system is not tuned with the intention of making communication unobservable. Very little information beyond the possible existence of communication can be inferred from observing the timing of Lynxtun datagrams. Furthermore, the security due to the use of fully-encrypted datagrams is significant even without any traffic shaping.

That being said, it is also possible to fine-tune a system to achieve even greater levels of security. Other solutions such as IPsec and TLS feature TFC mechanisms. However, we have shown that achieving TFC requires a deterministic dispatch process, which poses a difficult engineering challenge. Unlike other solutions, Lynxtun has been build around this central objective to make. Achieving this requires addressing cross-cutting issues at all layers of the system, including protocol design, implementation design, runtime system configuration, kernel compilation configuration and hardware issues. We have covered all of these issues in this thesis. We have analyzed the complex interactions between various parts of the system and identified ways in which these can affect deterministic runtime behavior. We have also presented solutions to these problems.

As we have stated in the introduction, a benefit of aiming for perfection is the discovery of what keeps us from attaining it. It is important to be aware of what the limitations are when developing such a system. Our discussion regarding achieving determinism transcends Lynxtun, and relates to all realtime applications. The problems that we have identified, the solutions that we suggest, and the limitations that we point out are relevant to all such applications. We have addressed the question of how one could achieve highly deterministic runtime behavior in a userspace application running on the Linux kernel. The answers we provide are pertinent to all developers who have realtime requirements.

We have developed our own network protocol: the Lynxtun Protocol. This is a Layer-3 encapsulation protocol that generates fully encrypted, timestamped and authenticated datagrams. It is designed with realtime requirements in mind. It can be used as a starting point to develop other network protocols with similar requirements.

The custom cryptographic protocol that we have designed is largely independent of the realtime requirements of the system. With slight modifications, this cryptographic protocol can be used in other contexts. There are two aspects of this protocol that we would like to emphasize. The first it that it allows authenticated encryption that relies on a single shared secret AES-256 key, while at the same time all datagrams can be decrypted and authenticated independently. We argue that the correct approach to implementing a Layer-3 tunneling protocol is to use unreliable delivery, as we have discussed. Our approach solves this problem. Secondly, our approach defines a method for authentication that is reliably able to discard unauthentic datagrams after a AES-256 decryption of a single block of data.

Another contribution we have made is the early-stopping modification to the GCM authenticated encryption algorithm. This is a way to combine pure GCM with CTR mode. Under pure GCM, the entire ciphertext is authenticated. It is also possible to authenticate additional data by using the additional authenticated data input. However, it is not possible to exclude parts of the ciphertext from being authenticated. This is what our modification allows. Lynxtun datagrams carry payload that are discarded upon being received. We still have to encrypt these payloads, so that the the entire datagram that is sent over the network is encrypted. However, by excluding the payload from the authentication allows us to perform authenticated decryption more efficiently. This approach is superior in comparison to using a separate pseudo-random number generator to generate random padding, which can exhibit different statistical properties than the actual ciphertext. It is also secure against padding-oracle attacks, as the only way to verify padding length is by knowing the secret key, which is used to generate the authentication tag. Our approach elegantly solves the problem of generating random padding without compromising the requirement that concatenating all captured datagrams appears to be statistically random data.

Our implementation section provides insight on how such a protocol can be implemented in practice in order to have deterministic runtime behavior. We have also talked about how we implemented our cryptographic algorithm and the underlying primitives.

The chapter on experimental analysis identifies certain challenges that come up when evaluating the security of a Lynxtun deployment, particularly with regards to data collection. Ultimately, assessing the actual security of a Lynxtun deployment requires observing its dispatch process. The discussion in this section provides insight on how this can be done. These ideas can also be used in assessing the security of TFC protocols in general. We have seen that, in the literature, simulation-based studies are often used. However, these fail to capture factors due to non-deterministic runtime behavior, and are therefore not suitable for assessing TFC countermeasures. The approach that we present in this section discusses how actual data collection can be done.

7.2 Operational Considerations

In this section, we provide a discussion of important operational considerations that arise when using Lynxtun in practice.

As we have stated from the outset, Lynxtun is not a general-purpose network protocol. As such, it is not a drop-in replacement for solutions like OpenVPN or IPsec. Lynxtun strives for perfect security, within a certain tolerance. This involves ensuring traffic flow confidentiality by maintaining a constant bitrate. This requires a fixed amount of bandwidth to be dedicated.

Deploying Lynxtun requires planning to be done beforehand. It is necessary to have a clear understanding of the communication that Lynxtun will be used to protect. There are two aspects to this. First, we need to know the statistical properties of the communication. Specifically, we need to know how much bandwidth we need to dedicate in order to sustain the communication. An important question to ask is whether the communication will come to an end after a fixed amount of time, or it will be allowed to continue indefinitely. When deciding how much bandwidth has to be dedicated, we have to consider the entire lifetime of the communication. Of course, we also have to consider whether or not we have sufficient network resources. This problem is more difficult as the amount of available network resources can also change over time. The planning has to account for all of these factors.

The second aspect is performing an analysis of the security requirements. It is necessary to understand what negative consequences will arise if confidentiality is compromised. As we have stated, this should always be evaluated in light of how actual people are affected. If there is a point in time after which confidentiality is no longer required, this should be understood. The analysis should also develop a threat model. That is, adversaries that could actively attempt to compromise security should be identified. If there are such adversaries, what are their capabilities and competencies? How much time, resources and effort are they expected to be willing in this task? Is there reason to believe that these adversaries may have preconceptions regarding the communication that allows them to devise particular hypothesis tests? These should all be determined.

It is helpful to present several example usecases in order to illustrate these concepts.

For our first scenario, consider a military command center. An engagement is being planned to be executed in one week. There are three bases along the front line: Alpha, Bravo and Charlie. The enemy is expecting an attack in the near future, but does not know the exact date. They also do not know from which of the three bases the attack will be launched. If they knew this, they could prepare their defences accordingly. As such, this information is highly confidential. It ceases to be confidential the moment that the enemy realizes that the operation is taking place. The HQ and the bases communicate over an isolated network, but enemy reconnaissance is able to capture packets as they are being sent along the links. Since the network is isolated, we know how much bandwidth is available at all times. The enemy has two preconceptions with regards to the communication. First, they assume that there will be more communication between the HQ and the base from which the attack will be launched. Second, they assume that there will be an increase in the amount of generated traffic leading up to the attack. Both of these are reasonable assumptions. We use Lynxtun to establish protect all communication between the HQ and each of the three bases[1]. We configure Lynxtun to use all available bandwidth on each of the three links, which we assume to be the same in all cases. This way, the amount of data that is observed on all three links is always the same.

In our second scenario, there are two people who wish to be able to exchange written messages over a secure channel. Specifically, they want to hide when messages are being exchanged. They create establish a Lynxtun tunnel with a dedicated bandwidth of several megabytes per day. This can be sustained indefinitely, and can be used to effectively carry text messages with sufficiently low latency. An observer will see that encrypted datagrams are being exchanged between the two parties, but will not be able to tell when actual communication is taking place.

In our third scenario, suppose that Alice is connecting to the internet from within a corporate network. Network traffic is analyzed at the periphery to detect which websites Alice visits. This involves statistical timing analysis that is able to detect restricted websites even if the traffic is being tunneled through SSH, for instance. Alice creates a Lynxtun tunnel to a host under her control that is outside this network, and uses it as her default network gateway. Her web browsing activity will last for one hour, which is the lifetime of the tunnel. She knows that a bandwidth of 1 Mbps will be sufficient. During this period, analysis at the network edge will only detect fully encrypted datagrams being exchanged between Alice and a single external host at a fixed rate.

[1] This example is for illustrative purposes only. We make no claims with regards to the suitability of Lynxtun in a military context.

It may be the case that the properties and therefore the requirements of the underlying communication change while the tunnel is being used. If the amount of communication decreases, this will mean that the tunnel utilization will become less. That is, the ratio of actual data to padding has become less, and it is now possible to dedicate less bandwidth in order to make communication possible. On the other hand, if the communication data rate increases, then the bandwidth specified by the tunnel configuration might no longer be sufficient. This would lead to queues filling up and packets being dropped. Both of these are related to the fact that the underlying communication may not be stationary. The important thing to realize here is that changing the tunnel bandwidth necessarily leads to an observable change on the network. If, for instance, we increased bandwidth whenever there is communication and decrease bandwidth when there is none in order to conserve network resources, then it is easy for an attacker to detect the times at which communication is taking place. This would defeat the purpose of a CBR approach. These considerations become more important in the case of tunnels that will remain active for a prolonged amount of time. Of course, there may be times where changing parameters is desirable. In such a case, the tunnel should be reestablished after manually modifying the configuration parameters. While it is conceivable to add automatic mechanisms that would adjust these parameters during tunnel operation, this would go against the design principles that we have described. The correct approach would depend on the situation. For instance, if the communication data rate fluctuates between 10 Kbps and 50 Kbps throughout the day, then setting the tunnel bandwidth to 50 Kbps would be appropriate. If the maximum data rate of the communication was 100 Kbps for one week, and then drops to 50 Kbps, and we also have reason to believe that it will not increase in the near future, then we can revise the parameters in order to conserve network parameters. As always, we have to be aware that we are making a change that will be publicly observable, and can be used to make inferences about the communication. In this example, an attacker might infer that the amount of communication has decreased.

We have earlier mentioned that running unrelated tasks on the same computer makes it more difficult to have a deterministic dispatch process, but can in fact be beneficial in terms of security. A similar argument can also be made to have non-confidential streams of communication being tunneled through a Lynxtun tunnel in addition to a confidential stream of communication. It is important to note that we do not desire deterministic dispatch for its own sake. It is only important insofar it relates to TFC. However, it should be noted that the more complicated a system becomes, the more difficult it is to reason about its long-term behavior.

7.3 Future Work

7.3.1 Kernel Implementation

The level of traffic flow confidentiality that can be achieved depends on how much control we have over the underlying hardware. The Linux kernel provides numerous features that allows us to be able to do this successfully, even in the context of a userspace application. However, greater control is possible by implementing Lynxtun as a kernel module. This would allow us to have direct control on issues such as scheduling, context switching and the network stack. It would also have performance benefits due to the fact that data does not have to be transferred back and forth between the kernel and userspace.

Kernel development is generally difficult as it debugging facilities are limited. Therefore, having a functional userspace implementation is highly beneficial to have as a starting point.

One of the key benefits of implementing Lynxtun in userspace is that it is simple to install and run. Installing and maintaining a custom kernel module, particularly one that is not in the mainline Linux kernel, is more involved. Due to the niche nature of Lynxtun, it is not suitable to consider it as a candidate for integration into the mainline kernel. On the other hand, we have seen that improving security beyond what is possible on a stock kernel requires a custom compiled kernel, which can also include applying the `RT_PREEMPT` kernel patch. We presume that a user that has requirements that justify such an approach would justify installing a custom kernel module.

7.3.2 Hardware Implemented Cryptographic Primitives

Modern processors feature hardware implementations of certain cryptographic primitives, such as AES-256. We have implemented AES-256 and GCM ourselves in software. There are performance benefits of using a hardware implementation, if it is available. This support can be added to our Lynxtun implementation.

7.3.3 Time Synchronization

Time synchronization between the two tunnel endpoints has implications on tunnel operation and security as discussed in Section 3.10.5. How this should be achieved has not been included in the scope of this thesis. However, due to the importance of the issue, it would be useful to add mechanisms to the Lynxtun Protocol that specify how time synchronization should be done in a secure manner. Specifically, time synchronization should be achieved when a key is used for the first time, and additional adjustments should be done to keep the two clocks synchronized without undermining the security, whilst keeping a correct representation of actual time.

7.3.3.1 Cryptanalysis of the GCM Early-Stopping Modification

We have modified the GCM algorithm to provide early-stopping. As with all cryptographic algorithms, extensive cryptanalysis study must be done in order to ensure that it is actually secure.

7.3.3.2 Further Experimentation

The security of Lynxtun, as it relates to the regularity of its dispatch process, can only be fully assessed by observing an actual deployment. Experimenting with Lynxtun running on different machines, across different types of networks, and being used to channel different types of communication will provide additional insight with regards to how successful Lynxtun can operate in different situations.

Additionally, more accurate results can be obtained by fully isolating machines that perform data collection, and ensuring that Lynxtun datagrams and packets belonging to the underlying communication do not traverse the same physical links.

7.3.4 IPv6 Support

As we have noted, our implementation of Lynxtun only supports IPv4 traffic. However, it can be easily extended to support IPv6 packets.

REFERENCES

[1] Lwn: On vsyscalls and the vdso. https://lwn.net/Articles/446528/.

[2] Lwn: The high-resolution timer api. https://lwn.net/Articles/167897/.

[3] man (7) pcap-tstamp. https://www.tcpdump.org/manpages/pcap-tstamp.7.html.

[4] Nmap security scanner. https://nmap.org.

[5] Openvpn. https://openvpn.net.

[6] Posix realtime extensions. http://pubs.opengroup.org/onlinepubs/7908799/xsh/realtime.html.

[7] Rt linux. https://rt.wiki.kernel.org/index.php/Main_Page.

[8] Softether vpn project. https://www.softether.org/.

[9] Two frequently used system calls are 77% slower on aws ec2. https://blog.packagecloud.io/eng/2017/03/08/system-calls-are-much-slower-on-ec2/.

[10] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid, P. D. Gallagher, et al. Nist special publication 800-57 recommendation for key management–part 1: General. 2012.

[11] T. Berger. Analysis of current vpn technologies. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 8–pp. IEEE, 2006.

[12] L. Bernaille, R. Teixeira, and K. Salamatian. Early application identification. In *Proceedings of the 2006 ACM CoNEXT conference*, page 6. ACM, 2006.

[13] P. Boucher, A. Shostack, and I. Goldberg. Freedom systems 2.0 architecture, 2000.

[14] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[15] M. A. Brown. Linux traffic control howto. `http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html`.

[16] X. Cai, R. Nithyanand, and R. Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 121–130. ACM, 2014.

[17] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a ssl/tls channel. In *Annual International Cryptology Conference*, pages 583–599. Springer, 2003.

[18] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[19] D. Cohen. On holy wars and a plea for peace, ien-137, 1980.

[20] J. Crichton, S. Shao, and J. Staten. Method and apparatus for tunneling tcp/ip over http and https, Nov. 20 2003. US Patent App. 10/152,281.

[21] J. Daemen and V. Rijmen. Aes proposal: Rijndael. 1999.

[22] T. Dierks and E. Rescorla. Ietf rfc 4346,“. *The Transport Layer Security (TLS) Protocol Version*, 1, 2006.

[23] W. Diffie and M. Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[24] J. A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *Proceedings of the 2017 Network and Distributed System Security Symposium, NDSS*, volume 17, 2017.

[25] M. J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. 2007.

[26] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 332–346. IEEE, 2012.

[27] D. C. Ferreira, F. I. Vázquez, G. Vormayr, M. Bachl, and T. Zseby. A meta-analysis approach for feature selection in network traffic research. In *Proceedings of the Reproducibility Workshop*, pages 17–20. ACM, 2017.

[28] S. Frankel and S. Krishnan. Ip security (ipsec) and internet key exchange (ike) document roadmap. rfc 6071 (informational). 2011.

[29] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206. ACM, 2002.

[30] S. Gianvecchio and H. Wang. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 307–316. ACM, 2007.

[31] B. Gladman. Input and output block conventions for aes encryption algorithms. *AES Round*, 2.

[32] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *2013 IEEE Symposium on Security and Privacy*, pages 65–79. IEEE, 2013.

[33] I. ISO. 7498-2: 1989. *Information processing systems-Open Systems Interconnection*, pages 7498–2.

[34] A. Johnson. *Design and analysis of efficient anonymous-communication protocols*. Yale University, 2009.

[35] C. Kiraly, S. Teofili, G. Bianchi, R. L. Cigno, M. Nardelli, and E. Delzeri. Traffic flow confidentiality in ipsec: Protocol and implementation. In *The Future of Identity in the Information Society*, pages 311–324. Springer, 2007.

[36] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. In *ACM SIG-*

COMM Computer Communication Review, volume 43, pages 303–314. ACM, 2013.

[37] M. Liberatore and B. N. Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 255–263. ACM, 2006.

[38] D. McGrew and J. Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20, 2004.

[39] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183–195. IEEE, 2005.

[40] T. T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[41] L. Nussbaum, P. Neyron, and O. Richard. On robust covert channels inside dns. In *IFIP International Information Security Conference*, pages 51–62. Springer, 2009.

[42] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2011.

[43] Y. Pang, S. Jin, S. Li, J. Li, and H. Ren. Openvpn traffic identification using traffic fingerprints and statistical characteristics. In *International Conference on Trustworthy Computing and Services*, pages 443–449. Springer, 2012.

[44] N. F. Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001.

[45] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Designing Privacy Enhancing Technologies*, pages 10–29. Springer, 2001.

[46] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998.

[47] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for qos: a statistical signature-based approach to ip traffic classification. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 135–148. ACM, 2004.

[48] C. Scott, P. Wolfe, and M. Erwin. *Virtual private networks.* " O'Reilly Media, Inc.", 1999.

[49] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th international conference on World Wide Web*, pages 512–521. ACM, 2004.

[50] C. Shanon. Weaver 1949 the mathematical theory of communication. *Urbana University of Dlionis Press.*

[51] M. Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[52] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001, 2001.

[53] P. Syverson, R. Dingledine, and N. Mathewson. Tor: The secondgeneration onion router. In *Usenix Security*, 2004.

[54] O. Titz. Why tcp over tcp is a bad idea. `http://sites.inka.de/bigred/devel/tcp-tcp.html`.

[55] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *USENIX Security Symposium*, pages 143–157, 2014.

[56] N. Williams and S. Zander. Evaluating machine learning algorithms for automated network application identification. 2006.

[57] C. V. Wright, S. E. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9, 2009.

[58] J. Yonan. The user-space vpn and openvpn. `https://openvpn.net/papers/BLUG-talk/index.html`.

[59] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao. On flow correlation attacks and countermeasures in mix networks. In *International Workshop on Privacy Enhancing Technologies*, pages 207–225. Springer, 2004.

[60] H. Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.

Appendix A

DATA DIRECTIONALITY NAMING CONVENTIONS

Descriptive words that are specify direction (inbound, outbound, ingress, egress) and actions (read, write, send, receive) are relative to a frame of reference. We use the following conventions.

We consider the outward direction to point toward the public network. That is, the network that is used to transmit Lynxtun datagrams. As for actions, the subject performing the action is always taken to be the Lynxtun endpoint.

The outward facing network interface is the egress. This can be an ethernet adapter, for instance. The inward facing interface is always the TUN device managed by the Lynxtun endpoint, called `lynx0` by default.

Various data flows in the system can be encoded using the following 3-bit encoding scheme.

$$(\text{Host ID}, \text{Action}, \text{Direction}) \tag{A.1}$$

The Host ID identifies the tunnel endpoint. It is defined such that the endpoint that has the smaller tunnel IP address (as interpreted as an unsigned 32-bit integer) is Host 0 and the other endpoint is Host 1.

Action is 0 for reading and 1 for writing. Here, the Lynxtun endpoint is the subject and either the TUN device (the ingress) or UDP/IP socket (attached to the egress) is the object.

Direction is 0 for outbound data and 1 for inbound data.

There are eight data flows in the system.

Flow ID	Description
000	Outbound IP packets read by Host 0 from the TUN device
001	Inbound Lynxtun Datagrams read by Host 0 from the UDP/IP socket
010	Outbound Lynxtun Datagrams written by Host 0 to the UDP/IP socket
011	Inbound IP packets written by Host 0 to the TUN device
100	Outbound IP packets read by Host 1 from the TUN device
101	Inbound Lynxtun Datagrams read by Host 1 from the UDP/IP socket
110	Outbound Lynxtun Datagrams written by Host 1 to the UDP/IP socket
111	Inbound IP packets written by Host 1 to the TUN device

Note that flows that are the bitwise complement of each other are related. Lynxtun datagrams received by Host 1 are those sent by Host 0. However, the flows in the inbound direction are subject to distortion due to network noise and interference. A datagram sent by Host 0 might not be received by Host 1, or an unrelated datagram might be received.

Appendix B

HARDWARE SPECIFICATION

The hardware specifications of the dedicated physical host that we use in our experiments is as follows.

Make and Model Acer Aspire One725 Netbook

Processor AMD Dual-Core C70 processor with Turbo Core technology up to 1.333 GHz.

Memory 2GB DDR3 Memory.

The system was always connected to a power supply.

Appendix C

KERNEL SPECIFICATION

The following Linux kernel configurations have been used in our experiments. All are variations of Linux 4.16.

lynx-0-nohz-rt A real-time kernel that uses the RT_PREEMPT kernel patch. Its preemption model is a fully preemptible kernel. It is compiled with `CONFIG_NO_HZ_FULL` to disable timer interrupts for cores with a single runnable task.

lynx-1-tick This is similar to lynx-0-nohz-rt, with the difference being that it uses a periodic timer interrupt with a fixed frequency of 1000 Hz.

lynx-2-desktop This kernel configuration represents a stock desktop kernel that is included with popular Linux distros. Its preemption model is Low-Latency Desktop. It uses dynamic clock ticks, with a frequency of 1000 Hz.

Appendix D

GCM FIELD MULTIPLICATION

The Galois Field $GF(p^n)$, where p is a prime, is a finite field with p^n elements. GCM uses $GF(2^{128})$. The elements of this field are 128-bit binary strings.

A finite field is defined by its multiplication and addition operations that obey the field axioms of commutativity, associativity and distributivity. Both operations map one field element to another field element. We use the $\cdot$ operator to mean field multiplication and $\oplus$ to mean field addition, when these are used in the context of field elements. Note that field addition is simply the XOR operation.

Let X and Y be arbitrary field elements. $X \cdot Y$ is performed by representing each element as a polynomial, multiplying these polynomials, then dividing the resulting polynomial with a special polynomial called the *field polynomial*. The result of the field multiplication is defined to be the result of this division, expressed as a field element.

The field polynomial used in GCM is:

$$f = 1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{128} \tag{D.1}$$

We use the notation X_i where i is from 0 to 127 to refer to the i th bit of a field element. X_0 is the leftmost bit.

Using this notation, we can show that the way to convert between field elements and corresponding polynomials is to use the following equivalence:

$$X = X_0 X_1 \ldots X_{127} \equiv X_0\alpha^0 + X_1\alpha^1 + \ldots + X_{127}\alpha^{127} \tag{D.2}$$

We define field element P that corresponds to the polynomial α. We have

$$P_i = \begin{cases} 1 & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{D.3}$$

Multiplication by polynomial α simply corresponds to a shift of indices.

Let χ be the polynomial that corresponds to field element X.

If $X_{127} = 0$, then the order of χ is less than 127. Then multiplying χ by α gives a polynomial with order less than 128. The remainder of dividing this polynomial by the field polynomial f (which is of order 128) is the χ itself. So whenever $X_{127} = 0$, $X \cdot P =$ rightshift(X). We define the rightshift operation such that rightshift$(X) = Y \iff Y_i = Xi - 1\ \forall i > 0$ and $Y_0 = 0$. Notice that involves shifting all the bits of X to the right by one position. The rightmost bit is discarded, and the leftmost bit becomes 0 after the shift. This operation is equivalent to integer division of a 128-bit unsigned integer by 2.

If $X_{127} = 1$, then the product of χ and α is a polynomial of degree 128. Let us call this $\alpha^{128} + a$. Then we have to find q and r such that $\alpha^{128} + a = q \cdot f + r$, and r is a polynomial of degree less then or equal to 128. Setting $q = 1$ gives us the solution

$$r = \alpha^{128} + a - f = a + 1 + \alpha + \alpha^2 + \alpha^7 \tag{D.4}$$

The α^{128} term cancels out with the same term in f, because addition done over $GF(2)$.

The polynomial $1 + \alpha + \alpha^2 + \alpha^7$ corresponds to a field element R with its leftmost bits set to 11100001 and the rest of the bits set to 0.

Notice also, that a corresponds to the field element rightshift(X).

Therefore, we can express $X \cdot P$ as,

$$X \cdot P = \begin{cases} \text{rightshift}(X) & \text{if } X_{127} = 0 \\ \text{rightshift}(X) \oplus R & \text{if } X_{127} = 1 \end{cases} \tag{D.5}$$

where R is the field element defined above.

Let I be the field element that corresponds to the polynomial 1. This is the identity element for field multiplication. While the original paper [38] does not explicitly mention this, defining $P^0 = I$. By doing this, we can say that $P^i = X$ such that $X_j = 1 \iff i = j$. Additionally, we say that the null element is the field element with all bits equal to zero. If we define a have a set that holds the null element and all the powers of P, it is possible to write all field elements as a sum of elements of this set.

This is precisely the approach that is used to multiply arbitrary field elements X and Y. Y is represented as a sum of powers of P, and then the distributive property is used to multiply X with these powers of P in the way we have outlined above, and then sum the individual products.

D.1 Multiplication of Arbitrary Field Elements

In order to multiply two arbitrary field elements X and Y, it is necessary to express Y as a sum of powers of P.

$$Y = c_0 P^0 + c_1 P^1 + \ldots + c_{127} P^1 27 \tag{D.6}$$

where c_i is either 1 or 0. Then,

$$X \cdot Y = X \cdot c_0 P^0 + X \cdot c_1 P^1 + \ldots + X \cdot c_{127} P^{127} \tag{D.7}$$

To calculate $X \cdot P^i$, we have to repeatedly multiply by P, a total of i times.

$$X \cdot P^i = X \cdot P \cdots P \tag{D.8}$$

This completes the procedure for multiplying arbitrary field elements.

D.2 Field Multiplication Using Lookup Tables

The method described here is based on the simple 8-bit decomposition method in [38]. In order to implement GCM, we do not require multiplication of arbitrary field elements. One of the operands of the field multiplication is always H, which is the hash key. The hash key is derived from the AES-256 key. We can generate lookup tables for a given hash key that will significantly increase the efficiency with which field multiplications operations are performed.

Let $X[i]$ be the i th byte of field element X.

For a byte x let $\rho(x)$ be a field element B such that $B[i] = x$ and the remaining 120 bits are all zero.

A field element X can be decomposed into 16 field elements $\rho(X[i]) \cdot P^{8i}$ where i is from 0 to 15 in the following way:

$$X = \bigoplus_{i=0}^{16} \rho(X[i]) \cdot P^{8i} \tag{D.9}$$

Note that P^{8i} means shifting to the right by i bytes in this situation.

Since $H \cdot X$ is linear in the bits of X over the field $GF(2)$, we have

$$H \cdot X = \bigoplus_{i=0}^{15} \rho(X[i]) \cdot H \cdot P^{8i} \tag{D.10}$$

We use 16 lookup tables: M_0 through M_{15}. Each lookup table corresponds to a separate byte in the decomposition above. Each lookup table has 256 rows, corresponding to the 256 values that $X[i]$ can take. Let $M_i[j]$ be the j th row of table i. Then $M_i[j] = \rho(j) \cdot H \cdot P^{8i}$.

Hence the multiplication $X \cdot H$ becomes

$$X \cdot H = \bigoplus_{i=0}^{15} M_i[X[i]] \tag{D.11}$$

Which is highly efficient as it only requires performing 16 lookups to get the correct row corresponding to each byte of X, XORing these rows.

Our lookup table generation strategy is optimized even tough we could have also used a lower performance solution, since it is only called during the initialization phase.

First, we use the following three rules to fill in all the rows of all lookup tables corresponding to a power of 2.

- $M_0[2^7] = H$, where H is the hash key.
- $M_i[2^k] = M_i[2^{k+1}] \cdot P \quad \forall\, i$ and $\forall\, k < 7$
- $M_i[2^7] = M_{i-1}[2^0] \cdot P \quad \forall\, i > 0$

Doing this requires a field multiplication of an arbitrary field element with P to be done a total of 127 times.

$M_i[0]$ is the null field element for all i. These rows are zeroed out.

We represent the remaining rows as sums of powers of P, which have already been calculated. For example, $M_0[2^7 + 2^5 + 2^1]$ can be expressed as the sum $M_0[2^7] \oplus M_0[2^5] \oplus M_0[2^1]$.

This completes our construction of the lookup tables.

Appendix E

DATA GENERATOR EXPERIMENT RESULTS

E.1 Configuration Specifications

E.1.1 Configuration 1

Dispatch Interval 25 ms

Payload Size 5000 Bytes

Kernel `lynx-0-nohz-rt`

CPU Isolation Enabled

Realtime Scheduling Enabled

E.1.2 Configuration 2

Dispatch Interval 25 ms

Payload Size 5000 Bytes

Kernel `lynx-0-nohz-rt`

CPU Isolation Disabled

Realtime Scheduling Enabled

E.1.3 Configuration 3

Dispatch Interval 25 ms

Payload Size 5000 Bytes

Kernel `lynx-0-nohz-rt`

CPU Isolation Enabled

Realtime Scheduling Disabled

E.1.4 Configuration 4

Dispatch Interval 25 ms

Payload Size 5000 Bytes

Kernel `lynx-2-desktop`

CPU Isolation Enabled

Realtime Scheduling Enabled

E.2 Results

For each configuration, we present three plots:

Internal Dispatch Variability The standard deviation taken over 10 seconds, of the difference between the measured and expected delay of dispatch events, as recorded within the Lynxtun process.

Internal Dispatch Delay The measured delay of dispatch events, as recorded within the Lynxtun process. The mean should be the dispatch interval.

External Dispatch Delay The measured delay between consecutive Lynxtun datagrams, as recorded by a separate physical host on the network using tcpdump.

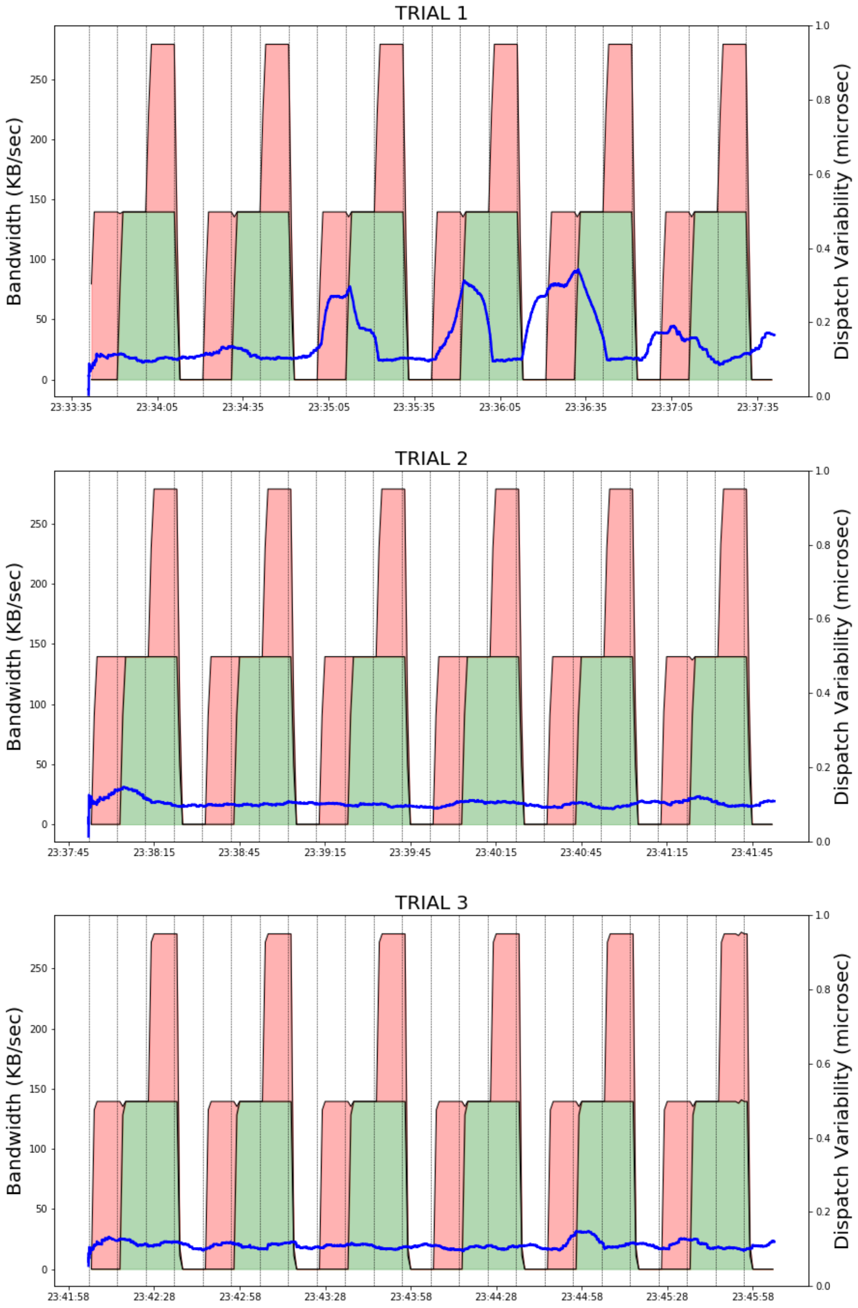


Figure E.1: Cfg 1 Internal Dispatch Variability

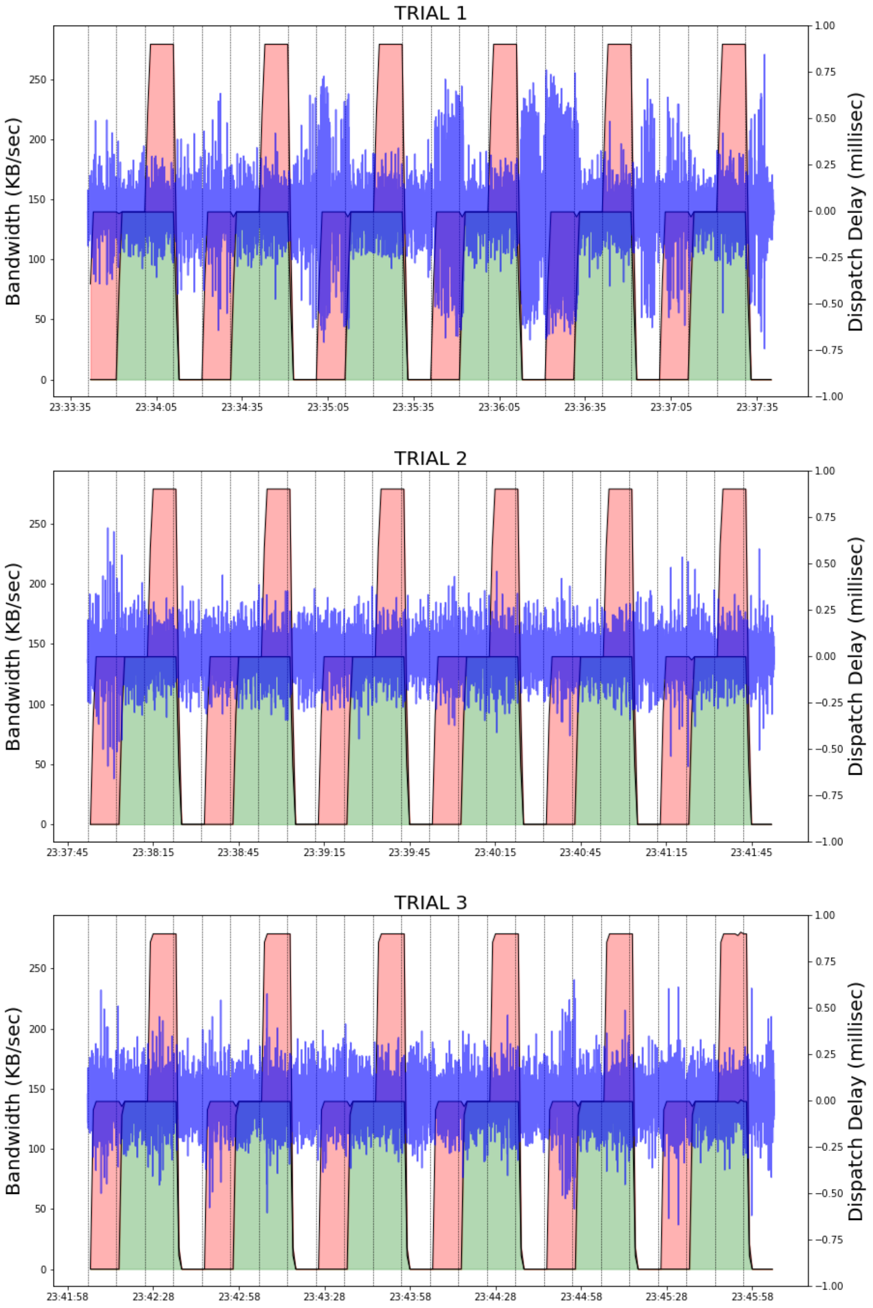


Figure E.2: Cfg 1 Internal Dispatch Delay

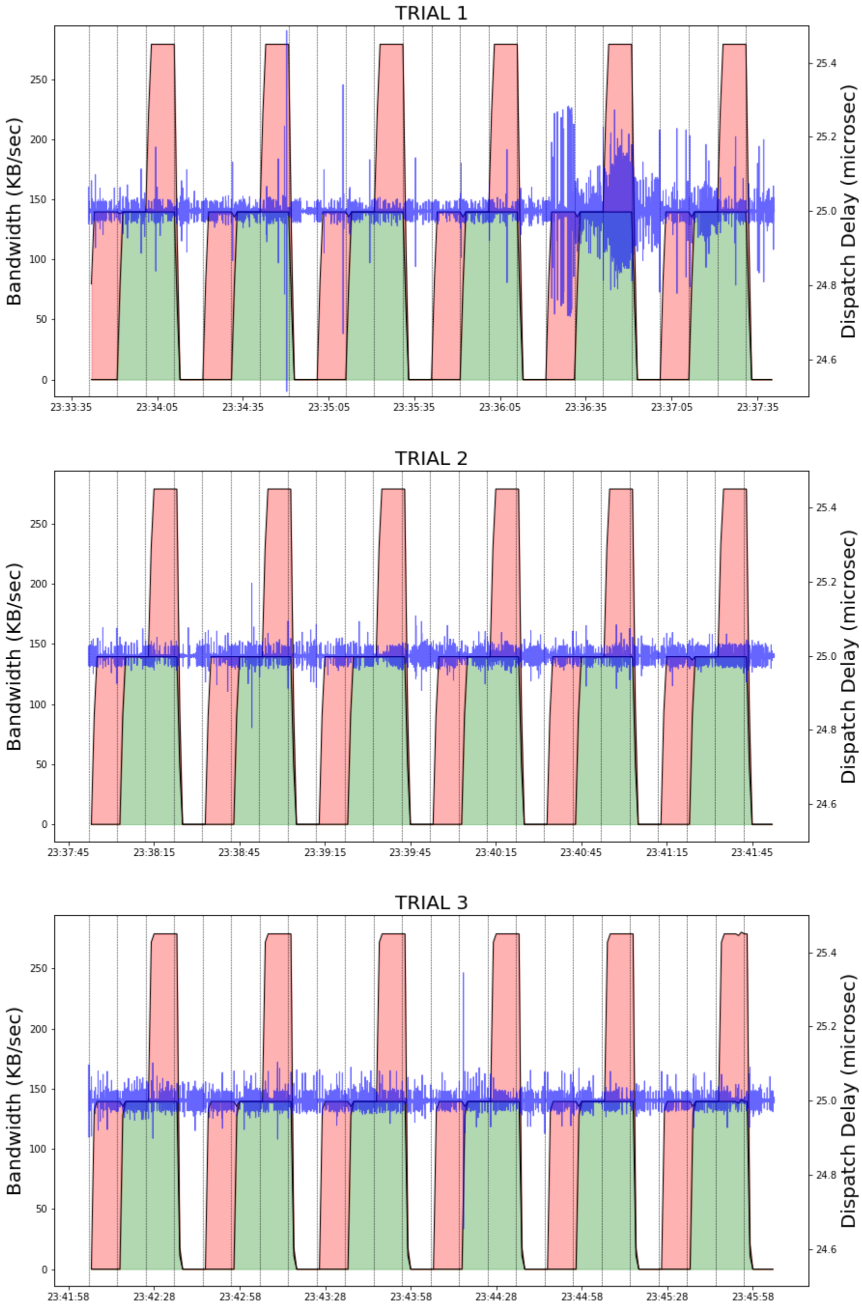


Figure E.3: Cfg 1 External Dispatch Delay

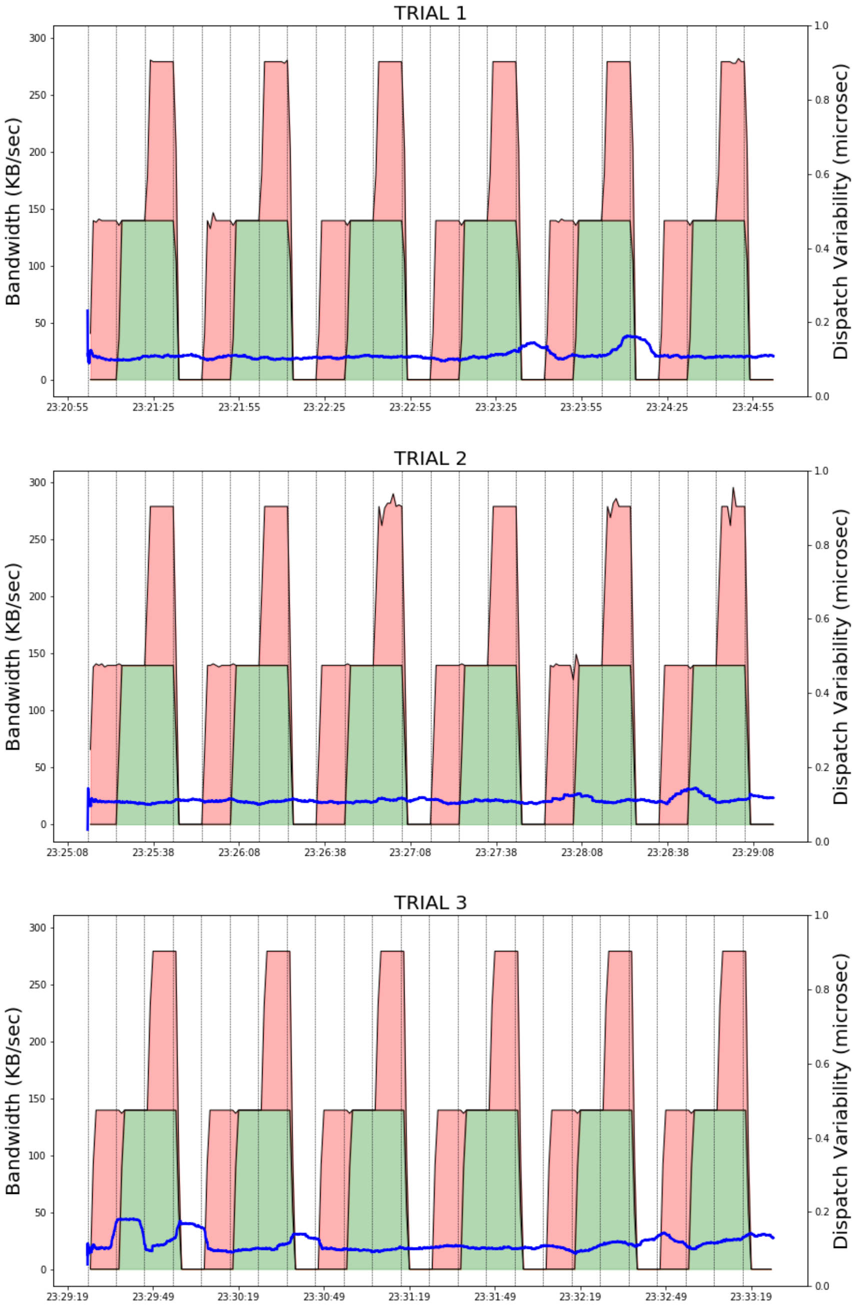


Figure E.4: Cfg 2 Internal Dispatch Variability

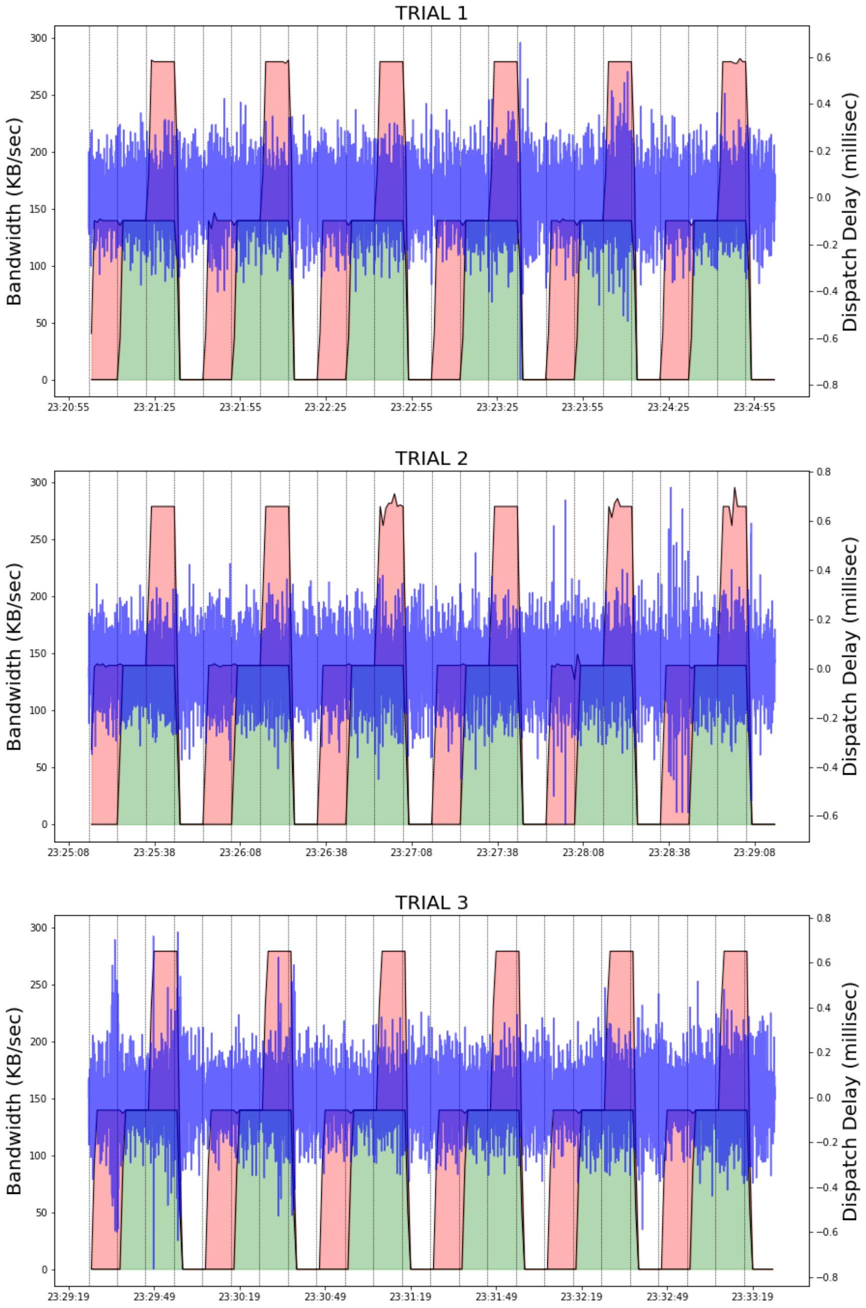


Figure E.5: Cfg 2 Internal Dispatch Delay

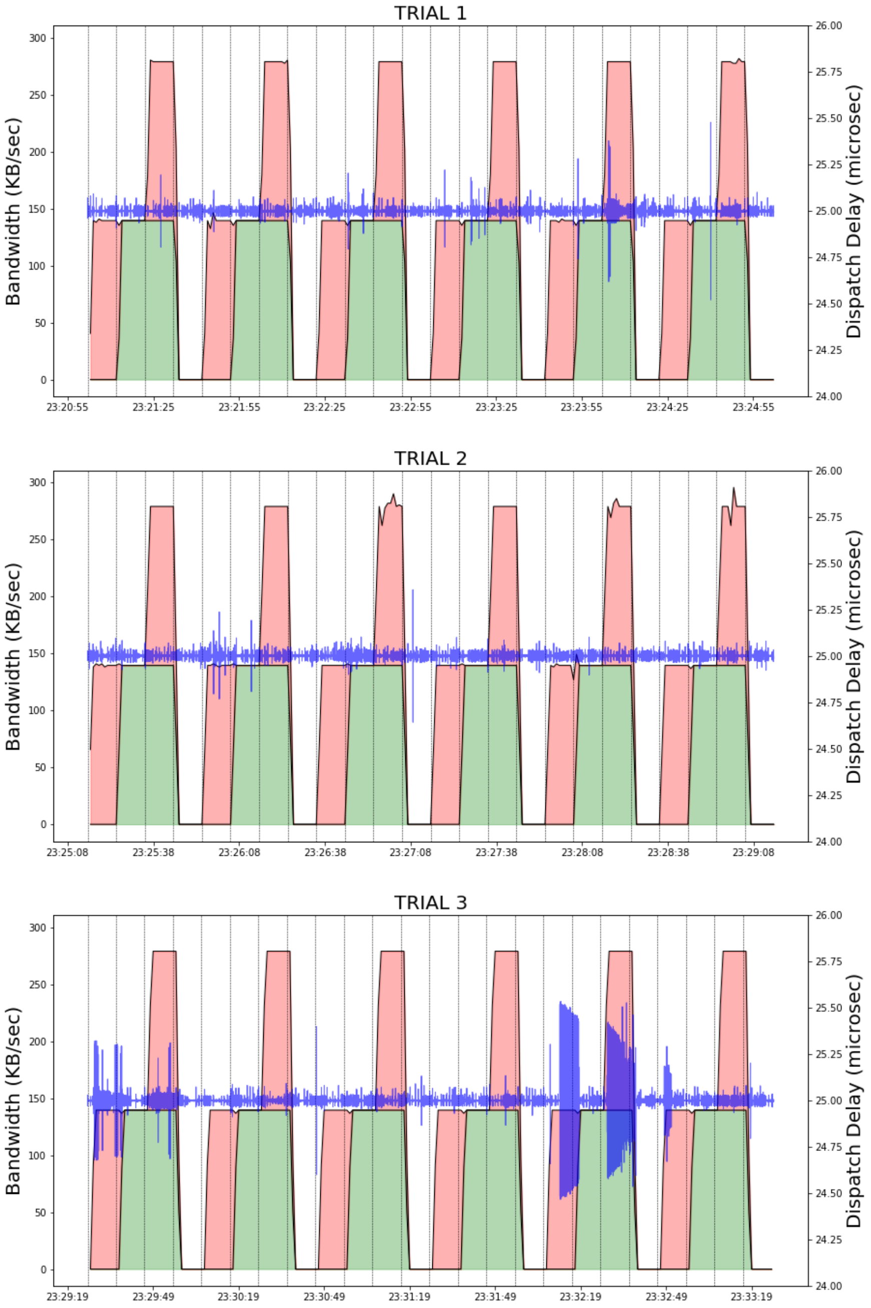


Figure E.6: Cfg 2 External Dispatch Delay

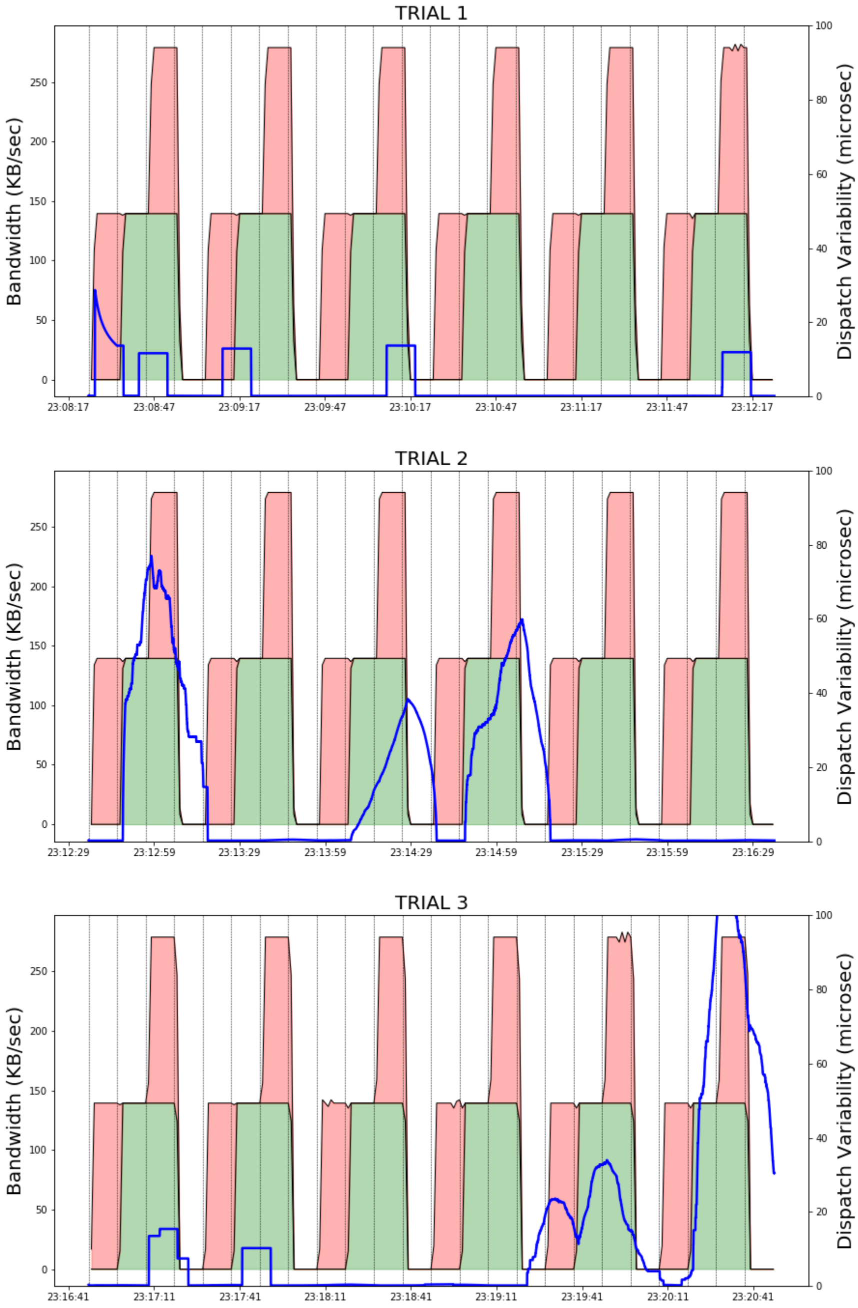


Figure E.7: Cfg 3 Internal Dispatch Variability

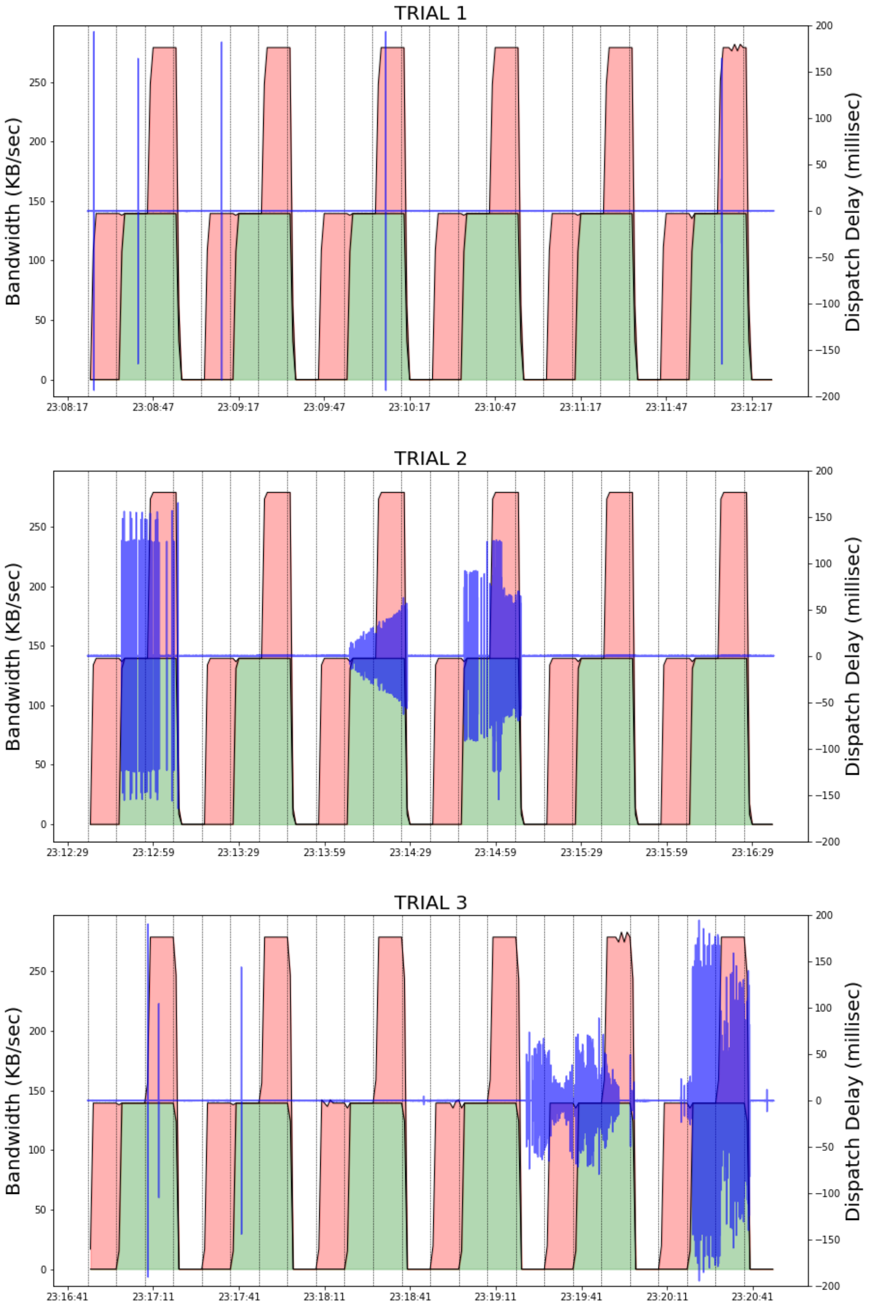


Figure E.8: Cfg 3 Internal Dispatch Delay

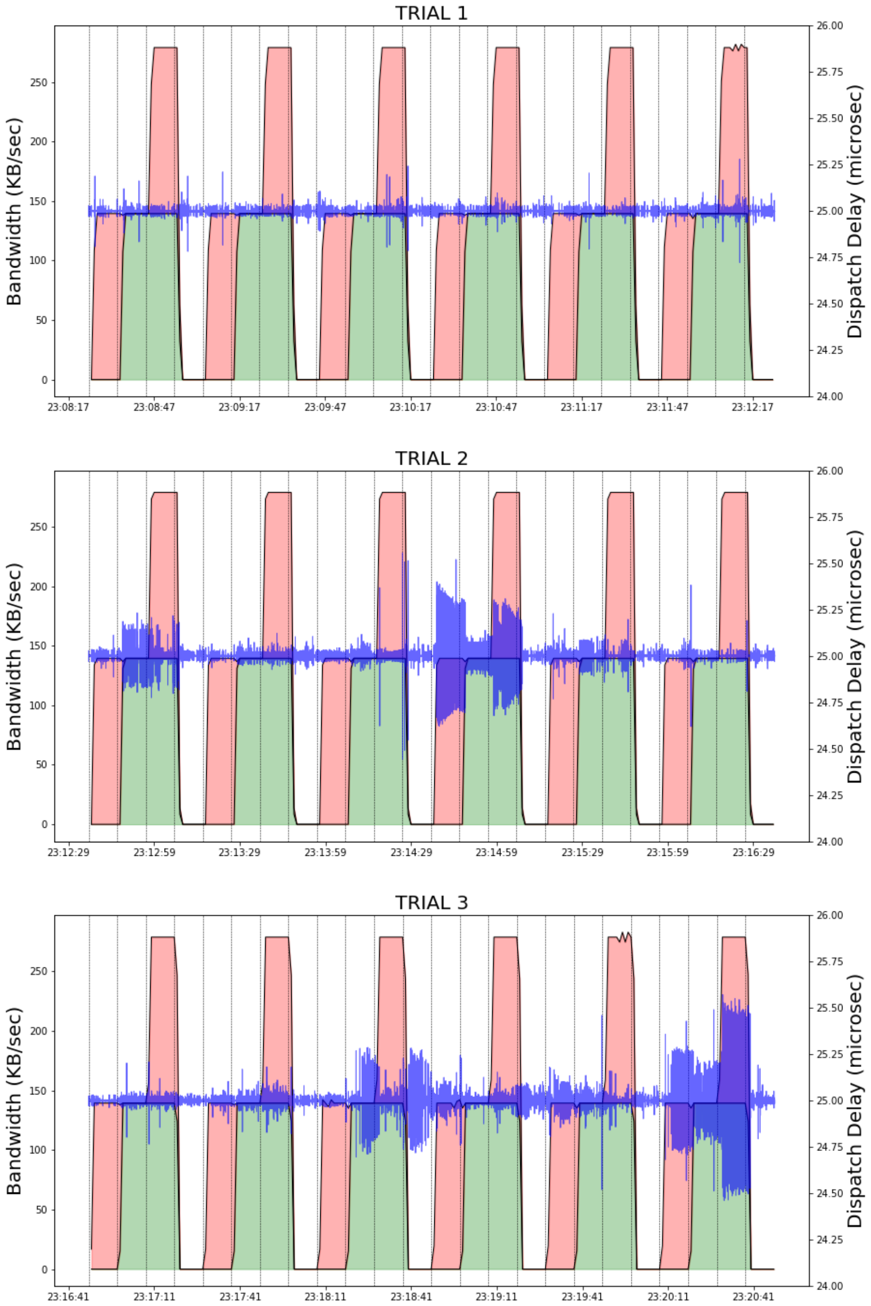


Figure E.9: Cfg 3 External Dispatch Delay

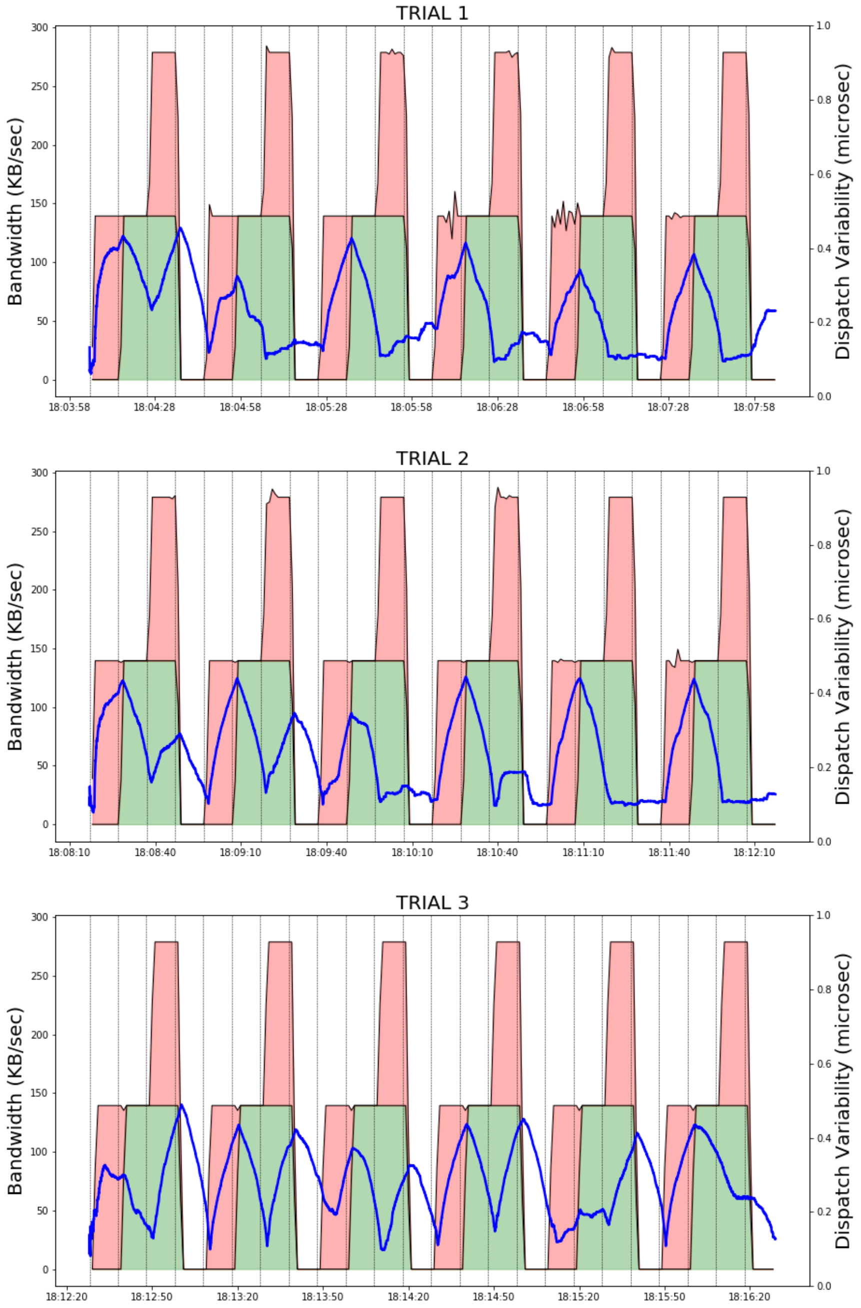


Figure E.10: Cfg 4 Internal Dispatch Variability

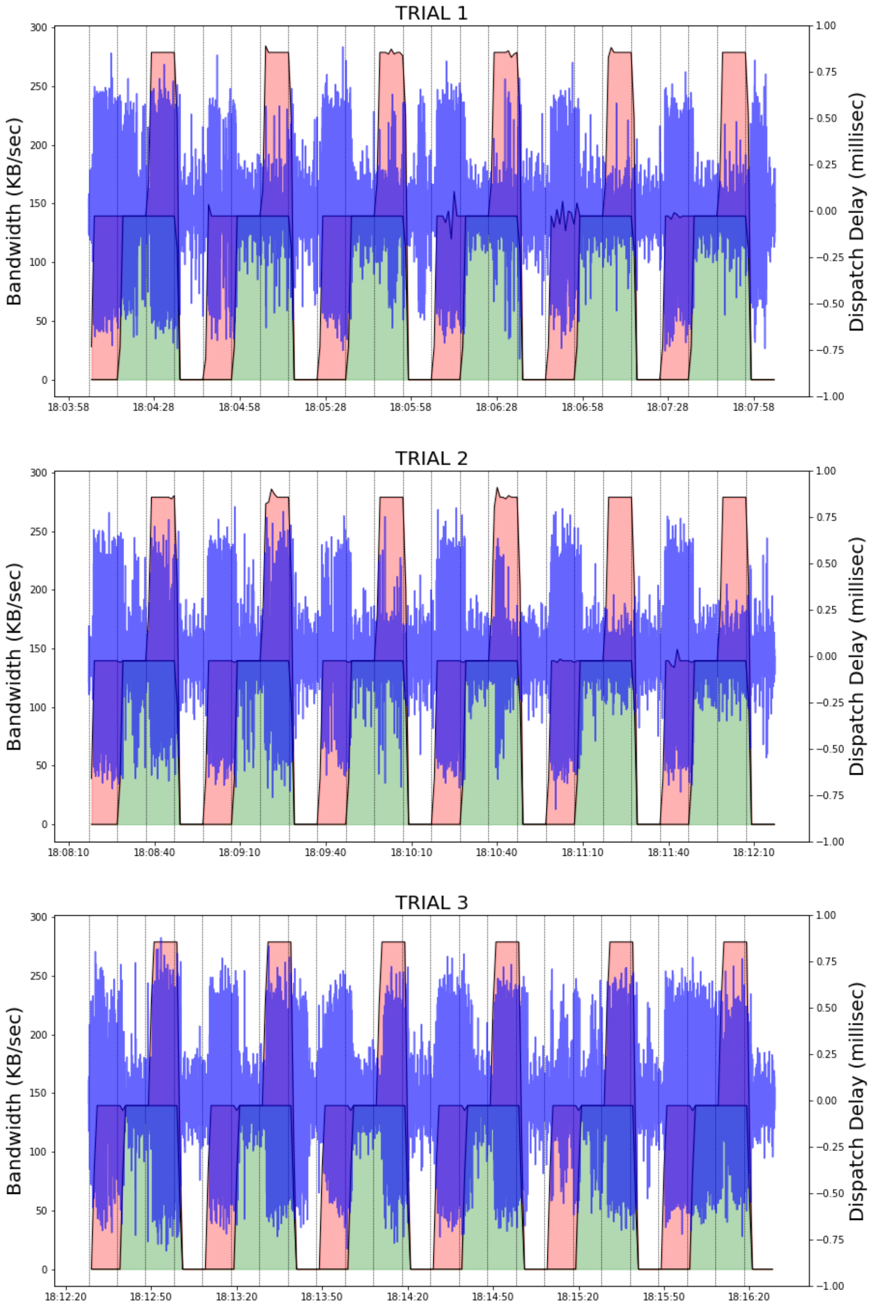


Figure E.11: Cfg 4 Internal Dispatch Delay

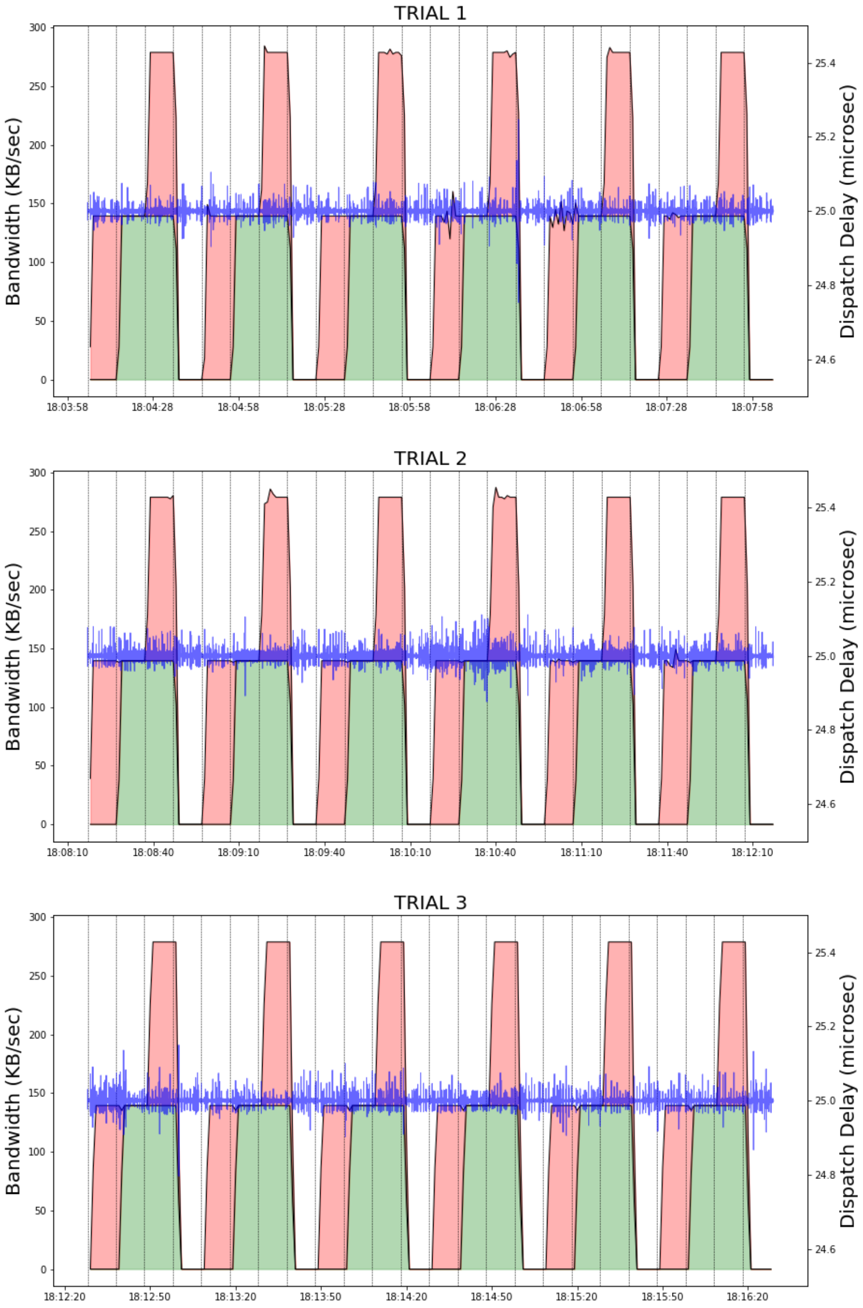


Figure E.12: Cfg 4 External Dispatch Delay

Appendix F

REALTIME EXPERIMENT IMPLEMENTATION

```
#define _GNU_SOURCE

#include <sched.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <lynx.h>

#define TSLEEP 20 /* msec */
#define SAMPLES 250

#define abs(X) ((X) >= 0 ? (X) : -(X))

int
main (int argc, char **argv)
{

    int spin = 0;
    if (argc > 1)
spin = atoi(argv[1]);

    if (argc > 2) {
int rt_prio = atoi(argv[2]);
if (rt_prio) {
    const struct sched_param sp = {
.sched_priority = 98,
    };
    if (sched_setscheduler(getpid(), SCHED_FIFO, &sp) == -1)
return 1;
}
    }

    if (argc > 3) {
int cpu_id = atoi(argv[3]);
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(cpu_id, &set);
if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
    return 1;
    }
    Time t_start, t_end, t_now, t_target;

    const Duration delta = msec_to_duration(TSLEEP);
    const long delta_nsec = duration_to_nsec(delta);

    Duration delta_ob;
    long delta_ob_nsec;

    struct timespec cur, tar;

    for (int i=0; i<SAMPLES; i++) {

     clock_gettime(CLOCK_REALTIME, &t_start);
t_target = time_add(t_start, delta);

if (spin) {
    while(get_nsec_until_time(t_target) > 0);
} else {
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t_target, NULL);
}

clock_gettime(CLOCK_REALTIME, &t_end);

delta_ob = time_sub(t_end, t_start);
delta_ob_nsec = duration_to_nsec(delta_ob);

printf("%ld\n", abs(delta_nsec - delta_ob_nsec));

    }

    return 0;

}
```