August 2020

Traffic Analysis Resistant Infrastructure

Jonathan Oakley
Clemson University, oakl.jon@gmail.com

Traffic Analysis Resistant Infrastructure

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Jonathan Oakley
August 2020

Accepted by:
Dr. Richard Brooks, Committee Chair
Dr. James Martin
Dr. Harlan Russell
Dr. Kuang-Ching Wang

Abstract

Network traffic analysis is using metadata to infer information from traffic flows. Network traffic flows are the tuple of source IP, source port, destination IP, and destination port. Additional information is derived from packet length, flow size, interpacket delay, Ja3 signature, and IP header options. Even connections using TLS leak site name and cipher suite to observers. This metadata can profile groups of users or individual behaviors.

Statistical properties yield even more information. The hidden Markov model can track the state of protocols where each state transition results in an observation. Format Transforming Encryption (FTE) encodes data as the payload of another protocol. The emulated protocol is called the *host protocol*. Observation-based FTE is a particular case of FTE that uses real observations from the *host protocol* for the transformation. By communicating using a shared dictionary according to the predefined protocol, it can difficult to detect anomalous traffic.

Combining observation-based FTEs with hidden Markov models (HMMs) emulates every aspect of a host protocol. Ideal host protocols would cause significant collateral damage if blocked (*protected*) and do not contain dynamic handshakes or states (*static*). We use *protected static* protocols with the Protocol Proxy–a proxy that defines the syntax of a protocol using an observation-based FTE and transforms data to payloads with actual field values. The Protocol Proxy massages the outgoing packet's interpacket delay to match the host protocol using an HMM. The HMM ensure the outgoing traffic is statistically equivalent to the host protocol. The Protocol Proxy is a covert channel, a method of communication with a low probability of detection (LPD). These covert channels trade-off throughput for LPD.

The multipath TCP (mpTCP) Linux kernel module splits a TCP streams across multiple interfaces. Two potential architectures involve splitting a covert channel across several interfaces (multipath) or splitting a single TCP stream across multiple covert channels (multisession). Splitting

a covert channel across multiple interfaces leads to higher throughput but is classified as mpTCP traffic. Splitting a TCP flow across multiple covert channels is not as performant as the previous case, but it provides added obfuscation and resiliency. Each covert channel is independent of the others, and a channel failure is recoverable.

The multipath and multisession frameworks provide independently address the issues associated with covert channels. Each tool addresses a challenge. The Protocol Proxy provides anonymity in a setting were detection could have critical consequences. The mpTCP kernel module offers an architecture that increases throughput despite the channel's low-bandwidth restrictions. Fusing these architectures improves the goodput of the Protocol Proxy without sacrificing the low probability of detection.

Acknowledgments

I would like to thank my adviser, Dr. Richard Brooks, who has helped me see problems in a different light. His unique insights have helped me grow as a scholar and a person. To my committee, Dr. Harlan Russell, Dr. Jim Martin, and Dr. Kuang-Ching Wang, thank you for your indispensable guidance and support. Each of you have shown me a different way of solving problems, and that has been invaluable to my studies. I am grateful for my honorary adviser, Dr. Lu Yu, who provided help, guidance, and countless revisions. I would also like to thank my research group, for their continued support–I have enjoyed working with all of you. Thanks to the Department of Education and the Graduate Assistants in Areas of National Need program, which allowed me to freely pursue my research interests while fostering my love for teaching. Finally, I would like to extend my gratitude to everyone who supported me along the way. This work was supported by the National Science Foundation grants CNS-1049765, OAC-1547245, and CNS-1544910.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traffic analysis is the process of using metadata to infer the purpose of network traffic, even if it is encrypted. There are several applications for network traffic analysis–security, data mining, and censorship. Every enterprise network has some form of traffic analysis to ensure the network is secure and protect critical systems. Most Internet Service Providers (ISPs) participate in data mining [55]. They use traffic analysis techniques to profile their customers, and they sell this information to other companies for a profit. The final use-case is censorship. Authoritarian regimes use traffic analysis to find citizens posting sensitive content that differs from their preferred reality [63].

1.1 Motivations

Traffic analysis is necessary for enterprise networks. According to a recent report, 2.6 billion internet users live in a country where people were imprisoned for posting controversial content [63]. More alarmingly, 38 countries bought telecommunications surveillance equipment from China [63]. While internet access is a fundamental human right, according to the United Nations [31], the internet is not a safe place for all users. There are many areas where traffic analysis resistant infrastructure is needed, but we examine two key areas:

1. Individual users who are operating in a contested network.

2. Users who wish to protect their personal information from ISPs.

By examining these pragmatic cases, we can provide a theoretical framework for addressing the challenge of ensuring unfettered access to the internet.

1.2 Organizations

- Chapter 2 provides an overview of traffic analysis, covert communications, and network-based moving target defense. We examine effective traffic analysis methods and compare them with recent developments in covert communications. Network-based moving target defense is a new subset of covert communication that focuses on randomizing host communication to foil traffic analysis.
- Chapter 3 describes the mathematical background behind the methods we present. This work builds on hidden Markov models (HMMs), and we discuss the basic principles behind inferring and comparing HMMs. We also discuss format transforming encryption (FTE), which encodes information in a medium that appears benign in most traffic analysis.
- Chapter 4 discusses our novel observation-based FTE, and our experimental results when using this encryption for various network activities.
- Chapter 5 details an approach for increasing the throughput of covert communication channels.
- Chapter 6 provides a succinct summary of the contributions and application of these technologies and discusses future work and open research challenges.

Chapter 2

Background

Traffic analysis is the process of using metadata to infer the purpose of network traffic, even if it is encrypted. Figure 2.1 shows examples of this metadata. Traffic flows are the four-tuple of source IP, source port, destination IP, and destination port. Metadata includes packet size, interpacket delay, flow duration, flow size, ja3 fingerprint, and various packet header options. Various tools aggregate this metadata and provide analysts with summaries that help identify anomalies. Other tools attempt to detect anomalies automatically from high volumes of traffic.

2.1 Traffic Analysis

Network traffic analysis is the process of using metadata to infer latent information from traffic flows. More specifically, it is the process of analyzing network communications to determine patterns, fingerprints, and properties that will aid in securing and optimizing the network. There are many tools for network traffic analysis, such as NetFlow [2], Zeek [19], Suricata [15], Suricata [15], Moloch [7], Wireshark [22], and tcpdump [16]. In this range of software, NetFlow is the most log-centric, and tcpdump is the most PCAP centric. NetFlow only samples several packet metadata fields and is designed to handle large volumes of traffic. Zeek takes this a step further and provides protocol analysis and an in-depth analysis of all packet metadata. Suricata and Snort focus on signature-based detection. Any incoming packet that matches a rule will trigger an alert. Moloch's primary focus is PCAP memorialization. Packet metadata is extracted and stored locally to facilitate searching large volumes of PCAP quickly. Wireshark provides protocol analysis but does not focus on processing large volumes of PCAP or storing packet metadata. On the other end of the spectrum, tcpdump's main focus is packet capture with minimal protocol analysis. Software focused on PCAP analysis (Wireshark, Moloch, or Zeek) should be used to view tcpdump's packet captures.

Nation-states often employ several of these tools (or their commercial equivalents) to provide a full range of coverage. A comprehensive network surveillance system would include NetFlow for network engineers, Zeek for detailed log generation, Suricata or Snort for in-line PCAP-based alerting, and Moloch for PCAP memorialization. The most robust covert channels must withstand scrutiny at each of these levels.

2.2 Circumventing Traffic Analysis

The simplest way to avoid traffic analysis is to use a single-hop proxy to encrypt traffic and mask the real destination. Psiphon [13] and Lantern [6] are such solutions that fill this niche. While proxies usually only deal with web browsing traffic, VPNs encrypt all traffic, potentially saving users from side-channel information leakage [80, 46]. As [24] and [80] found, these solutions are imperfect. There are also solution specific issues–Lantern is only active when a website is blocked [64], leading to a myriad of potential attacks. In practice, VPN companies must choose between turning over logs or facing federal charges [70]. In all of these cases, the users' privacy is in the hands of their chosen solution. Additionally, these solutions are easily detected with IP blacklist or PCAP-based rules to detect VPNs.

Since proxies and VPNs fail to provide sufficient privacy in several cases, anonymity networks like Tor [17] and I2P [5] have arisen. Tor's Onion Routing encrypts traffic at least three times, letting only the current node know the next destination. I2P is not widely used, despite being similar to Tor in many ways. There have been proof-of-concept attacks against the anonymity of Tor users [32], [23]. In order to combat attacks on anonymity and PCAP-based detections, Tor employs pluggable transports, which are modular proxies that mask the underlying protocol. For example, Marionette [35] has configurable ciphertext formats, protocol features, and statistical properties.

While both Tor and I2P are client-centric approaches, there have also been developments surrounding infrastructure–TapDance (Decoy Routing) employs control sequences embedded in innocuous traffic streams to reroute traffic to an undisclosed location [75]. While this proves to be a significant advance over the existing technology, TapDance flows must route through a TapDance

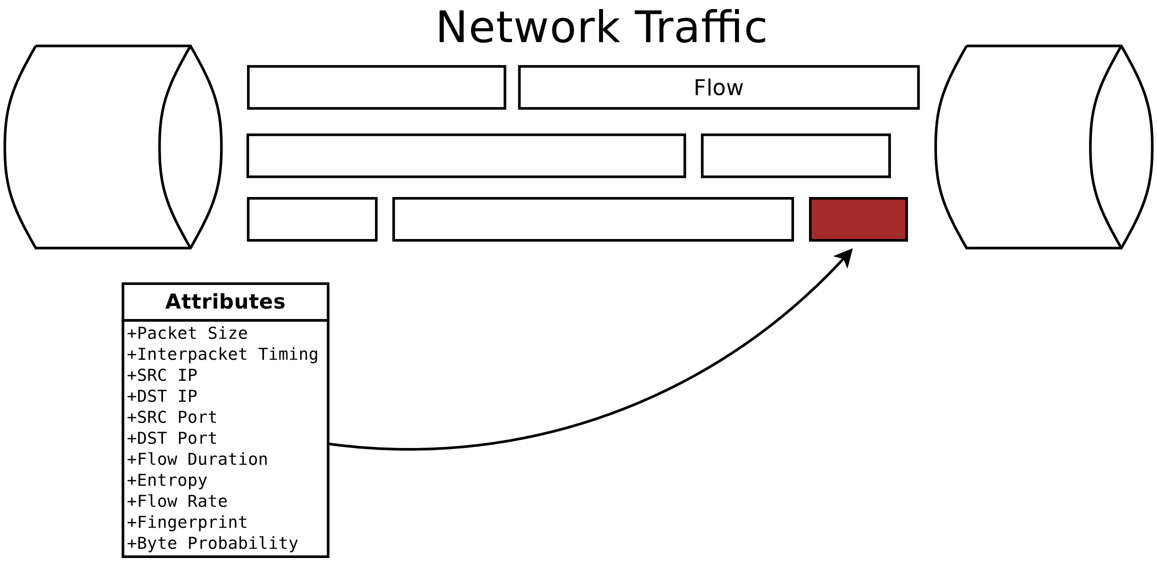

Figure 2.1: Traffic analysis using metadata.

node, which is not guaranteed.

2.3 Convert Communications [1]

Creating covert online communication tools has been the focus of many privacy advocacy groups. Since the data in covert channels is encrypted, the goal is to balance the probability of detection with throughput based on the desired application [65]. Timing side-channels prioritize low-probability of detection over throughput [45, 76]. However, this is not appealing to users who may prefer jail time over a slow connection [56].

Tor's anonymity network wraps network traffic in layers of encryption. Each layer can only be decrypted by the next hop in the *onion* network. While it provides anonymous access to the Internet, the Tor protocol is easy to detect and block [72, 32]. Undetectable communication was not one of Tor's goals, and as previously mentioned, it spawned the Pluggable Transport project to address this challenge and encourage the development of other covert communication tools [44].

GNUnet [4], I2P, and Freenet [3] all seek to provide anonymous access to the Internet. GNUnet is a toolbox for developing secure decentralized applications. I2P uses a Tor-like onion-based routing protocol to route traffic securely but was intended to be a self-contained network.

[1] This section has been adapted from [54].

Freenet focuses on using prior knowledge to form connections. It arguably provides more anonymity, but it is resource-intensive. Many governments block access to these tools, which makes the first-hop important.

Pluggable Transports (PTs) [12] address this concern. PTs offer a generic way to obfuscate traffic. *Shape-shifting* PTs transform traffic into a different protocol. SkypeMorph [50] makes network traffic resemble a Skype session. StegoTorus [69] demultiplexes connections to avoid traffic analysis and uses steganography to hide information in different protocols (including Skype). In *The parrot is dead: Observing unobservable network communications*, Houmansadr et al. [43] found both approaches fell short of true protocol mimicry. In both cases, handshake packets were incorrect. StegoTorus's implementation of HTTP steganography contained other flaws [43]. Censorspoofer mimics the Ekiga VoIP software, but it also falls short of mimicking protocol intricacies [43].

A number of PTs *scramble* traffic to remove fingerprints. Obfs2 [11], Obfs3 [11], Obfs4 [10], and ScrambleSuite [73] each attempt to remove a network fingerprint by *scrambling* the data. Dust2 and its previous version (Dust) change the statistical properties of traffic to bypass firewalls [71]. With technologies like software-defined networking (SDN), an adaptive firewall will block these statistical PTs.

Recent PTs use *domain fronting.* Traffic is sent to a known-good destination (Google, Amazon, Azure, CloudFront) and allowed through the firewall because blocking an entire domain would cause unintended collateral damage. FlashProxy [51], SnowFlake[14], and meek [37] all use variations of *domain fronting.* The companies that own the domains do not condone this practice because of potential backlash.

As previously discussed, Refraction Networking (TapDance) [75] spoofs the destination IP address. *If* the packet is routed through a decoy router, the real destination IP address is substituted for the spoofed address. Recent work has shown it may be inexpensive to censor decoy routers [60]. Alternatively, TARN [79, 68, 29, 66] provides an approach that mixes traffic from different autonomous systems at the software-defined exchange (SDX) level. This SDX-based approach provides a high level of anonymity and is resistant to a malicious ISP or BGP injection, but it is not realistic for a covert channel. Network-based moving target defense solutions have also been proposed [42] for covert channels.

Traditional Virtual Private Networks (VPNs) are not usually effective in a contested environment because encrypted data can indicate malicious activity [27]. As a result, Psiphon [13], Lantern [6], and Ultrasurf [18] have started using PTs. With Lantern, traffic is only forwarded through the PT if it is likely to be blocked.

FTE PTs are a subset of *shape-shifting* PTs that steganographically encode traffic using values typical of the host protocol [34]. It is best to use a widely adopted protocol, such as DHCP [58] or VoIP [59]. Marionette [35] is a *shape-shifting* PT that uses a probabilistic context-free grammar (PCFG) and production rules to mimic the host protocol. The PCFG ensures the traffic is syntactically and semantically correct, and the production rules occur at the expected frequency. Marionette ensures interpacket timing, packet size, and session count mimic the host protocol.

Image steganography is an effective means of covert communication. Fridrich investigated the relationship between distortion and information capacity [38]. Unfortunately, the model derived in [38] does not directly apply to FTE-based covert channels.

2.4 Bandwidth Limitations

Covert channels trade-off throughput for a low probability of detection. Traditionally, low throughput covert channels are tough to detect. In the observation-based FTE example, simpler protocols are much less likely to be detected, since there are no complex handshakes and limited fields. According to [56], users are less likely to use tools with higher latency, *even if they are aware of the security implications.*

Consider using observation-based FTE with the Network Time Protocol (NTP) [9], which is widely used, and blocking it would cause serious collateral damage. An example NTP packet is shown below in Figure 2.2. The client sends a request with only the transmit timestamp set, and the reply contains the same transmit timestamp, the receive timestamp, the originate timestamp, and the reference timestamp.

The timestamp fields are not conducive to replaying observed values since the observed timestamps will all be significantly outdated. In practice, this is still hard to detect due to the diverse nature of the Internet–many devices have internal clock drift, and their timestamps are wrong, and some NTP services (like chrony) use random timestamps [1]. However, it would be desirable to encode data in clock drift. When masquerading as a client, only the transmit timestamp can be used to encode data, and it needs to be relatively close to the actual time. Similarly, the server only encodes information in the receive, originate, and reference timestamp. A conservative channel

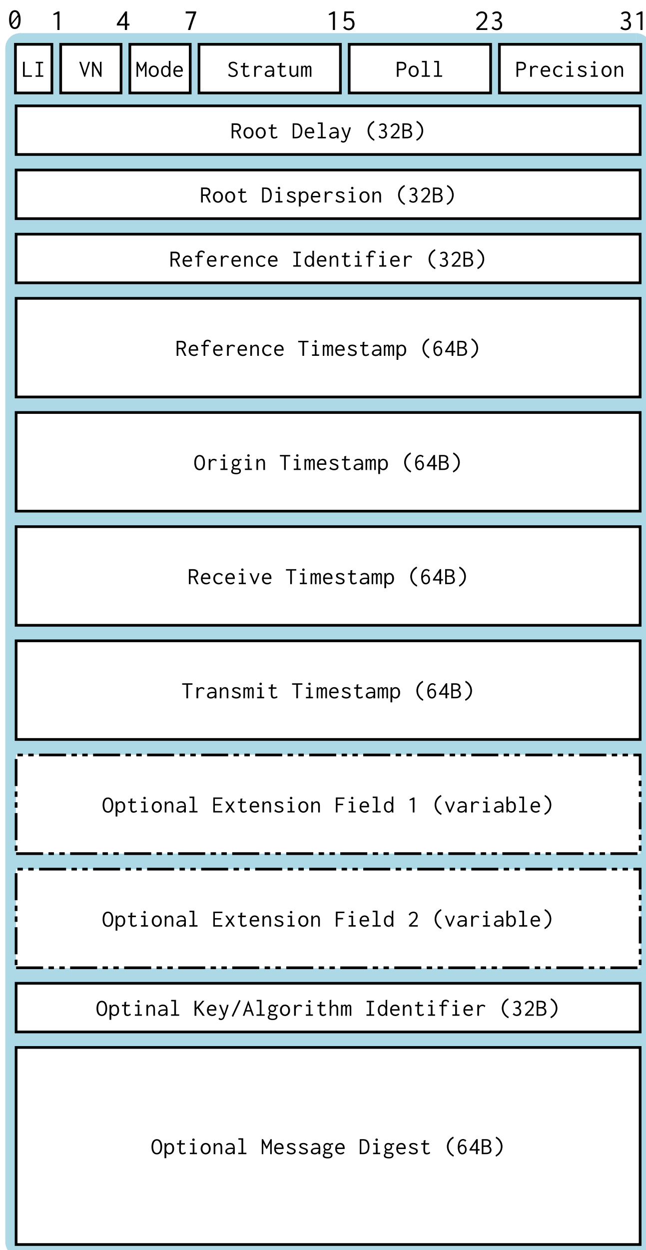

Figure 2.2: The structure of an NTP packet.

would only encode data in the fractional part of the timestamp (32 bits) and use the actual seconds.

With this toy channel, a client could encode 32 bits of information with every request and would expect 96 bits in every reply. Additional bits may be encoded in other fields, but these fields would hold the vast majority of the information. Requests occur roughly every three seconds, making the effective throughput of information approximately 10.6 bps upstream, and 32 bps downstream. This throughput does not include the overhead described in the previous section.

There are two approaches to this challenge. Both approaches employ a Linux kernel module that splits TCP flow across all mpTCP-enabled interfaces–mpTCP [8]. One approach is to split the covert channel into separate TCP streams. The second approach is to use multiple covert channels. Both are useful in particular cases, but for this example, consider the latter. When splitting a single TCP stream across N covert channels, the throughput should scale linearly. Consider a user masquerading as an NTP server instead of an NTP client. A private NTP server with a small number of unique clients in not abnormal. Each client provides a channel with 32 bps upstream and 10.6 bps downstream. With ten bonded channels, the effective channel is 320 bps upstream and 106 bps downstream. While this is still very slow, it provides a path forward for a very secure covert channel that would previously have been unusable.

Chapter 3

Mathematical Background [1]

Stochastic processes can model traffic flows. Depending on the underlying application, some of these models are a better fit than others. For our work, we consider protocols that have probabilistic state transitions. Each state transitions generates observable metadata. For instance, a transition from state 'a' to state 'b' will consistently generate an interpacket delay between 0.026 seconds and 0.031 seconds.

3.1 HMM

A Markov model is a tuple $G = (S, T, P)$ where S is a set of states of a model, T is a set of directed transitions between the states, and $P = \{p(s_i, s_j)\}$ is a probability matrix associated with transitions from state s_i to s_j such that:

$$\sum_{s_j \in S} p(s_i, s_j) = 1, \forall s_i \in S \tag{3.1}$$

A Markov model satisfies the Markov property, where the next state only depends on the current state. An HMM is a Markov model with unobservable states. A standard HMM [36, 57] has two sets of random processes: one for state transition and the other for symbol outputs. HMMs can model time series data [25]. This work uses a deterministic HMM [48, 47, 61, 30], and it has one random process for state transitions. Different output symbols are associated with transitions with

[1] This chapter has been adapted from [54].

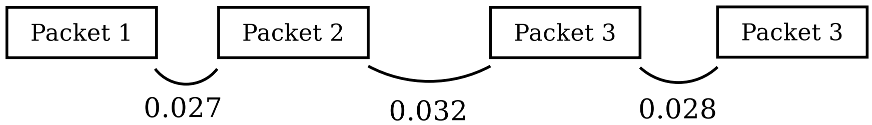

Figure 3.1: Stream of incoming packets with timing denoted [54].

different probabilities. This representations is equivalent to the standard HMM [67, 48].

3.1.1 Inferring Deterministic HMMs

Deterministic HMM inference is depicted in Figures 3.1-3.4. A stream of network packets, Figure 3.1, is observed. We calculate the interpacket delay (time between each packet) and plot the values in a histogram. Peaks in the histogram define the different states of the HMM. In Figure 3.2, there are three peaks, and we assign each peak a unique label. The stream of interpacket delays is re-interpreted using the assigned labels. A stream of labels, as shown in Figure 3.3, is used to infer the deterministic HMM shown in Figure 3.4. Each state in the HMM corresponds to a label, and each transition represents an output expression. The probability of an 'a' output expression from state 'b', divide the number of 'ba' strings by the number of 'b' strings. If there were 1000 occurrences of the string 'b', and we know the string 'ba' occurred 250 times, then 25% of the time we transitioned to state 'a'. The complete process for inferring deterministic HMMs is detailed in [40, 62]. Given a deterministic HMM, it is possible to generate a stream of packet timings.

3.1.2 Comparing Deterministic HMMs

In [48], the authors develop a normalized metric space for comparing HMMs, and in [78], the authors show a method for ensuring an HMM is significant. We use an alternative approach tailored to this challenge. Before determining whether two deterministic HMMs are equal, it is desirable to ensure the probability distribution functions (PDFs) used to generate the HMM are equal. To do this, we use the two-sample Kolmogorov-Smirnov (KS) test [20], which tests the null hypothesis (two sets of samples come from the same underlying distribution) against the alternate hypothesis (two sets of samples come from different underlying distributions). The KS statistic is the empirical

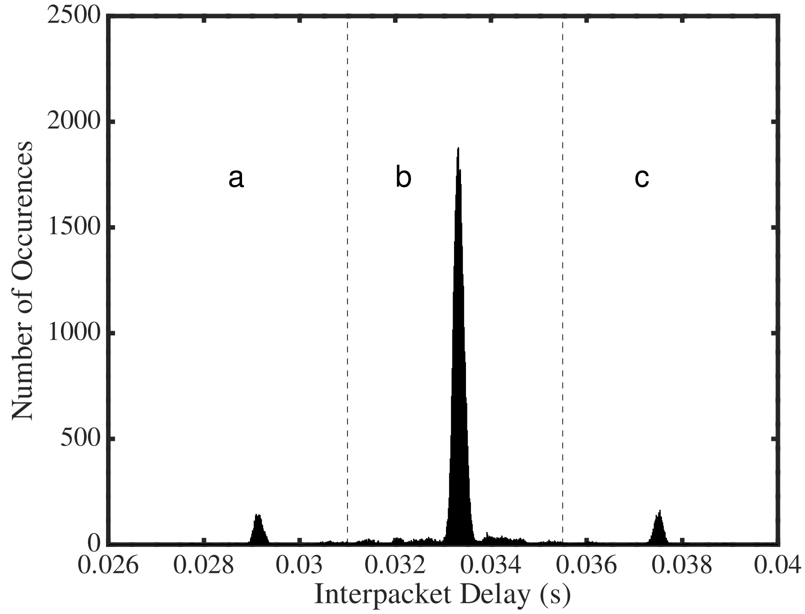

Figure 3.2: The timing values plotted in a histogram [54].

aba....aacbacab...

Figure 3.3: The packet timings converted to a stream [54].

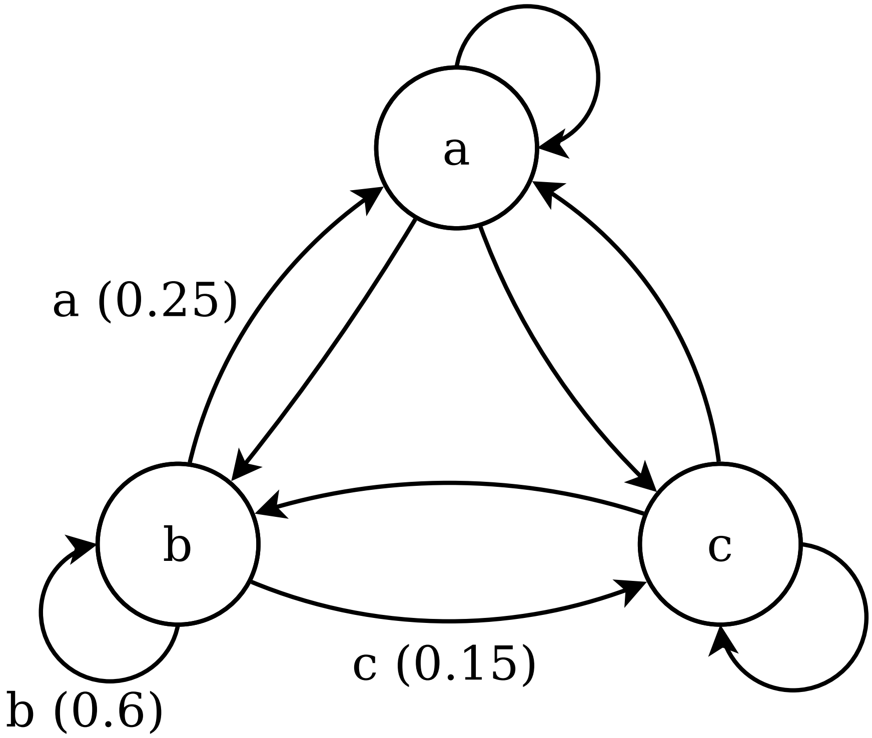

Figure 3.4: The deterministic HMM inferred from the stream of labels [54].

distribution function F_n, defined below.

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} I_{(-\infty,x]}(X_i) \tag{3.2}$$

Here, n refers to the number of identically independently distributed samples (X_i) taken from the sample space (X). Samples (X_i) are randomly chosen observations from Figure 3.1. The indicator function, $I_{(-\infty,x]}(X_i)$, is defined in Equation (3.3).

$$I_{[-\infty,x]}(X_i) = \begin{cases} 1, & X_i < x \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

The two-sample KS test compares the distance between the two empirical distribution functions using Equation (3.4).

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{2,m}(x)| \tag{3.4}$$

We reject the null hypothesis at the 5% confidence level using the following criterion.

$$D_{n,m} > 1.36\sqrt{\frac{n+m}{nm}} \tag{3.5}$$

For show equivalence between two deterministic HMMs are equivalent, it is sufficient to show all corresponding states in the deterministic HMM are equivalent. If all states are equivalent, the HMMs are equivalent. To show two states of a deterministic HMM are equivalent, we use the χ^2 test for homogeneity to test if the probability distributions for outgoing state transitions are statistically equivalent. Equation 3.6 shows the generic expression for the χ^2 statistic for homogeneity given P populations and C levels of the categorical variable.

$$\chi^2 = \sum_{i \in P} \sum_{j \in C} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}} \tag{3.6}$$

In this representation, $O_{i,j}$ is the number of occurrences observed in the state corresponding to i and the output expression corresponding to j. Similarly, $E_{i,j}$ is the number of *expected* occurrences for the combination of state and output expression. Equation (3.7) provides the expected number of occurrences.

$$E_{i,j} = \frac{n_i n_j}{n} \tag{3.7}$$

Here, n_i is the number of observations in state i, n_j is the number of observations at that level of the categorical variable, and n is the sample size. For threshold testing, Equation 3.8 gives the degrees of freedom (DF).

$$DF = (P-1)(C-1) \tag{3.8}$$

In this work, we compare two states (populations), so P is 2. Therefore, the DF for any given state is simply the number of output expressions (C) minus one.

3.2 Observation-based FTE

Directly sending UDP packets to a specific port is not enough. Capturing the packet in an analysis tool like Wireshark [22] will reveal the packet is malformed. While this rises to the level

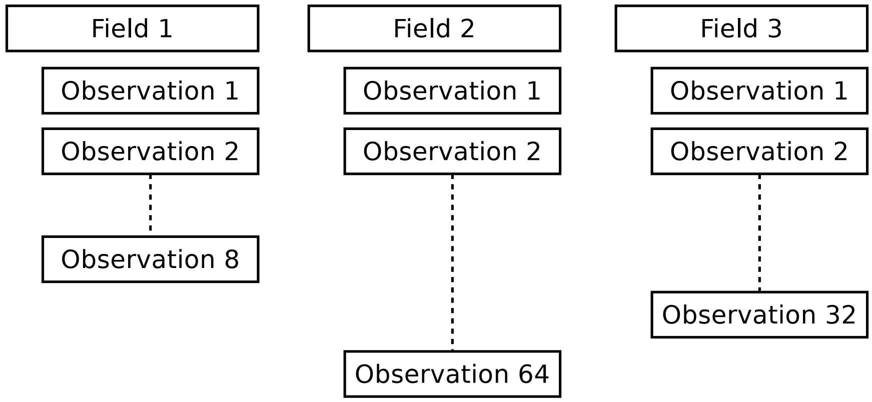

Figure 3.5: Example protocol to illustrate observation-based FTE [54].

of existing obfuscation PTs, it does not solve the problem. Traditional FTE takes the syntax of a protocol and creates a PCFG to map raw binary data to that protocol's syntax [33]. Determining the appropriate PCFG to model a protocol is left as an open research question, making it unrealistic to deploy [34, 35]. We propose observation-based FTE as a simple alternative. We collect a substantial amount of traffic and record all the unique observations for each field in the protocol. Zhong et al.'s work [81] used a primitive version of observation-based FTE that did not consider the upper bound on an FTE channel's information capacity.

Consider the fundamental information theory problem: Alice and Bob want to encode information using the protocol shown in Figure 3.5. Assume both Alice and Bob have the same list of unique observed values for each field in the protocol.

Theorem 1. *For a given protocol, the maximum amount of information that can be encoded in a packet using observation-based FTE is given by:*

$$S = \sum_{\gamma_i \in \Gamma} log_2(|\boldsymbol{\gamma}_i|) \tag{3.9}$$

Where $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$ *is the set of* n *fields in the protocol, and* $|\gamma_i|$ *is the number of unique observations in that field.*

Proof. The Shannon entropy of that field gives the maximum amount of information encoded in a particular field using observation-based FTE.

$$H(\gamma_i) = -\sum_{x \in \gamma_i} p(x)\log_2(p(x)) \tag{3.10}$$

Each stream of n bits is equally likely–since the data mapped to the protocol is encrypted using AES encryption and AES produces a high-entropy bitstream [49], we can assume 0 and 1 are equally likely in practice. Therefore, the choice of each observation is equally likely. This simplifies Equation (3.10) as follows.

$$\begin{aligned} H(\gamma_i) &= -\sum_{x \in \gamma_i} \frac{1}{|\gamma_i|}\log_2\left(\frac{1}{|\gamma_i|}\right) \\ &= -|\gamma_i|\frac{1}{|\gamma_i|}\log_2\left(\frac{1}{|\gamma_i|}\right) \\ &= -\log_2\left(\frac{1}{|\gamma_i|}\right) \\ &= \log_2(|\gamma_i|) \end{aligned} \tag{3.11}$$

The maximum amount of information encoded in a single packet is the sum of information encoded in each field in the packet.

$$\mathrm{S} = \sum_{\gamma_i \in \Gamma} \log_2(|\gamma_i|)$$

□

Performing these calculations on the Synchrophasor protocol finds a single UDP packet can contain 516 bits. Since this is smaller than the typical TCP packet, it is necessary to segment TCP packets for transmission. The optimal average goodput ($\mathrm{G_{avg}}$) can be calculated with Equation (3.12), where S is found using Equation (3.9), and $\mathrm{T_{avg}}$ is the average interpacket delay, which is 0.03334 seconds for Synchrophasor traffic. Equation 3.12 yields an theoretical average goodput of 15,477 bits per second.

$$\mathrm{G_{avg}} = \frac{\mathrm{S}}{\mathrm{T_{avg}}} \tag{3.12}$$

Figure 3.6 shows data segmentation:

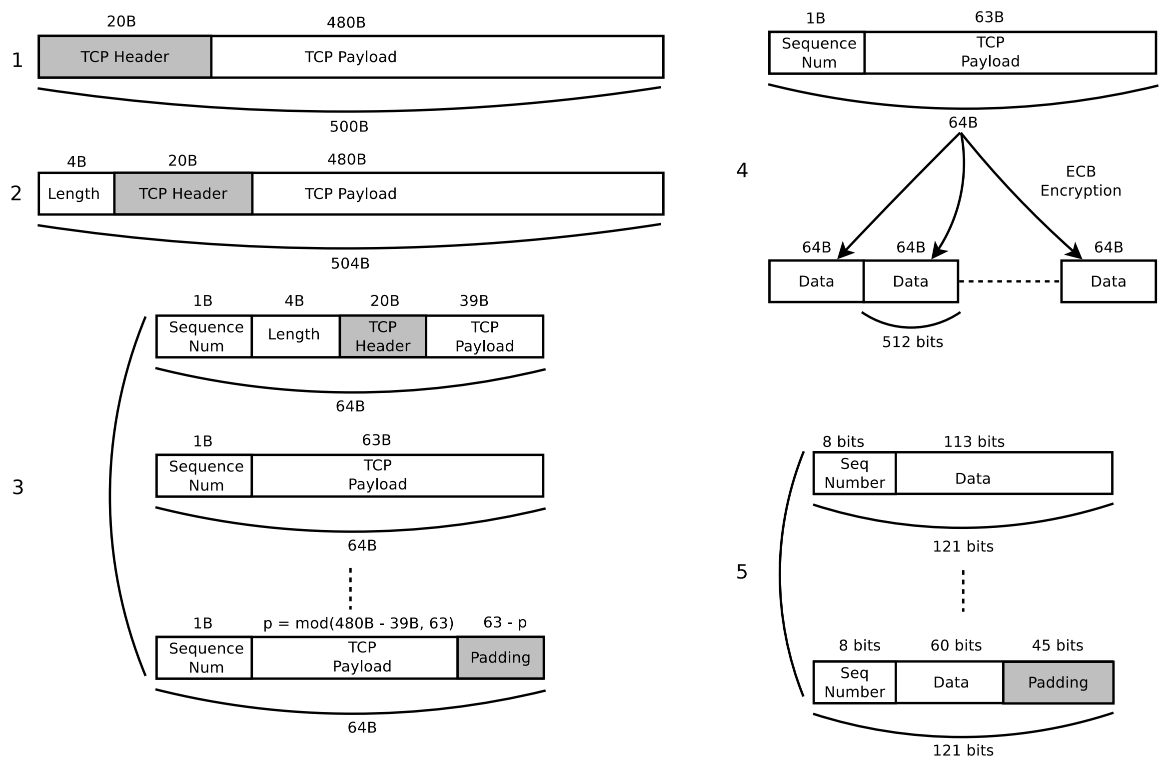

Figure 3.6: Process for segmenting TCP packets for transmission [54].

1. The original TCP packet.
2. Prepend the packet length to the beginning of the packet as a four-byte unsigned integer.
3. Break the packet into 63-byte chunks and prepend each chunk with a one-byte sequence number. The sequence number allows the chunks to be reassembled later into the original TCP packet. Depending on the packet's size, there may not be enough payload data to fill the final chunk. In this case, append random data to the end.
4. Encrypt each 64-byte chunk with Electronic Code Book (ECB) encryption. Since packets may arrive out-of-order, cipher block chaining (CBC) is impractical.
5. Encode each 64-byte (512 bit) chunk into a 516 bit UDP payload using the observation-based FTE method.

Chapter 4

Developing an Observation-based FTE Covert Communication Channel [1]

We combined the insights from Chapter 2 with the tools from Chapter 3 to inform a framework that will transform arbitrary network traffic into statistically equivalent traffic that can be dissected by a protocol analyzer without errors. For this framework, we consider *protected static* protocols. Nation-states cannot block these protocols without significant collateral damage, and they are stateless at the transport layer (typically UDP).

There are two main challenges–payloads and timing. We must convert arbitrary TCP traffic into the *host protocol's* UDP payloads. Next, the outgoing UDP packets must conform to the *host protocol's* timing. In Section 4.1, we discuss how to convert arbitrary TCP traffic into UDP payloads, and in Section 4.2, we discuss how to adjust the interpacket delay to emulate the host protocol.

4.1 Transport Converter

The transport converter encapsulates the TCP header and payload in a UDP packet.

We use Scapy [26] (a Python library for packet capture, manipulation, and injection) to

[1] This chapter has been adapted from [54].

capture TCP packets. The captured packet includes both the TCP header and payload. We send traffic to a closed loopback port, and Scapy sniffs the loopback interface looking for traffic destined to that port. By default, when a closed port receives a TCP packet, it responds with a TCP RST (reset) packet. The RST packet immediately terminates the connection and ceases all communication with the other host. The firewall dictates this low-level response, so we modified it (`iptables` on Linux) to disable sending these packets. The following command disables RST packets on all ports on Linux.

```
sudo iptables -A OUTPUT -p tcp --tcp-flags RST \
  RST -j DROP
```

Listing 4.1: The iptables command to disable RST packets [54].

In Figure 4.1, an application on the client sends a packet to a closed local port (8001 in our example). Scapy sniffs the network stack and captures the entire packet. Then, we strip the Ethernet and IP layers and send the packet to an encapsulator. The encapsulator performs the transformation described in Section 4 and generates new Ethernet, IP, and UDP layers. We send all UDP packets to the port that corresponds to the *host protocol*, and we massage he packets timing as described in Section 4.2.

On the server, the decapsulator listens to the predetermined port for the UDP packet. Once it receives the UDP packet, it takes the UDP payload, reverses the transformation described in Section 4, and sends it to a Scapy packet injector. The Scapy packet injector creates new Ethernet and IP layers to make the packet look like it originated from the local machine. Then, Scapy injects the new packet into the local network stack, and the application listening on Port 8001 receives it. While this example is unidirectional, it is trivial to make a bidirectional example–the client and server include both sides of this design.

4.2 Packet Timing

The HMM timing model described in Section 3.1 was input to the Protocol Proxy, which queries the timing model for a timing value. When a timing value is requested, we examine the current state, choose an output expression based on the probability distribution of the current state, and choose a timing value from the output expression group. The model advances to the chosen state. The Protocol Proxy waits for the allotted time before sending a packet. If there are no packets to send, random data are encoded and sent. The server drops these packets. The Protocol Proxy sends placeholder packets to maintain the *host protocol's* timing model.

4.3 Experiment Setup

We collected 770,000 samples from the PMUs in Clemson's RTPIS Laboratory [21] and used these samples to build the timing model described in Section 4.2. All testing was performed in Clemson's security lab with clean installations of Arch Linux (kernel version 4.17.2-1). Figure 4.2 shows the experiment setup. We used `scp` to transfer data over the Protocol Proxy to an SSH server on a remote machine. We launched the Protocol Proxy server using the following command.

```
server# protocol_proxy server 192.168.10.23 8001
```

Since the Protocol Proxy requires privileged access because it uses raw sockets. The 'server' option tells the Protocol Proxy to expect packets originating from the specified port (8001). The IP address (192.168.10.23) is the client's IP address that will connect to the server. The next step is We execute the `iptables` command shown in Listing 4.1 to disable reset packets (TCP RST) sent in response to a connection attempt on a closed port (8001). Next, we configure the SSH server to listen to port 8001 for incoming connections in the `/etc/ssh/sshd_config` file. We used SSH (OpenSSH_7.7p1) for the server. It was necessary to configure a non-standard port to avoid conflict when forwarding the traffic. We launched the Protocol Proxy client as a privileged user with the following command.

```
client # protocol_proxy client 192.168.10.24 8001
```

The 'client' option tells the transport to expect packets destined for the specified port (8001). The IP address (192.168.10.24) is the IP address of the host executing the program. The client also does not open a local port, so we must apply the same rules in Listing 4.1 to the client.

A one-kilobyte data file was transferred from the client to the server using `scp` as shown below.

```
client # scp -P 8001 file 127.0.0.1: file
```

We captured traffic between the client and the server to infer another HMM using this generated traffic. The χ^2-test tested equality between this second HMM to the original HMM used by the Protocol Proxy. Finally, we measured goodput as the time required to transfer the one-kilobyte data file.

4.4 Results

Figure 4.3 shows the histogram of interpacket delay times for the Synchrophasor traffic captured in Clemson's RTPIS laboratory. The prominent peaks in the histogram are the output expressions. Using the techniques described in Section 3.1, we inferred the deterministic HMM in Figure 4.4.

With this HMM, it was then possible to generate Synchrophasor traffic with observation-based FTE and accurate timing. Figure 4.5 shows a Wireshark deconstruction of the traffic generated with our Protocol Proxy. Wireshark correctly identifies the Protocol Proxy traffic as Synchrophasor traffic and can parse the values from the payload. The checksum is also correctly calculated.

Figure 4.6 shows the histogram of interpacket delay times for the generated traffic with the output expressions labeled. Figure 4.7 shows the deterministic HMM inferred from the histogram to model the timing patterns of the Protocol Proxy traffic. Visually, this model appears almost identical to the model used to generate the traffic.

Before determining if the two deterministic HMMs were equal, we used the two-sample KS test to compare the two distributions (shown in Figure 4.3 and Figure 4.6). We applied this test to 100 random samples from each distribution. The p-value for the two-sample KS test was 0.21, so with a threshold of 0.05, we fail to reject the null hypothesis. The Protocol Proxy's interpacket delay times are from the same probability distribution as the interpacket delay times of the original Synchrophasor traffic.

We checked the HMMs for state-wise equality using the χ^2 test for homogeneity to determine if the two deterministic HMMs were equal. The p-values for the χ^2 test are shown in Table 4.1. The first comparison (inferred-inferred) infers two HMMs using 10,000 samples and a random starting point in the original traffic. From these values, we fail to reject the null hypothesis (with an α value of 0.05) for every state and are left to conclude the traffic is *homogeneous*, which means it does not change over time. The second comparison (generated-inferred) infers one HMM from the Protocol Proxy traffic and another HMM from the original Synchrophasor traffic. From these values, we fail to reject the null hypothesis (with an α value of 0.05) for every state and are left to conclude the

Table 4.1: State-wise χ^2 test for homogeneity comparing HMMs [54].

State Comparison	Inferred-Inferred (p-value)	Generated-Inferred (p-value)
a-a	0.75	0.82
b-b	0.19	0.37
c-c	0.06	0.15

Table 4.2: Comparison of observed and theoretical goodputs through the Protocol Proxy [54].

	Baseline	Theoretical	Observed
Goodput	54 Mbps	15,477 bps	182 bps

traffic from the Protocol Proxy is equivalent to the *homogeneous* Synchrophasor traffic.

We measured the baseline goodput (link speed) at 54 Mbps and the goodput through the PMU Protocol Proxy at 182.2 bits per second. These values are compared to the theoretical goodput in Table 4.2. The difference between theoretical and observed goodput is attributed to retransmission and packet overhead (sending the TCP header through the Protocol Proxy).

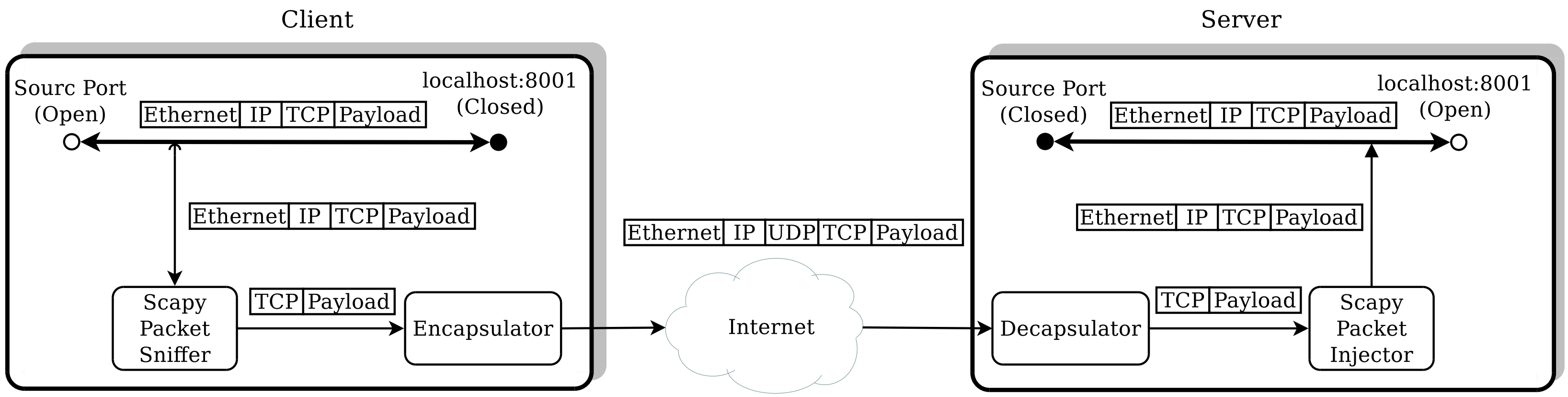

Figure 4.1: Architecture for the transport converter. TCP traffic is encapsulated in a UDP payload [54].

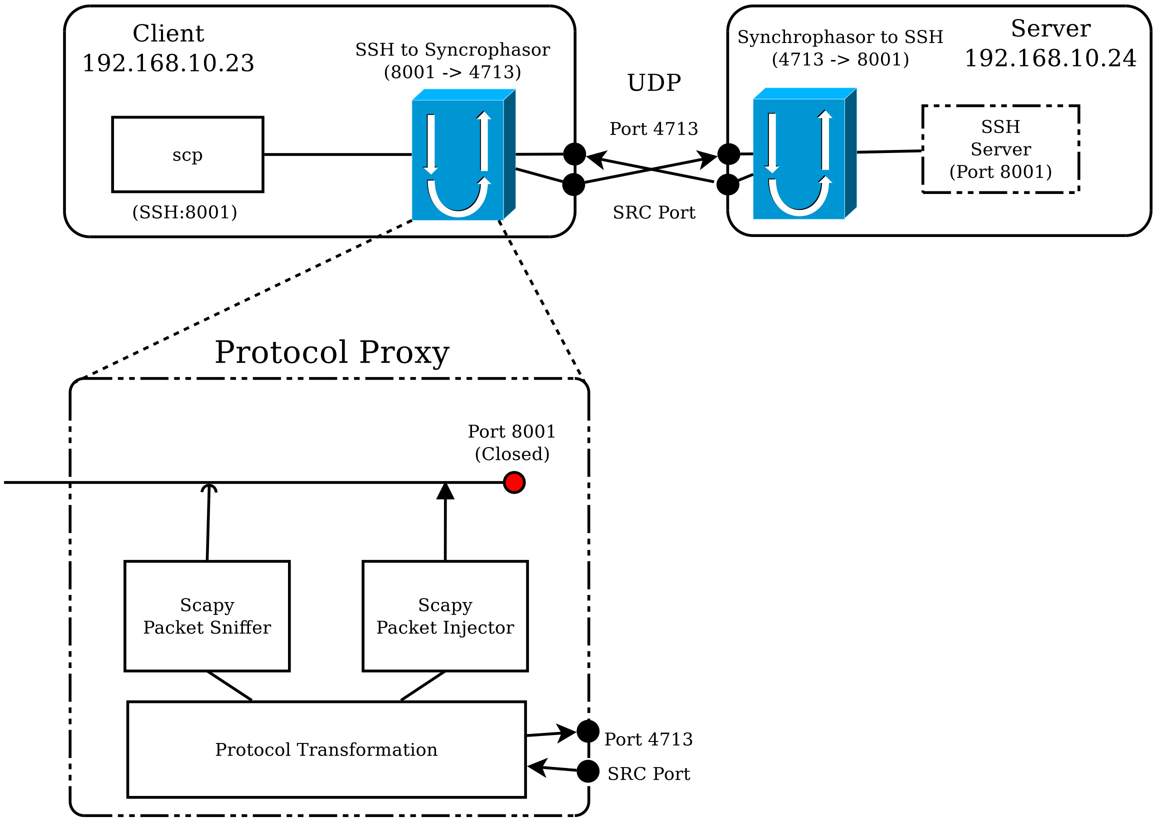

Figure 4.2: The Protocol Proxy integrated for use with SCP [54].

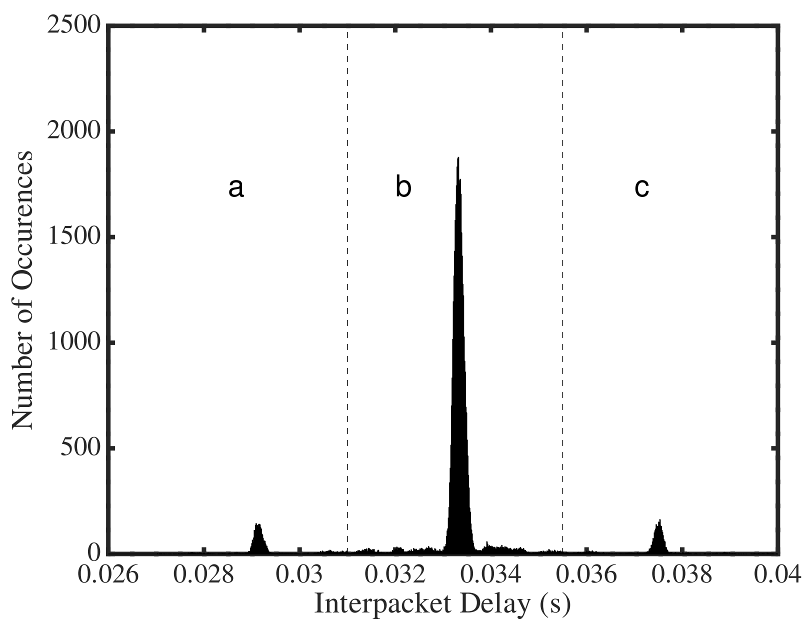

Figure 4.3: Histogram of the interpacket delay of real Synchrophasor traffic with states labeled [54].

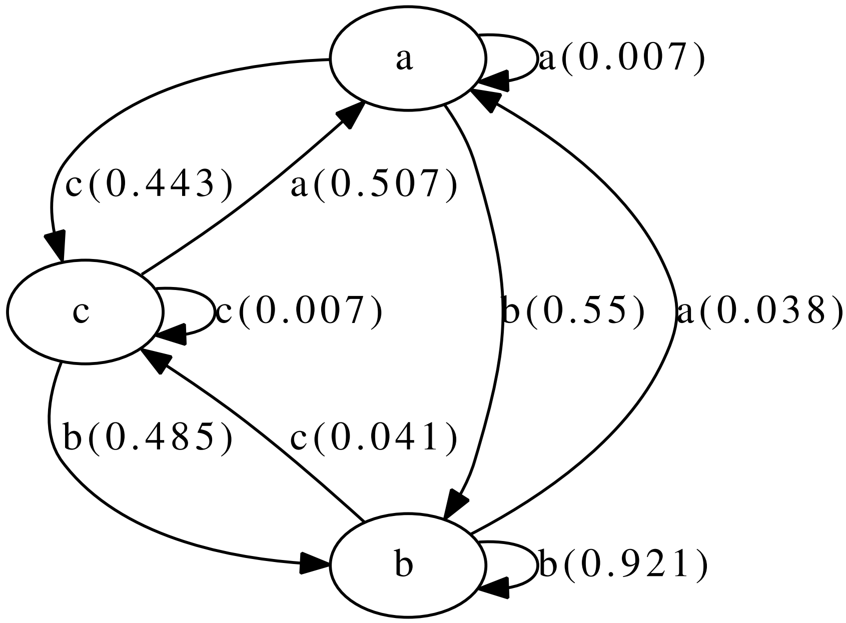

Figure 4.4: HMM generated from the interpacket delay of Synchrophasor traffic [54].

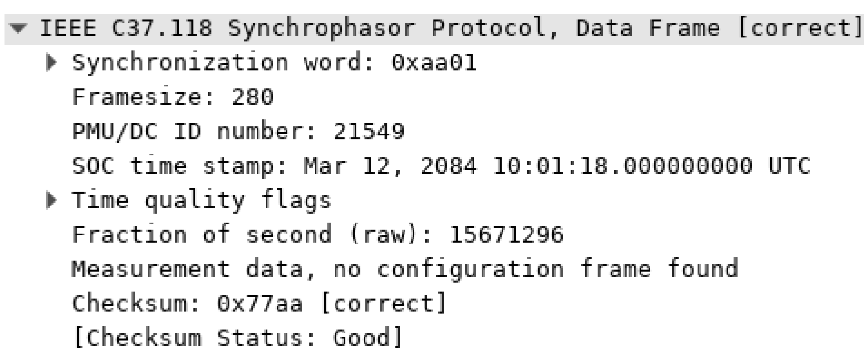

Figure 4.5: Wireshark decoding of the Protocol Proxy traffic [54].

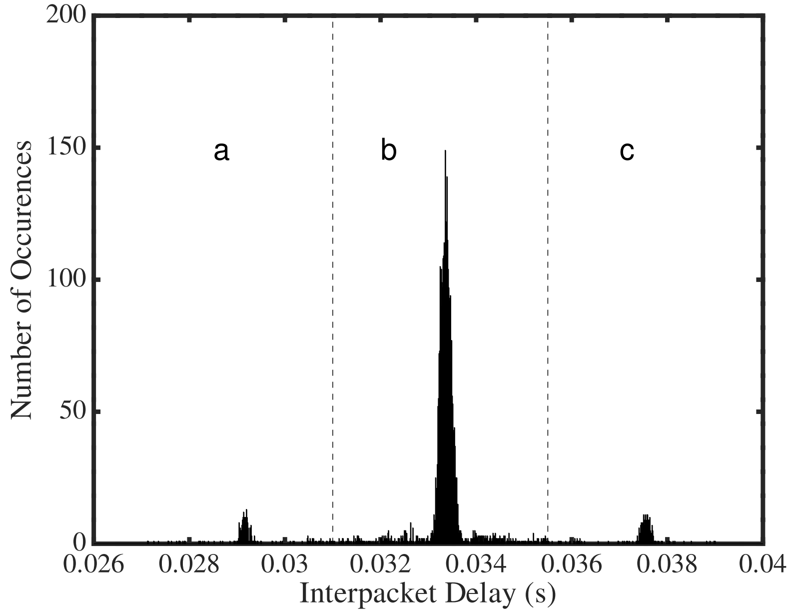

Figure 4.6: Histogram of the interpacket delay of generated Synchrophasor traffic with states labeled [54].

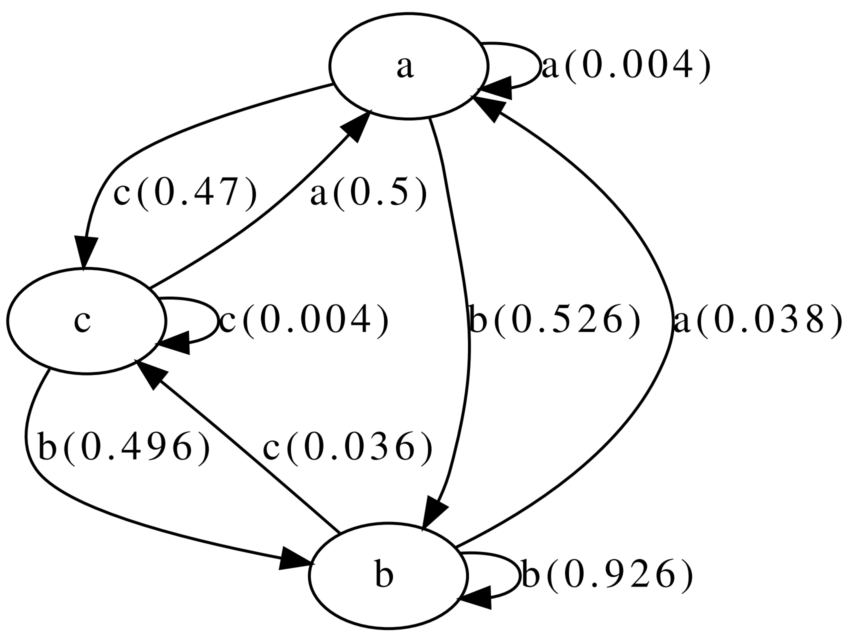

Figure 4.7: HMM generated from the interpacket delay of the generated Synchrophasor traffic [54].

Chapter 5

Increasing Throughput of Covert Channels

There are two notable mpTCP-based approaches to increasing the bandwidth of a covert channel. The first approach is to split the covert channel into multiple TCP streams, and the second approach is to bond multiple covert channels. Each approach has benefits and limitations and is suited to a particular use-case.

Both architectures leverage multipath TCP (mpTCP) [8], a Linux kernel implementation. This native support for multipath routing enables multipath TCP on certain interfaces. The mpTCP kernel module splits TCP connections with mpTCP-enabled servers across those interfaces.

5.1 Multipath Architecture

The multipath bonded channel architecture takes a single TCP stream from Tor, a VPN, or a PT, and splits it into several mpTCP streams as shown in Figure 5.1. We route each of these streams through an intermediary node in a separate legal jurisdiction. The intermediary nodes re-route traffic back to a proxy server, which forwards the traffic to its final destination.

The main advantage of this approach is increased throughput. If a client uses separate interfaces that avoid a shared bottleneck, it is possible to increase the throughput of tools like Tor and OpenVPN. Even without separate interfaces, this approach still increases the bandwidth of some covert channels. Coincidentally, this makes fingerprinting some covert channels more difficult. Some traffic analyzers may classify these streams as split-routing traffic. Split routing occurs whenever a traffic analyzer sees only part of a stream, and it is a common struggle for network traffic engineers and security researchers alike.

Leveraging these multipath TCP streams, clients can effectively increases and anonymity and throughput. However, these TCP streams are identifiable as mpTCP traffic

The mpTCP protocol is legitimate, and its use is not inherently suspicious. However, it has not seen widespread adoption, and no critical services rely on it yet, so blocking it would not cause enough collateral damage to deter a nation-state actor.

5.2 Multisession Architecture

The multisession bonded channel architecture improves upon the multipath bonded channel architecture by splitting a TCP stream across encrypted or covert channels, as shown in Figure 5.2. For example, instead of having an OpenVPN connection split into N different mpTCP streams, a regular TCP connection is split into N different mpTCP streams, and we tunnel each of those streams through OpenVPN. OpenVPN was necessary because mpTCP is only capable of splitting a TCP stream across multiple tun interfaces. We used socat, a Linux utility that created a tun pair between the client and the server for testing the PP. OpenVPN could not initialize over the PP, but socat had a (relatively) low overhead and was able to establish the tunnel in the low-bandwidth environment. In all use-cases, the channels would pass through intermediary nodes in different legal jurisdictions.

This approach also has the advantage of increased throughput, and in some cases, it will improve throughput where the multipath architecture does not. For instance, with covert-channels that massage timing, such as the PP, the multipath architecture permutes the timing observed by an attacker, and since both architectures are transparent to the client, the timing model limits the effective throughput. However, in the case of the multisession bonded channel architecture, N independent covert channels can be instantiated, and each one has its timing model. This architecture is still transparent to the client, but in this case, the throughput scales linearly.

It is also possible to combine different covert channels to increase anonymity or improve resiliency. Often, PT servers experience outages or nation-states block specific transports. This

Table 5.1: Experiment Network Topology.

Host	Device	IP
Client	IFace-1	10.1.0.1
	IFace-2	10.2.0.1
	tun1	10.11.0.6
	tun2	10.12.0.6
Node-1	IFace-1	10.1.0.2
	IFace-2	10.3.0.2
Node-2	IFace-1	10.2.0.2
	IFace-2	10.4.0.2
Server	IFace-1	10.3.0.1
	IFace-2	10.4.0.1
	tun1	10.11.0.1
	tun2	10.12.0.1

approach allows users to try a wide range of techniques simultaneously and adaptively respond to channels taken offline. This approach provides the most benefits with the fewest drawbacks.

5.3 Experiment

To measure the performance of the multipath and multisession architectures, we conducted several baseline experiments. The experiments allowed us to determine the baseline performance and assess the performance increase of each architecture. We conducted the experiments on GENI with XenVM Ubuntu 16.04 hosts. The client and the server used kernel 4.19.55 and the multipath TCP (mpTCP) kernel module installed. The topology is shown below in Figure 5.3.

The IP address assignments are shown below in Table 5.1. Table 4.2 shows the results of all experiments.

5.3.1 Baseline

The first experiment, shown in Figure 5.4, was conducted to determine the link speed through the intermediary nodes using Iperf.

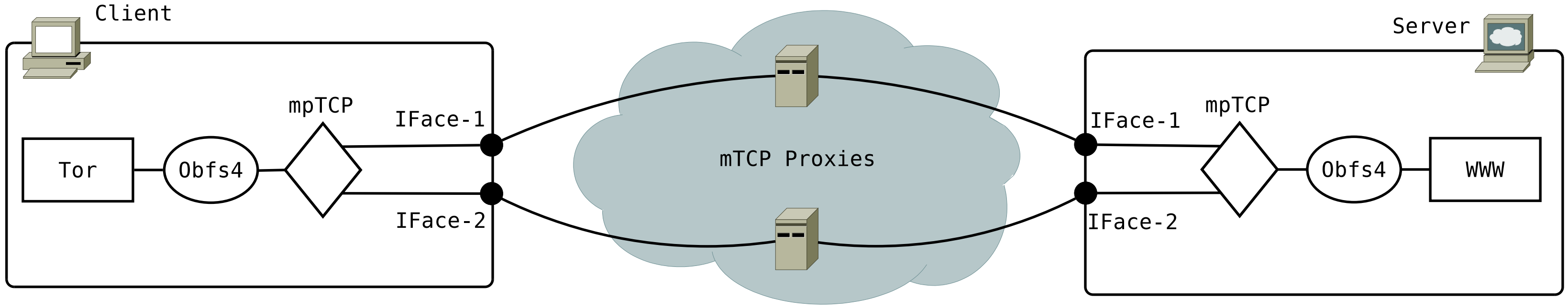

Figure 5.1: Multipath TCP architecture.

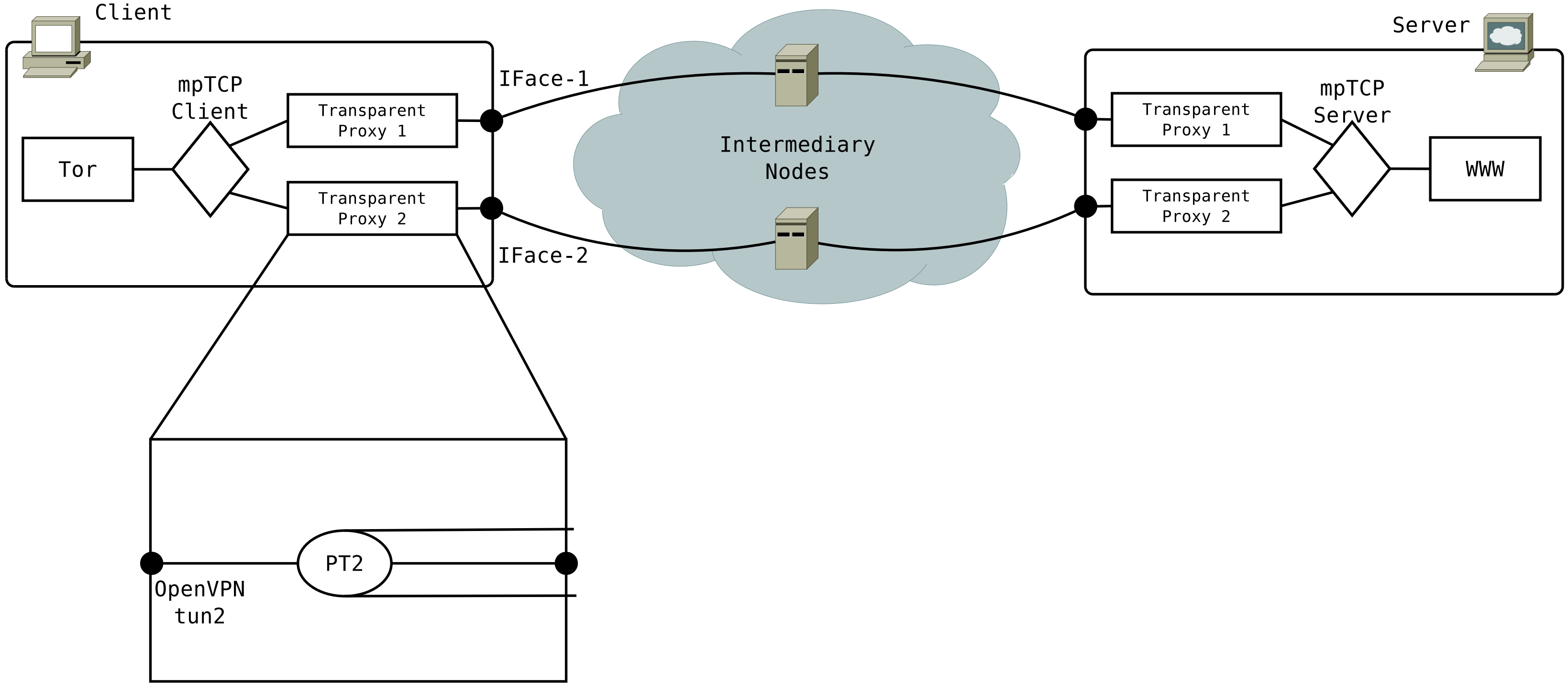

Figure 5.2: Multi-session TCP architecture.

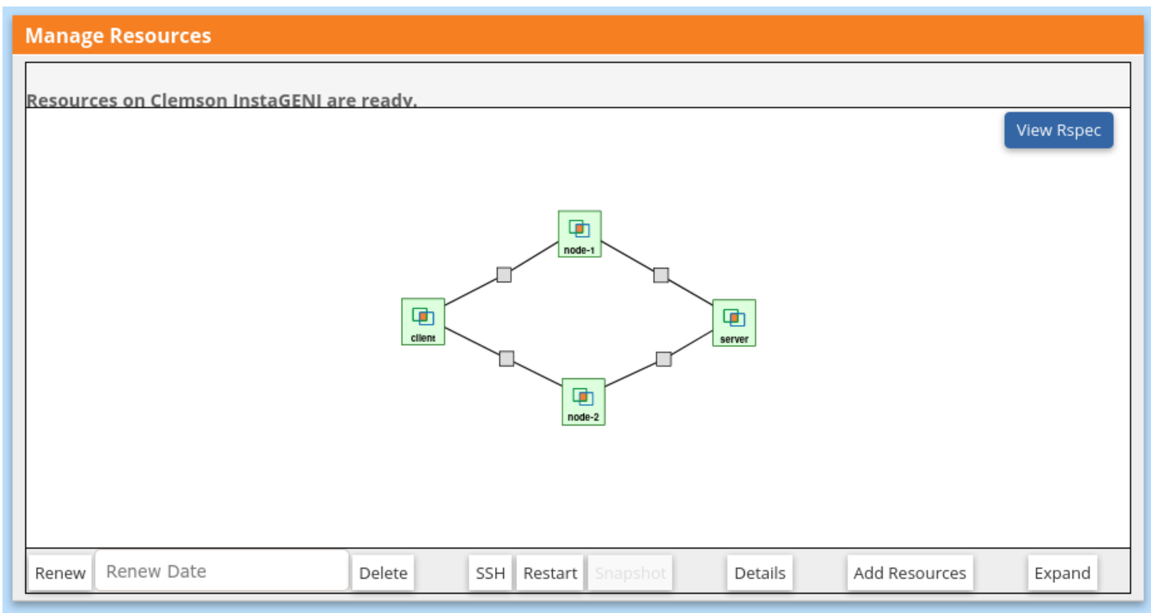

Figure 5.3: GENI experimental setup.

From Listing 5.1, we ran Iperf with the server flag on the server, and we disabled mpTCP on eth0, eth1, and eth2 on the client before running Iperf in client mode and connecting to ‘10.1.0.2‘. The client leverages the NAT forwarding on the intermediary nodes to route the traffic to from the client-facing interface (‘10.1.0.2‘, Table 5.1) to the server (‘10.3.0.1‘, Table 5.1).

```
server$ iperf -s

client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ iperf -c 10.1.0.2 -t 1
```

Listing 5.1: Commands to run the baseline throughput experiment.

5.3.2 Baseline VPN

Next, we conducted the experiment shown in Figure 5.5 to establish the baseline goodput through a VPN. We used OpenVPN in client-server mode with the default settings. The client was assigned ‘10.11.0.6’ on tun1, and the server was assigned ‘10.11.0.1’ on tun1, as shown in Table 5.1.

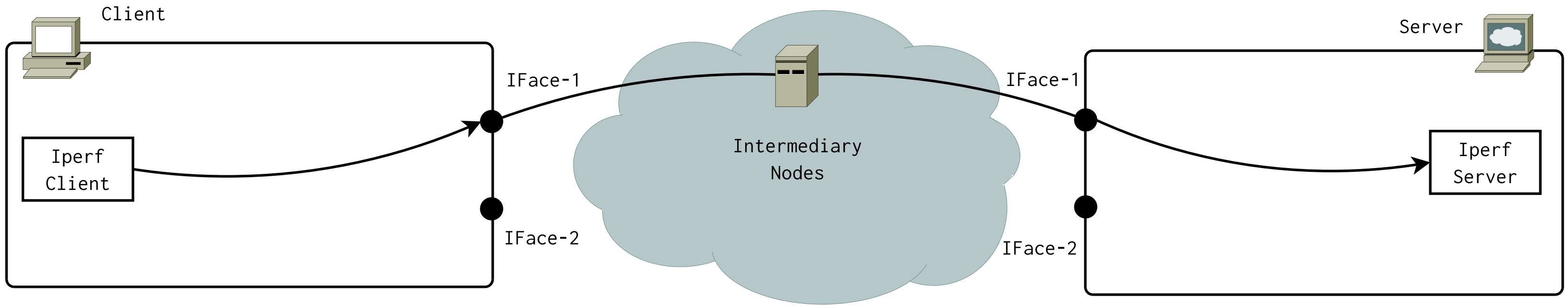

Figure 5.4: Baseline throughput experiment.

From Listing 5.2, we start OpenVPN and Iperf on the server, then start OpenVPN on the client. Next, we disable multipath TCP on all interfaces on the client and start the Iperf client.

```
server$ sudo openvpn --config openvpn-server/server-1.conf

server$ iperf -s


client$ sudo openvpn --config openvpn-server/client-1.2.conf

client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath off

client$ iperf -c 10.11.0.1 -t 15
```

Listing 5.2: Commands to run the baseline VPN throughput experiment.

5.3.3 Baseline PT

Figure 5.6 shows the experiment to calculate baseline goodput through the obfs4 PT. The obfs4 PT is widely used, easily configurable, and one of the first successful PTs. We used shapeshifter-dispatcher to enable obfs4.

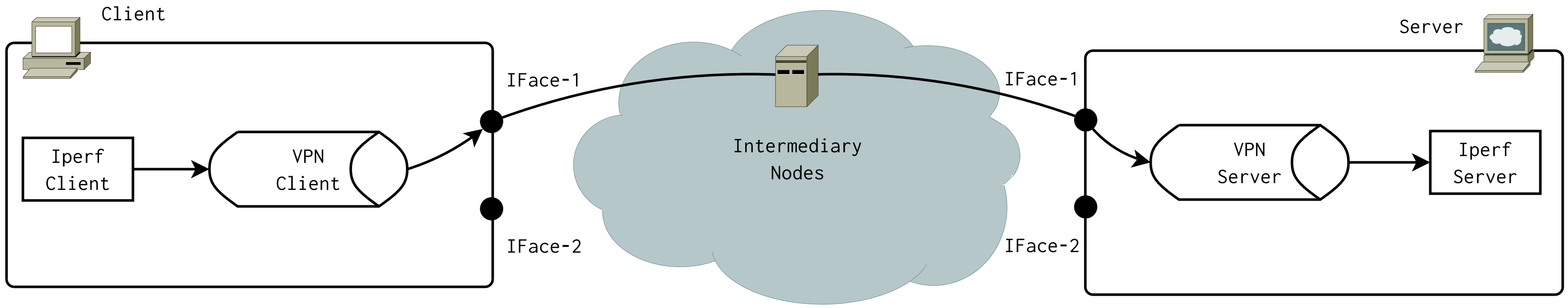

Figure 5.5: Baseline VPN throughput experiment.

As shown in Listing 5.3, we start the Iperf server and the PT server. The PT server uses the '-transparent' option to forward traffic, and it specified '-ptversion 2' as the standard interface version. We specify transports using '-transports obfs4', and we specify the state directory (certificate location) using '-state state'. The bind-address ('-bindaddr obfs4-10.3.0.1:1190') specifies that the PT server listen on eth0 (Table 5.1) port 1190 for incoming PT connections. The 'orport 10:3.0.1:5001' specifies where the PT server will forward incoming connections after decoding.

On the client, multipath TCP is disabled on all interfaces. We start the PT client with shapeshifter-dispatcher using the previously described options. The PT client listens on 'proxylistenaddr 127.0.0.1:5001' and forwards traffic to '-target 10.1.0.2:1190'. The intermediary node translates '10.1.0.2' to '10.3.0.1'. The '-options' specify JSON options containing the certificate fingerprint. The Iperf client is then run to determine the throughput through the PT.

```
server$ iperf -s
sudo shapeshifter-dispatcher -server -transparent -ptversion 2
  -transports obfs4 -state state -bindaddr obfs4-10.3.0.1:1190
  -orport 10.3.0.1:5001 -logLevel DEBUG -enableLogging


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off

client$ shapeshifter-dispatcher -client -transparent -ptversion 2
  -transports obfs4 -proxylistenaddr 127.0.0.1:5001 -state state
  -target 10.1.0.2:1190 -options
  '{"cert": "<FINGERPRINT>", "iat-mode": "0"}'
  -logLevel DEBUG -enableLogging

client$ iperf -c 127.0.0.1 -t 15
```

Listing 5.3: Commands to run the baseline PT throughput experiment.

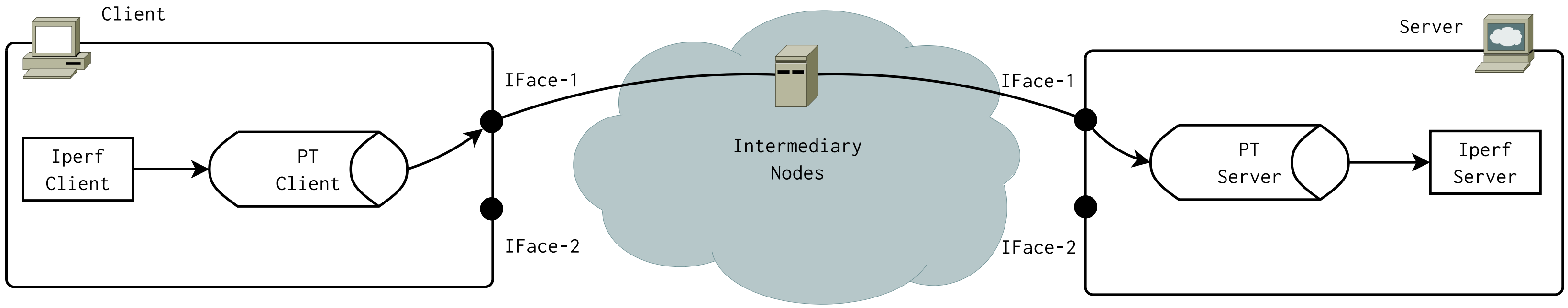

Figure 5.6: Baseline PT throughput experiment.

5.3.4 Baseline VPN-PT

Figure 5.7 shows the experiment to calculate the baseline goodput through a VPN using a PT for obfuscation. We used OpenVPN obfuscated with obfs4 because of their extensive utilization. In this experiment, we calculate the goodput of a single TCP stream between the client and the server.

As shown in Listing 5.4, we start the PT server as described in Section 5.3.3. Then, we start the OpenVPN server listening on port 1190, where the PT server sends traffic. On the client, multipath TCP is disabled on all interfaces. The PT client is started with shapeshifter-dispatcher using the same options described in Section 5.3.3. The Iperf client is then run to determine the goodput through the VPN-PT.

```
server$ sudo openvpn --config openvpn-server/server-1.conf

server$ sudo shapeshifter-dispatcher -server -transparent -ptversion 2
  -transports obfs4 -state ~/shapeshifter-dispatcher/state -bindaddr obfs4-10.3.0.1:1190
  -orport 10.3.0.1:1194 -logLevel DEBUG -enableLogging

server$ iperf -s


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath off

client$ sudo openvpn --config openvpn-server/client-1.conf

shapeshifter-dispatcher -client -transparent -ptversion 2 -transports obfs4
  -proxylistenaddr 127.0.0.1:1191 -state state -target 10.1.0.2:1190
  -options '{"cert": "<FINGERPRINT>", "iat-mode": "0"}'
  -logLevel DEBUG -enableLogging

client$ iperf -c 10.11.0.1 -t 15
```

Listing 5.4: Commands to run the baseline VPN-PT throughput experiment.

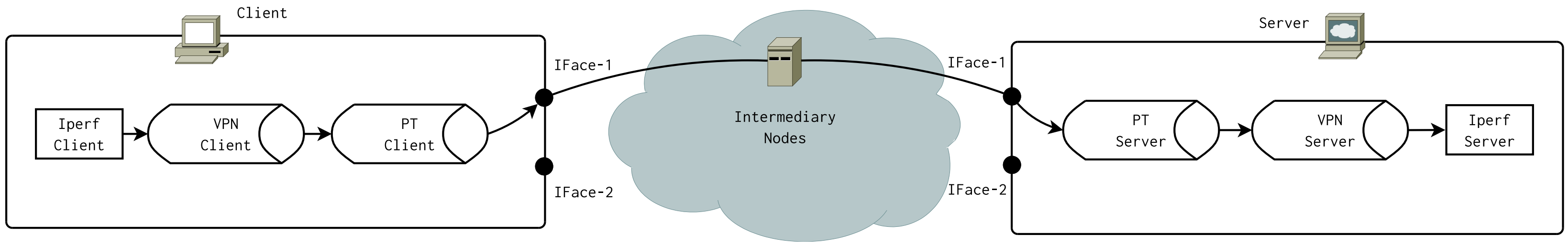

Figure 5.7: Baseline VPN-PT throughput experiment.

5.3.5 Baseline Socat

Next, we conducted the experiment shown in Figure 5.8 to establish the baseline goodput through a socat. Socat created a paired tun connection between the client and the server. The client was assigned '10.11.0.6' on tun1, and the server was assigned '10.11.0.1' on tun1, as shown in Table 5.1.

From Listing 5.5, we start socat and Iperf on the server, then start socat on the client. Next, we disable multipath TCP on all interfaces on the client and start the Iperf client.

```
server$ sudo socat -d -d TCP-LISTEN:9001,reuseaddr TUN:10.11.0.1/24,up,tun-name=tun1
server$ iperf -s
client$ sudo socat TCP:10.1.0.2:9001 TUN:10.11.0.6/24,up,tun-name=tun1
client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath off
client$ iperf -c 10.11.0.1 -t 15
```

Listing 5.5: Commands to run the baseline socat throughput experiment.

5.3.6 Baseline PP

Figure 5.9 shows the experiment to calculate baseline goodput through the PMU PP. The PMU channel is very low throughput, so we elected to test throughput by curling a 500B file instead of using Iperf, which does not function over the PP.

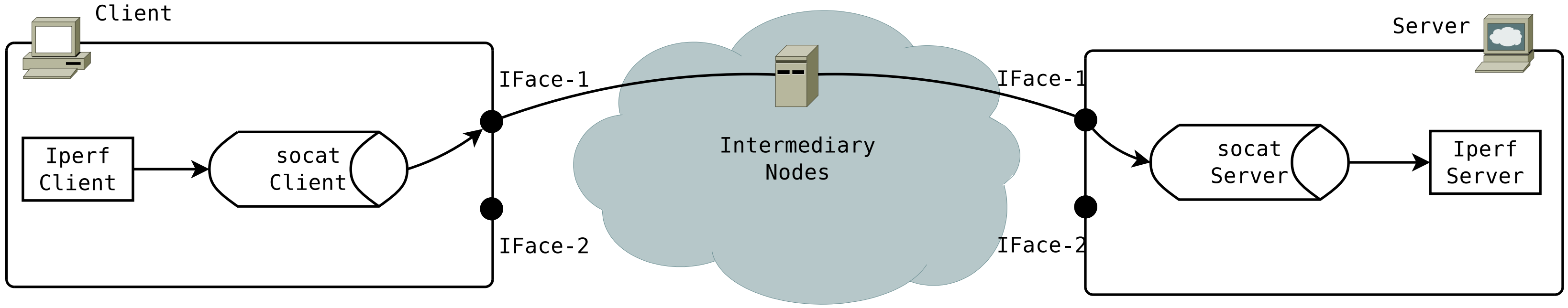

Figure 5.8: Baseline socat throughput experiment.

As shown in Listing 5.6, we start the HTTP server and the PP server. We pass the PP server a config file and tell it to use the '-p pmu' protocol in server mode, and we forward traffic on port 9001 to 10.1.0.1.

On the client, multipath TCP is disabled on all interfaces. We start the PP client using the previously described configuration file and protocol. The PP client listens on port 9001 and forwards traffic to 10.1.0.2. The intermediary node translates '10.1.0.2' to'10.3.0.1', and then we POST data with curl to determine the throughput of the channel.

```
server$ python3 ./http_server.py
server$ sudo ./protocol_proxy.py -c config/protocol-proxy.cfg -p pmu
  server 10.1.0.1 9001


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off

client$ sudo ./protocol_proxy.py -c config/protocol-proxy.cfg -p pmu
  client 10.1.0.2 9001

client$ time curl -X POST --data-binary "@500B.data" 127.0.0.1:9001/store.data
```

Listing 5.6: Commands to run the baseline PP throughput experiment.

5.3.7 Baseline socat-PP

Figure 5.10 shows the experiment to calculate the baseline goodput through a VPN using a PT for obfuscation. We used OpenVPN obfuscated with obfs4 because of their extensive utilization. In this experiment, we calculate the goodput of a single TCP stream between the client and the server.

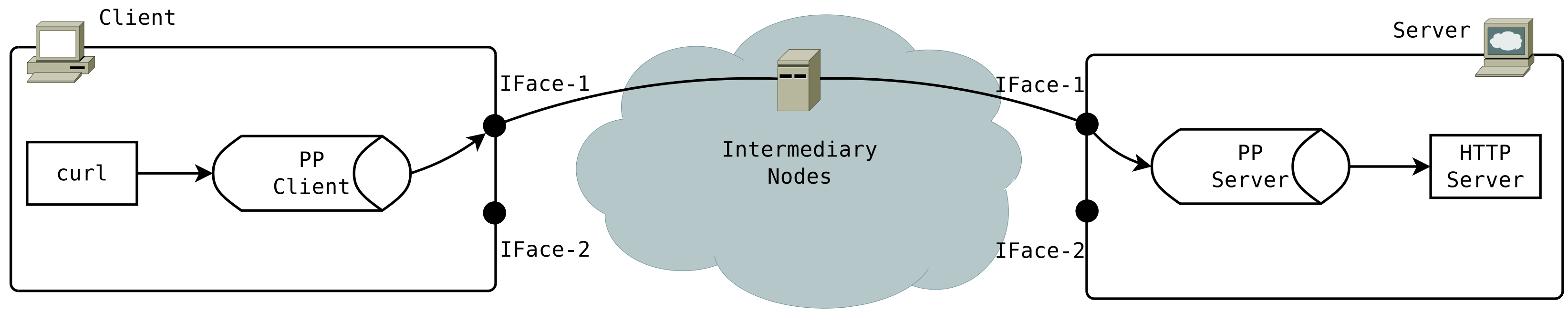

Figure 5.9: Baseline PP throughput experiment.

As shown in Listing 5.7, we start the PP server as described in Section 5.3.6. Then, we start the socat server listening on port 9001, where the PP server forwards traffic. On the client, multipath TCP is disabled on all interfaces. We start the PP client using the same options described in Section 5.3.6. We used curl to test the goodput through the socat-PP.

```
server$ sudo ./protocol_proxy.py -c config/protocol-proxy.cfg -p pmu server 10.1.0.1 9001

server$ sudo socat -d -d TCP-LISTEN:9001,reuseaddr TUN:10.11.0.1/24,up,tun-name=tun1

server$ python3 ./http_server.py


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath off

client$ sudo ./protocol_proxy.py -c ./config/protocol-proxy.cfg -p pmu client 10.1.0.2 9001

client$ sudo socat TCP:127.0.0.1:9001 TUN:10.11.0.6/24,up,tun-name=tun1

client$ time curl -X POST --data-binary ”@500B.data” 10.11.0.1:8080/store.data
```

Listing 5.7: Commands to run the baseline socat-PP throughput experiment.

5.3.8 Baseline mpTCP

Figure 5.11 shows the experiment to calculate the baseline goodput through a mpTCP connection. In this experiment, we calculate the goodput of two TCP streams between the client and the server.

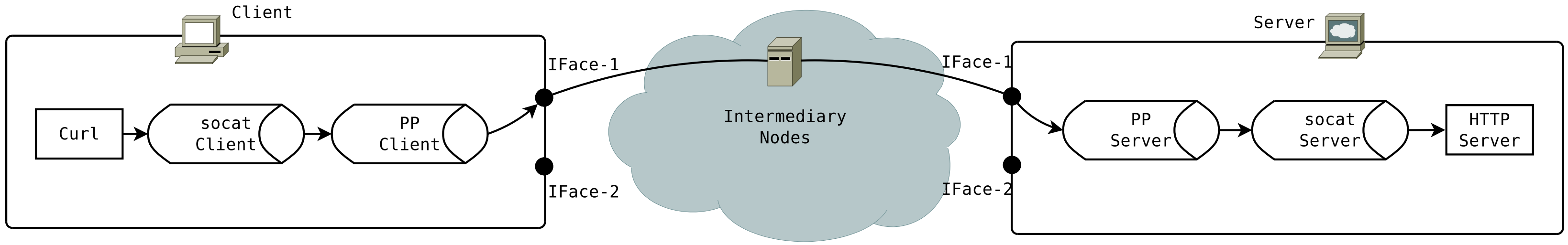

Figure 5.10: Baseline socat-PP throughput experiment.

As shown in Listing 5.8, we repeat the commands in Section 5.3.1 but we enable mpTCP on the external interfaces eth1 and eth2.

```
server$ iperf -s

client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath on
client$ sudo ip link set dev eth2 multipath on
client$ iperf -c 10.1.0.2 -t 1
```

Listing 5.8: Commands to run the baseline mpTCP throughput experiment.

5.3.9 mpTCP VPN

In order to assess the performance mpTCP offers over a traditional VPN, we use the architecture shown in Figure 5.12 to split a single TCP stream over two separate VPN tunnels using mpTCP.

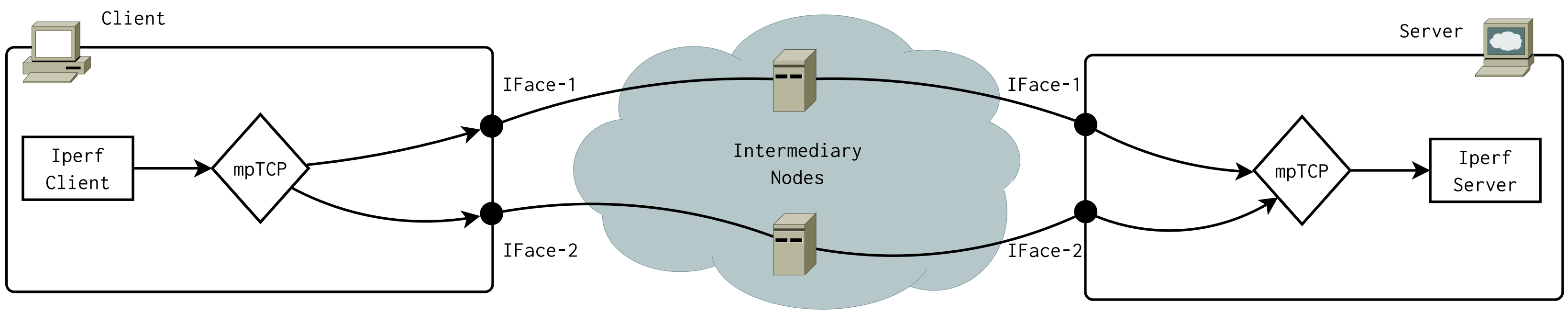

Figure 5.11: Baseline mpTCP throughput experiment.

In Listing 5.9, two separate OpenVPN servers are started. On the client, we set up two separate OpenVPN tunnels. Then, we disable mpTCP on every interface except tun1 and tun2 (the OpenVPN tunnels). We start the Iperf client, and mpTCP splits the TCP stream across each OpenVPN connection and effectively increases the throughput.

```
server$ sudo openvpn --config openvpn-server/server-1.conf
server$ sudo openvpn --config openvpn-server/server-2.conf

server$ iperf -s


server$ sudo openvpn --config openvpn-server/client-1.2.conf
server$ sudo openvpn --config openvpn-server/client-2.2.conf

client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath on
client$ sudo ip link set dev tun2 multipath on

client$ iperf -c 10.11.0.1 -t 15
```

Listing 5.9: Commands to run the mpTCP VPN throughput experiment.

5.3.10 Multisession (mpTCP VPN-PT)

Testing the goodput through multiple VPN-PTs is the next logical progression shown in Figure 5.13.

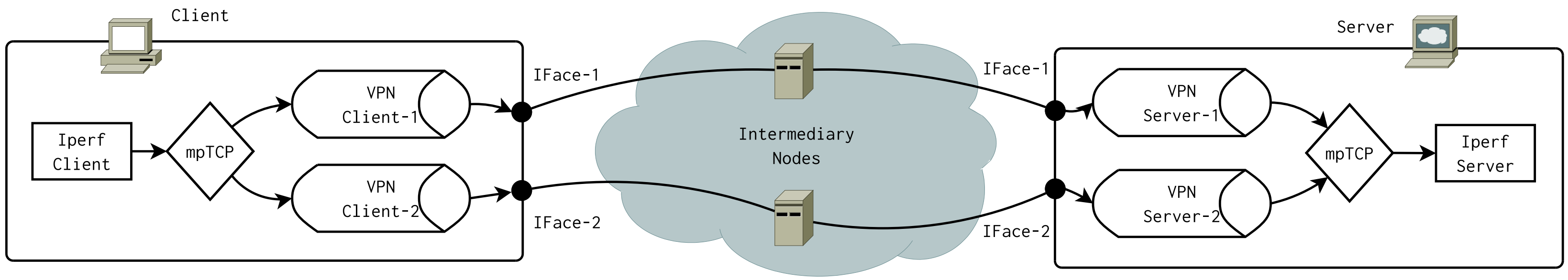

Figure 5.12: Testing mpTCP-VPN throughput.

From Listing 5.10, we start two obfs4 PT servers and setup two OpenVPN servers to forward traffic to each. On the client, we set up the corresponding PT clients and configure them to connect to the PT server and listen on port 1191 and 1192 for the OpenVPN clients. Then, we start the OpenVPN clients, and they connect to the server through the PTs. Next, mpTCP is disabled on all interfaces except tun1 and tun2. When we start the Iperf client, the TCP stream is split between tun1 and tun2, increasing the effective goodput through the obfuscated VPN.

```
server$ sudo openvpn --config openvpn-server/server-1.conf
server$ sudo openvpn --config openvpn-server/server-2.conf

sudo shapeshifter-dispatcher -server -transparent -ptversion 2
  -transports obfs4 -state state -bindaddr obfs4-10.3.0.1:1190
  -orport 10.3.0.1:1194 -logLevel DEBUG -enableLogging
sudo shapeshifter-dispatcher -server -transparent -ptversion 2
  -transports obfs4 -state state -bindaddr obfs4-10.4.0.1:1190
  -orport 10.4.0.1:1194 -logLevel DEBUG -enableLogging

server$ iperf -s


client$ shapeshifter-dispatcher -client -transparent -ptversion 2
  -transports obfs4 -proxylistenaddr 127.0.0.1:1191 -state state
  -target 10.1.0.2:1190  -options
  '{"cert": "<FINGERPRINT>", "iat-mode": "0"}'
  -logLevel DEBUG -enableLogging
client$ shapeshifter-dispatcher -client -transparent -ptversion 2
  -transports obfs4 -proxylistenaddr 127.0.0.1:1192 -state state
  -target 10.2.0.2:1190  -options
  '{"cert": "<FINGERPRINT>", "iat-mode": "0"}'
  -logLevel DEBUG -enableLogging

client$ sudo openvpn --config openvpn-server/client-1.conf
```

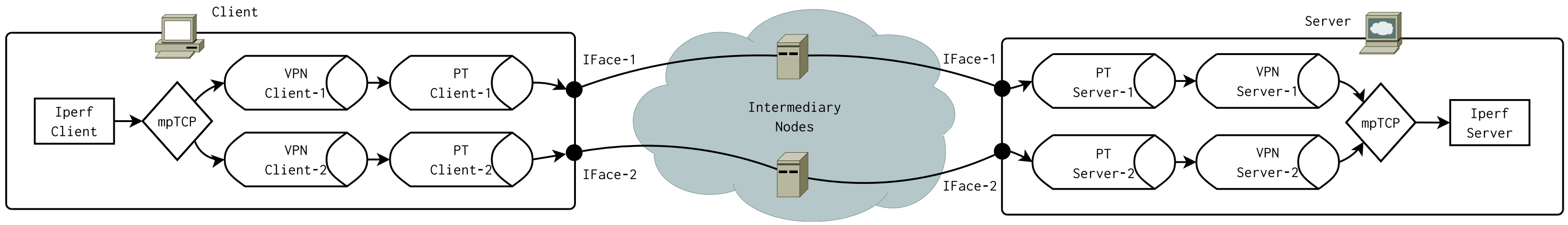

Figure 5.13: Testing mpTCP-VPN-PT ("multisession") throughput.

```
client$ sudo openvpn --config openvpn-server/client-2.conf

client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off

client$ iperf -c 10.11.0.1 -t 15
```

Listing 5.10: Commands to run the mpTCP VPN-PT throughput experiment.

5.3.11 mpTCP Socat

To assess the performance mpTCP offers over a single socat connection, we use the architecture shown in Figure 5.14 to split a single TCP stream over two separate socat tunnels using mpTCP.

In Listing 5.11, two separate socat servers are started. On the client, we set up two separate socat tunnels. Then, we disable mpTCP on every interface except tun1 and tun2 (the socat tunnels). We start the Iperf client, and mpTCP splits the TCP stream across each socat connection and effectively increases the throughput.

```
server$ sudo socat -d -d TCP-LISTEN:9001,reuseaddr TUN:10.11.0.1/24,up,tun-name=tun1
server$ sudo socat -d -d TCP-LISTEN:9002,reuseaddr TUN:10.12.0.1/24,up,tun-name=tun2

server$ iperf -s


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath on
client$ sudo ip link set dev tun2 multipath on

client$ sudo socat TCP:10.1.0.2:9001 TUN:10.11.0.6/24,up,tun-name=tun1
client$ sudo socat TCP:10.2.0.2:9002 TUN:10.12.0.6/24,up,tun-name=tun2

client$ iperf -c 10.11.0.1 -t 15
```

Listing 5.11: Commands to run the mpTCP socat throughput experiment.

5.3.12 Multisession (mpTCP socat-PP)

Testing the goodput through multiple socat-PPs an extension of the mpTCP VPN-PT architecture. The experiment is shown in Figure 5.15.

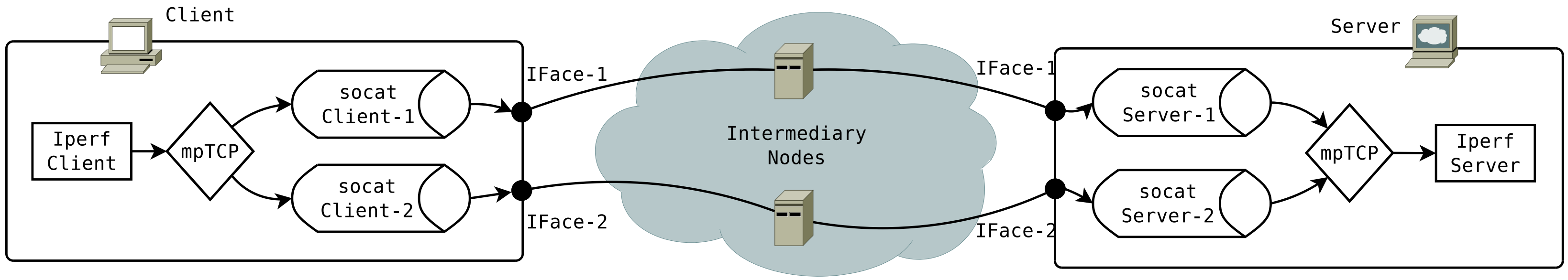

Figure 5.14: Testing mpTCP-socat throughput.

From Listing 5.12, we start two PP servers and setup two socat servers to forward traffic to each. The second PP server uses a different config file, which allows it to send the *host protocol* traffic over a different port to avoid confusion. On the client, we set up the corresponding PP clients and configure them to connect to the PP server and listen on port 9001 and 9002 for the socat clients. Then, we start the socat clients, and they connect to the server through the PPs. Next, mpTCP is disabled on all interfaces except tun1 and tun2. When we curl the file to the server, the TCP stream is split between tun1 and tun2, increasing the effective goodput through the obfuscated socat tunnel.

```
server$ sudo socat -d -d TCP-LISTEN:9001,reuseaddr TUN:10.11.0.1/24,up,tun-name=tun1
server$ sudo socat -d -d TCP-LISTEN:9002,reuseaddr TUN:10.12.0.1/24,up,tun-name=tun2

server$ sudo ./protocol_proxy.py -c config/protocol-proxy.cfg -p pmu server 10.1.0.1 9001
server$ sudo ./protocol_proxy.py -c config/protocol-proxy-2.cfg -p pmu server 10.2.0.1 9002

server$ python3 ./http_server.py


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath off
client$ sudo ip link set dev eth2 multipath off
client$ sudo ip link set dev tun1 multipath on
client$ sudo ip link set dev tun2 multipath on

client$ sudo ./protocol_proxy.py -c ./config/protocol-proxy.cfg -p pmu client 10.1.0.2 9001
client$ sudo ./protocol_proxy.py -c ./config/protocol-proxy-2.cfg -p pmu client 10.2.0.2 9002

client$ sudo socat TCP:127.0.0.1:9001 TUN:10.11.0.6/24,up,tun-name=tun1
client$ sudo socat TCP:127.0.0.1:9002 TUN:10.12.0.6/24,up,tun-name=tun2

client$ time curl -X POST --data-binary "@500B.data" 10.11.0.1:8080/store.data
```

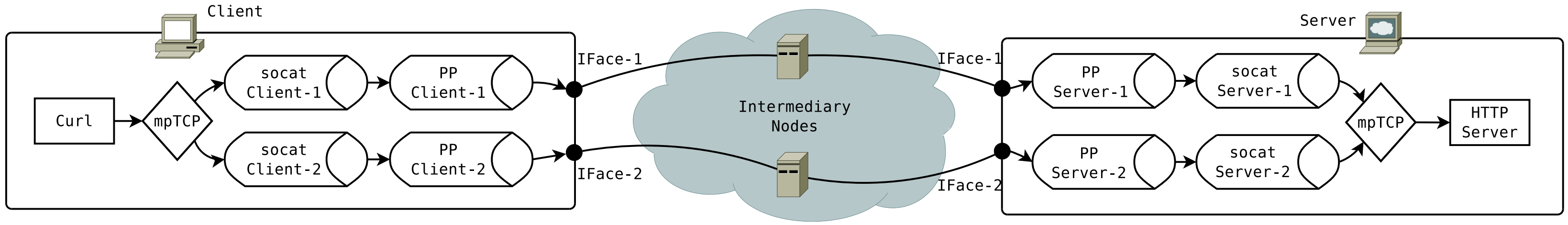

Figure 5.15: Testing mpTCP-socat-PP (“multisession”) throughput.

Listing 5.12: Commands to run the mpTCP socat-PP throughput experiment.

5.3.13 Multipath (PT mpTCP)

An alternative to the architecture in Section 5.3.10 is a single PT split across multiple paths as shown in Figure 5.16.

From Listing 5.13, we start the PT server, and we enable mpTCP on eth1 and eth2 on the client. When we start the PT client, mpTCP splits the connection over both eth1 and eth2. Running Iperf on the client realizes this effective increase in throughput.

```
server$ iperf -s

sudo shapeshifter-dispatcher -server -transparent -ptversion 2
  -transports obfs4 -state state -bindaddr obfs4-10.3.0.1:1190
  -orport 10.3.0.1:5001 -logLevel DEBUG -enableLogging


client$ sudo ip link set dev eth0 multipath off
client$ sudo ip link set dev eth1 multipath on
client$ sudo ip link set dev eth2 multipath on


client$ shapeshifter-dispatcher -client -transparent -ptversion 2
  -transports obfs4 -proxylistenaddr 127.0.0.1:5001 -state state
  -target 10.1.0.2:1190  -options
  '{"cert": "<FINGERPRINT>", "iat-mode": "0"}'
  -logLevel DEBUG -enableLogging

client$ iperf -c 127.0.0.1 -t 15
```

Listing 5.13: Commands to run the baseline PT mpTCP throughput experiment.

5.3.14 Results

The baseline goodput results are shown in Tables 5.2. Given our GENI architecture, the upper bound on goodput was approximately 96.6 Mbps. Interestingly, the PT by itself did not introduce significant overhead, as the goodput was still 96.6 Mbps. However, the introduction of a VPN reduced the goodput by 12.7%. The introduction of an obfuscated VPN (VPN-PT) reduced throughput even more substantially (37.1%). Introducing mpTCP improved the baseline goodput

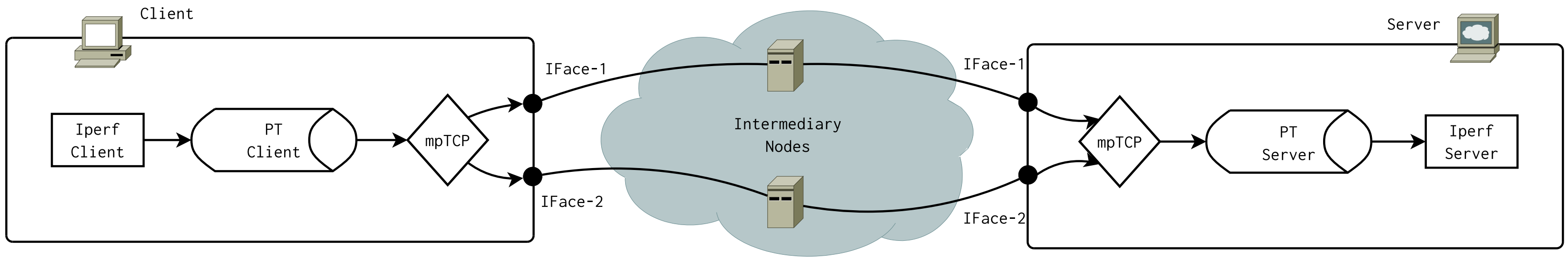

Figure 5.16: Testing PT-mpTCP (“multipath”) throughput.

Table 5.2: Baseline throughput results.

Use-Case	Goodput (Mbps)
baseline	96.6
VPN	84.3
PT	96.6
VPN-PT	60.8
mpTCP baseline	109

Table 5.3: Baseline Protocol Proxy throughput results.

Use-Case	Goodput (bps)
PP	104.1
socat	468,000
socat-PP	26.7

by 12.8%.

Table 5.3 shows the baseline throughput results for the Protocol Proxy. The Protocol Proxy goodput was limited to 104.1 bps, and obfuscating a socat tunnel with the PMU Protocol Proxy reduced the goodput to 26.7 bps. The goodput of the socat tunnel alone was 468 kbps, which is substantially lower than the baseline throughput.

The multipath (PT mpTCP) use-case provided improved goodput over the PT use-case (105 Mbps vs. 96.6 Mbps) for an increase of 8.7%. Adding a second VPN connection improved the goodput over a single VPN connection (110 Mbps vs. 84.3 Mbps) for an increase of 30.5%. It follows that the multisession (mpTCP VPN-PT) use-case also offered a significant increase in goodput (69.0 Mbps vs. 60.8 Mbps) for an increase of 13.5%.

The multisession architecture improved both the socat tunnel obfuscated with the PMU Protocol Proxy. The multisession socat channel increased goodput by 78.6%, and the multisession socat-PP tunnel increased goodput by 22.8% over the PMU Protocol Proxy.

In both the multipath and multisession use-cases, mpTCP improved the effective goodput. In the case of multisession, the baseline goodput better than the obfuscated VPN, but mpTCP improved this with independent tunnels. In the multipath use-case, the goodput also improved

Table 5.4: Experimental throughput results.

Use-Case	Goodput (Mbps)	Change
Multipath (PT mpTCP)	105	+8.7%
mpTCP VPN	110	+30.5%
Multisession (mpTCP VPN-PT)	69.0	+13.5%

Table 5.5: Experimental Protocol Proxy throughput results.

Use-Case	Goodput (bps)	Change
mpTCP socat	836,000	+78.6%
Multisession (mpTCP socat-PP)	32.8	+22.8%

beyond the baseline PT goodput. Through the baseline experiments, it appears the upper bound for outgoing traffic is approximately 110 Mbps. This limitation is likely due to the limitations of generating traffic locally. Since the single baseline throughput is 96.6 Mbps, and the baseline mpTCP throughput is only 109 Mbps, we can conclude another limiting factor beyond the packet network interface. Due to the nature of GENI, there may have been other complicating factors that limited throughput.

While the multipath use-case provided a higher goodput than the multisession use-case (105 Mbps vs. 69.0 Mbps), there are additional considerations. One major disadvantage of the multipath use-case is the mpTCP headers. The mpTCP protocol broadcasts headers containing the server's IP address over all mpTCP-enabled interfaces. This broadcast makes the protocol easy to fingerprint, and it can also reveal information about the bridge node depending on the configuration of the intermediary nodes. However, since mpTCP is a legitimate protocol and actively used for research, this risk is mitigated in certain circumstances. The multisession use-case avoids this entirely by tunneling each mpTCP flow through independent covert channels. The VPN encapsulates the mpTCP header, and the PT obfuscates the VPN traffic. Any observer would only see PT traffic originating from the client. The multipath use-case also requires a TCP-based PT. Since the PMU Protocol Proxy is UDP-based, the mpTCP kernel module is unable to split the session. Since some PT development is moving towards UDP, this is an important consideration.

The multisession architecture improved the performance of both obfs4 and the PMU Protocol Proxy, but on very different scales. Obfs4's is avoiding firewalls, not detection. The PMU Protocol Proxy trades off goodput for a very low probability of detection. The resulting goodput is unusable in most circumstances. However, it serves the purpose of a very low probability of detection transport. Using the socat tunnel introduced additional overhead and decreased the goodput. In reality, such a tunnel is often necessary, as it may be desirable to tunnel multiple applications or protocols through a covert channel, so it is often a necessary sacrifice. The multisession architecture improves this goodput by 22.8%. Downloading a one kB file would be 1 minute faster using the multisession architecture. Since files are larger than one kB, the aggregate improvement is substantial.

Chapter 6

Conclusions

The protocol proxy illustrates the trade-off between enhanced anonymity and goodput. Our observed goodput is on the order of 200 bits per second, while typical users expect megabits per second. We note that our observed goodput differs from the theoretical goodput, and it is likely due to implementation choices in the Protocol Proxy. There are areas for improvement that would increase the covert channel's capacity, but even the theoretical capacity is far below what most users would consider tolerable. Therefore, the Protocol Proxy is most applicable in extreme conditions when detection could have critical consequences.

In order to address goodput concerns, we presented two novel architectures for improving PT goodput. The multipath (PT mpTCP) architecture tunnels a single PT connection through multiple paths using the mpTCP protocol, which provided an 8.7% increase in goodput over a lone obfs4 PT. The multisession (mpTCP VPN-PT) architecture tunnels a TCP stream through multiple independent VPNs, each obfuscated with their own obfs4 PT. This approach provides significantly more obfuscation since the client will appear to be running multiple PTs. The multisession approach afforded a 13.5% increase in goodput and a lower overall goodput than the multipath approach (69 Mbps vs. 105 Mbps). The multisession PMU Protocol Proxy results mirrored the multisession obfs4 results. The multipath (PT mpTCP) architecture was not possible as the PMU Protocol Proxy is UDP-based, but the multisession (mpTCP socat-PP) architecture resulted in a 22.8% goodput improvement over the obfuscated socat tunnel (socat-PP).

Both approaches have ideal use-cases. Without the OpenVPN overhead, the multipath use-case has significantly more goodput overall. However, depending on the network traffic analyzer in question, the traffic may appear as split-routing traffic or mpTCP traffic. The former is much harder to run detection against, and the latter may not be suspicious in all circumstances. MpTCP will likely become much more common in the future, and this may be even more desirable. The multisession use-case has lower goodput than the baseline, but it affords more obfuscation by allowing the client to run multiple independent PTs. When a PT is identified or disconnected, mpTCP will resume using the remaining PTs. The multipath use-case does this as well, but the result here manifests itself as the client using $N - 1$ independent PTs.

These approaches to increase goodput address one of the critical issues surrounding PT development: the trade-off between detection probability and goodput. It is nearly impossible to detect a single bit. It is much easier to detect several trillion. The multipath and multisession use-cases provide viable alternatives for PTs that are traditionally low goodput, such as the PMU Protocol Proxy. These techniques drastically improve the usability and reduce the download times of a one kB file by up to a minute. Improving usability is key to adoption [56], and the multisession PMU Protocol Proxy provides improved goodput while maintaining the same low probability of detection.

Appendices

Appendix A Protocol Proxy Code

```
#!/usr/bin/env python3
""" Protocol Proxy

This file implements the threads necessary to transform network traffic into a
different protocol.

Read in a protocol and HMM

Thread 1: Generate HMM timings

Thread 2: Filter incoming packets to be forwarded

Thread 3: Encoding incoming packets to be forwarded

Thread 4: Generate placholder packets to be forwarded

Thread 5: Forward UDP packets

Thread 6: Receive incoming UDP packets

Thread 7: Decoding the incoming UDP packets

Thread 8: Forward the resulting TCP packets


 Author:
     Jon Oakley

 File:
     protocol_proxy.py

 Date:
     2017-06-22 Version 0.0
     2019-06-15 Version 1.0
 """

import signal
import multiprocessing as mp
import configparser
import argparse
import socket
import time
import sys
import os
import psutil
import scapy.all as scapy
# pylint: disable=wrong-import-position
# pylint: disable=import-error
# pylint: disable=no-name-in-module
sys.path.append('./src')
import iputils
from hmm import HMM
from encoder import Encoder
from decoder import Decoder
# pylint: enable=wrong-import-position
# pylint: enable=import-error
# pylint: enable=no-name-in-module
# pylint: disable=no-member

## Number of timings in the Q
Q_THRESHOLD = 100
```

```
## Seconds to sleep after Q is full
TIMING_SLEEP = .005
## Global threads
PROCESSES = []

def generate_timings(hmm_model, timing_q):
    """Thread 1: Generate HMM timings

    Use a HMM to generate interpacket delay timings according to the host
    protocol. Runs in an infinite loop, constantly making sure there are
    q_threshold timings in

    Args:
        hmm_model (HMM): an HMM model that has been inferred
        timing_q (queue): A queue to hold the generated timings
    """
    while True:
        if timing_q.qsize() >= Q_THRESHOLD:
            time.sleep(TIMING_SLEEP)
        else:
            val = hmm_model.get_observation()
            timing_q.put(val)
            # print val


def filter_packets(incoming_tcp_q, duplicate_packets):
    """Function to use on filtered packets

    This structure is used so that additional arguments can be passed
    to the function

    Args:
        incoming_tcp_q (queue): A queue that holds incoming TCP packets
        duplicate_packets(queue): A queue to holds duplicate packets
    """
    def send_filtered_packets(packet):
        data = bytes(packet['TCP'])
        if not data in duplicate_packets:
            print("New Packet!")
            incoming_tcp_q.put(data)
            duplicate_packets.append(data)
        else:
            duplicate_packets.remove(data)

    return send_filtered_packets


def receive_tcp_data(fwd_port, direction, incoming_tcp_q):
    """Thread 2: Filter incoming packets to be forwarded

    Scapy is used to filter incoming packets and apply the filter_packets
    function to all of the packets that match the filter.

    Args:
        fwd_port (int): The port to forward traffic to/from
        direction (string): If TCP port is 'src' or 'dst'
        incoming_tcp_q (queue): the queue to store incoming packets in
    """
    duplicate_packets = []
    filt = "host 127.0.0.1 and ( tcp {} port {} )".format(direction, fwd_port)
    print(filt)
    scapy.sniff(filter=filt,
                prn=filter_packets(incoming_tcp_q, duplicate_packets), iface="lo")
```

```python


def encode_tcp_data(incoming_tcp_q, encode_q, enc):
    """Thread 3: Encoding incoming packets to be forwarded

    Incoming packets are encoded using FTE

    Args:
        incoming_tcp_q (queue): the queue to store incoming packets in
        encode_q (queue): the queue to store encoded payloads
        enc (Encoder): The FTE encoder
    """

    while True:
        if incoming_tcp_q.qsize() > 0:
            b_packet = incoming_tcp_q.get()
            data = enc.encode(b_packet)
            encode_q.put(data)


def generate_placeholders(placeholder_q, enc):
    """Thread 4: Generate placholder packets to be forwarded

    This process creates placholder packets that can be sent to
    maintain uniform timing

    Args:
        placeholder_q (queue): the queue to store placeholder payloads
        enc (Encoder): The FTE encoder
    """
    while True:
        if placeholder_q.qsize() >= Q_THRESHOLD:
            time.sleep(TIMING_SLEEP)
        else:
            placeholder_q.put(enc.encode_placeholder())

# pylint: disable=too-many-arguments
def send_udp_data(timing_q, encode_q, placeholder_q, udp_send_socket, fwd_addr,
                  use_timing):
    """Thread 5: Forward UDP packets

    The main forwarding function.  This process waits for a given time (specified
    by the time in the timing_q) and either sends an encodeed data packet or a
    placeholder packet using the UDP)socket to the fwd_addr.

    Args:
        timing_q (queue): A queue to hold the generated timings
        encode_q (queue): the queue to store encoded payloads
        placeholder_q (queue): the queue to store placeholder payloads
        udp_send_socket (socket): Outgoing UDP socket
        fwd_addr (tuple): Outgoing UDP socket
        use_timing (bool): Whether or not to send placeholder packets
    """
    psutil.Process(os.getpid()).nice(-19)
    data_l = []
    while True:
        start = time.time()
        timing = timing_q.get()

        # Wait until timing is right
        while timing > (time.time() - start):
            pass

```

```
        # Check for data more data
        if data_l == []:
            if not encode_q.empty():
                data_l = encode_q.get()

        # Check if there is data available
        if data_l != []:
            # Send data
            data = data_l.pop()
            udp_send_socket.sendto(data, fwd_addr)
            print("sending_data")
        elif use_timing:
            # Send junk
            data = placeholder_q.get()
            udp_send_socket.sendto(data, fwd_addr)
        else:
            pass


# pylint: enable=too-many-arguments
def receive_udp_data(recv_addr, incoming_udp_q):
    """Thread 6: Receive incoming UDP packets

    This process simply receives incoming UDP packets

    Args:
        recv_addr (addr): Address that UDP packets are arriving on
        incoming_udp_q (queue): Queue to store incoming UDP packets
    """

    udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    udp.bind(recv_addr)
    while True:
        (data, _) = udp.recvfrom(4096)
        pkt = scapy.Ether(data)
        incoming_udp_q.put(bytes(pkt))


def decode_udp_data(incoming_udp_q, decode_q, dec):
    """Thread 7: Decrypt the incoming UDP packets

    This process decodes the incoming UDP data.

    Args:
        incoming_udp_q (queue): Queue to store incoming UDP packets
        decode_q (queue): Queue to store decroded payloads

    Note:
        Junk data is identified by the three bytes: '#\\x04\\x08'
        at the beginning of the data sequence
    """
    while True:
        if incoming_udp_q.qsize() > 0:
            data = incoming_udp_q.get()

            # Check for junk data
            if not data[0:3] == '#\x04\x08':
                b_packet = dec.decode(data)
                if not b_packet is None:
                    print("received_data")
                    decode_q.put(b_packet)

```

```python

def send_tcp_data(decode_q, mode, fwd_port):
    """Thread 8: Forward the resulting TCP packets

    This process sends the decodeed data as spoofed TCP packets using
    scapy.

    Args:
        decode_q (queue): A queue to hold the decoded UDP traffic
    """
    # This socket type is required to inject packets in the 'lo'
    sock = scapy.L3RawSocket(iface="lo")
    while True:
        if decode_q.qsize() > 0:
            b_pkt = decode_q.get()
            pkt = scapy.TCP(b_pkt)
            tcp = scapy.IP(dst='127.0.0.1')/pkt['TCP']

            if mode == iputils.SERVER:
                tcp.dport = int(fwd_port)
            else:
                tcp.sport = int(fwd_port)

            del tcp['TCP'].chksum
            sock.send(tcp)
            print("Sending TCP!")

# pylint: disable=unused-argument
def signal_handler(sig, frame):
    """Signal Handler

    Iterates over all of the running processes and terminates them.

    Args:
        signal (int): Signal to handle
        frame (?): unused?
    """
    for proc in PROCESSES:
        proc.terminate()


# pylint: enable=unused-argument
def main():
    # pylint: disable=too-many-statements
    # pylint: disable=too-many-locals
    """Main

    Main function that starts the threads and closes things when the server
    is stopped.

    """
    # Ensure user is root
    if os.geteuid() != 0:
        print("Must be ROOT!")
        sys.exit()

    parser = argparse.ArgumentParser(description="Protocol Proxy framework.")
    parser.add_argument('--version', action='version', version='%(prog)s 0.3')
    parser.add_argument('-c', '--config', nargs=1, required=False,
                        type=str, dest='config', help='Configuration file',
                        default='config/protocol-proxy.cfg')
    parser.add_argument('--no-timing', required=False, action='store_false',
                        dest='timing', help='Use host protocol timing?',
```

```
                        default=True)
    parser.add_argument('-l', '--local', required=False, default=False,
                        action='store_true', dest='local',
                        help='Local dev mode -- increment host port')
    parser.add_argument('-p', '--proto', nargs=1, required=False, default='PMU',
                        type=lambda s: s.upper(), dest='proto',
                        help='Host protocol to emulate')
    parser.add_argument('mode', choices=[iputils.SERVER, iputils.CLIENT],
                        type=lambda s: s.lower(), help='Client or Server mode')
    parser.add_argument('dest_ip', type=iputils.valid_ip,
                        help='IP to forward traffic to')
    parser.add_argument('fwd_port', type=int, help='Port to forward traffic from')

    # Add config file to command line args
    # Check protocol command line arg against config file
    # Add dev mode

    args = parser.parse_args()
    # Eventually, this should be specified from the command line
    #config_file = 'config/protocol-proxy.cfg'
    config_file = args.config
    # Also specified from the command line (and verified)
    host_proto = args.proto[0]

    # fwd port optional from cmd line (override config file)

    config = configparser.ConfigParser()
    config.read(config_file)

    if not host_proto in config.sections() or host_proto == 'DEFAULT':
        print("Invalid protocol")
        parser.print_help()
        sys.exit()

    ## The default port for the syncrophasor protocol
    # Later configured via config file
    # Rename
    host_port = int(config[host_proto]['port'])

    # UDP Ports client/server are listening to
    client_udp_port = host_port
    # Only for local testing
    if args.local:
        server_udp_port = host_port + 1
    else:
        server_udp_port = host_port

    if args.mode == iputils.CLIENT:
        direction = 'dst'
        fwd_addr = (args.dest_ip, server_udp_port)
        print("Sending traffic to %s on port %d" % fwd_addr)
        recv_addr = ('0.0.0.0', client_udp_port)
        print("Receiving traffic to %s on port %d" % recv_addr)
    elif args.mode == iputils.SERVER:
        direction = 'src'
        fwd_addr = (args.dest_ip, client_udp_port)
        print("Sending traffic to %s on port %d" % fwd_addr)
        recv_addr = ('0.0.0.0', server_udp_port)
        print("Receiving traffic to %s on port %d" % recv_addr)

    # UDP Socket
    udp_send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_send_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```

    # HMM Setup
    # Determine this from protocol config
    proto_config = configparser.ConfigParser()
    proto_config.read(config[host_proto]['config'])
    proto_base = os.path.dirname(os.path.abspath(config[host_proto]['config']))

    hmm_folder = os.path.join(proto_base, proto_config['HMM']['hmm_folder'])
    #hmm_file_list = proto_config['HMM']['hmm_files'].split(',')
    hmm_object = proto_config['HMM']['hmm_object']

    hmm_model = HMM().load_hmm(os.path.join(hmm_folder, hmm_object))
    hmm_model.import_observations(hmm_folder)
    hmm_model.print_txt_graph()

    # Encoder/Decoder Setup
    keyfile = config['DEFAULT']['aes_key']
    protocol_cfg = config[host_proto]['config']

    enc = Encoder(protocol_cfg, keyfile)
    placeholder_enc = Encoder(protocol_cfg, keyfile)
    dec = Decoder(protocol_cfg, keyfile)

    iputils.add_iptables_rules(args.mode, args.fwd_port)

    # Handle interrupt
    signal.signal(signal.SIGINT, signal_handler)
    # Create the processes
    # Try loop catches keyboard interrupts for clean stop
    print('Generating Timings')
    print('Done')
    # Thread 1
    timing_q = mp.Queue()
    timing_p = mp.Process(target=generate_timings,
                          args=(hmm_model, timing_q,))
    timing_p.start()
    PROCESSES.append(timing_p)

    # Thread 2
    incoming_tcp_q = mp.Queue()
    incoming_tcp_p = mp.Process(target=receive_tcp_data,
                                args=(args.fwd_port, direction,
                                      incoming_tcp_q,))
    incoming_tcp_p.start()
    PROCESSES.append(incoming_tcp_p)

    # Thread 3
    encode_q = mp.Queue()
    encode_p = mp.Process(target=encode_tcp_data,
                          args=(incoming_tcp_q, encode_q, enc,))
    encode_p.start()
    PROCESSES.append(encode_p)

    # Thread 4
    placeholder_q = mp.Queue()
    placeholder_p = mp.Process(target=generate_placeholders,
                               args=(placeholder_q, placeholder_enc,))
    placeholder_p.start()
    PROCESSES.append(placeholder_p)

    # Thread 5
    udp_send_p = mp.Process(target=send_udp_data,
```

```
                            args=(timing_q, encode_q, placeholder_q,
                                  udp_send_socket, fwd_addr, args.timing,))
    udp_send_p.start()
    PROCESSES.append(udp_send_p)

    # Thread 6
    incoming_udp_q = mp.Queue()
    incoming_udp_p = mp.Process(target=receive_udp_data,
                                args=(recv_addr, incoming_udp_q,))
    incoming_udp_p.start()
    PROCESSES.append(incoming_udp_p)

    # Thread 7
    decode_q = mp.Queue()
    decode_p = mp.Process(target=decode_udp_data,
                          args=(incoming_udp_q, decode_q, dec,))
    decode_p.start()
    PROCESSES.append(decode_p)

    # Thread 8
    tcp_send_p = mp.Process(target=send_tcp_data,
                            args=(decode_q, args.mode, args.fwd_port,))
    tcp_send_p.start()
    PROCESSES.append(tcp_send_p)

    # Joining threads
    for proc in PROCESSES:
        proc.join()

    iputils.del_iptables_rules(args.mode, args.fwd_port)

    print("Exiting")

if __name__ == '__main__':
    main()
```

Listing 1: The main protocol-proxy code.

```
#!/usr/bin/env python3
""" Encoder Module

These classes allow data to be encoded as a given protocol.
All elements of the protocol are maintained, mainly timing and payload values.

 Author:
     Jon Oakley

 File:
     encoder.py

 Date:
     2017-06-22
     2019-06-15
 """

import sys
import random
import os
import struct
import configparser
from Crypto.Cipher import AES

## Encode data into the payload for any given protocol
```

```python
class Encoder():
    """A class to encode a static network protocol

    This class can take arbitrary data and encode it using FTE as a
    static network protocol.

    Args:
        protocol_cfg (string): absolute path to the protocol config file
        keyfile (string): path to the AES public/private keypair
    """
    def __init__(self, protocol_cfg, keyfile):
        ## Protocol configuraiton options
        config = configparser.ConfigParser()
        config.read(protocol_cfg)
        ## The size of each block
        self.aes_encrypted_block_size = int(config['DEFAULT']['aes_chunk_size'])
        ## The size of each AES chunk (minus seq num size)
        self.aes_chunk_size = int(config['DEFAULT']['aes_chunk_size']) - 4

        with open(keyfile, 'rb') as key:
            ## AES key data
            self.aes_key = key.read()

        ## The AES cipher
        self.cipher = AES.new(self.aes_key, AES.MODE_ECB)
        ## The protocol instance
        sys.path.append(os.path.dirname(protocol_cfg))
        cls = getattr(__import__(config['DEFAULT']['mapper']),
                      config['DEFAULT']['class'])

        # Load protocols from here.
        if config['DEFAULT'].getboolean('load'):
            proto_base = os.path.dirname(protocol_cfg)
            proto_obj = os.path.join(proto_base,
                                     config['DEFAULT']['protocol_object'])
            self.proto = cls().load_protocol(proto_obj)
        else:
            self.proto = cls()
            self.proto.import_protocol(protocol_cfg)
        # self.proto.print_stats()

        # get the number of bits that can be encoded in the protocol
        # payload and leave 8 bits for the sequence number
        ## Number of data bits per protocol payload
        self.binary_chunk_size = self.proto.get_enc_protocol_size() - 8

        print("Encoder Initialized")


    def encode(self, data):
        """Encode

        Encode data using FTE and the defined protocol. Chunks data into
        ECB-sized chunks (with a prepended sequence number. Then encrypts those
        chunks. Each encrypted chunk is then broken up into smaller chunks that
        can fit into a single payload (with a sequence number prepended). These
        smaller chunks are mapped to a payload (Protocol) and returned.

        Args:
            data (string): Binary data of arbitrary length to encode.

        Returns:
            Array of binary payloads
```

```
        """
        # Prepend data length
        data = struct.pack('i', len(data)) + data

        # Breaks the data into chunks to be encrypted
        encrypted_blocks = [block for block in self.encrypt(data)]

        # Converts each encrypted chunk to a binary string
        str_chunks = [self.bin2str(block) for block in encrypted_blocks]

        # Map binary strings to Protocol payloads
        payloads = [payload for payload in self.map_payloads(str_chunks)]

        # Reverse so that list.pop() can be used
        payloads.reverse()

        return payloads


    def encode_placeholder(self):
        """Encode placeholder

        Create a dummy payload that will be ignored because of sequence of
        '11111111' at the beginning.

        Returns:
            Binary payload (with inidicator sequence)
        """

        # Generate random data to transmit
        length = self.binary_chunk_size + 8
        rand_str = ''.join([random.choice(['1', '0']) for x in range(length)])

        # add delimeter
        payload = '11111111' + rand_str[8:]

        return self.proto.map_data(payload)

    @staticmethod
    def chunk(data, step):
        """Chunk

        Chunks data into length-n chunks.

        Args:
            data (iter): Data to chunkify
            length (int): Chunk length

        Returns:
            generator with the next chunk
        """
        for i in range(0, len(data), step):
            yield data[i:i + step]


    def encrypt(self, data):
        """Encrypt

        Encrypt an arbitrary blob of data. Breaks data into chunks, prepends
        sequence number, and encrypts using AES ECB encryption.

        Args:
            data (bytes): b'\x01\x02...'
```

```
        Returns:
            generator object of encrypted chunks
        """
        idx = -1
        for block in self.chunk(data, self.aes_chunk_size):
            idx += 1
            yield self.encrypt_chunk(struct.pack('>I', idx) + block)


    def encrypt_chunk(self, block):
        """Encrypt Chunk

        This function handles the actual encryption. The block is padded to the
        appropriate length and then encrypted.

        Args:
            block (bytes): b'\x01\x02...'

        Returns:
            Encrypted block
        """
        # Calculate number of bytes to pad
        pad_len = (self.aes_encrypted_block_size) - len(block)

        # Pad with random data
        padding = os.urandom(pad_len)

        # encrypt and return the result
        return self.cipher.encrypt(block + padding)

    @staticmethod
    def bin2str(data):
        """bin2str

        This function converts binary data to a string of '1's and '0's. This
        is necessary because the protocol mapping happens at the bit level, and
        this is the easiest way to track bits in Python.

        Args:
            data (bytes): b'\x01\x02...'

        Returns:
            String: '1010..11'
        """
        binstr = ""
        for byte in data:
            binstr += '{:08b}'.format(byte)

        return binstr


    def map_payloads(self, data_strings):
        """Map Payloads

        Maps data strings ('1011...01') to Protocol payloads using the protocol
        class. Sequence numbers are prepended to each sub-chunk (the AES chunk is
        too big to fit in a single payload). The data is mapped to the target
        payload and returned as a generator object.

        Args:
            data (list of bytes): b'\x01\x02...'

```

```
        Returns:
            generator object: contains binary payload for target protocol
        """
        # Iterate through each byte string
        for data in data_strings:
            idx = -1
            for block in self.chunk(data, self.binary_chunk_size):
                # Add sequence number
                idx += 1
                chunk = self.bin2str(bytes([idx])) + block
                # Generate padding
                pad_len = self.proto.get_enc_protocol_size() - len(chunk)
                padding = ''.join([random.choice(['1', '0']) for x in range(
    pad_len)])
                # Map to target protocol
                yield self.proto.map_data(chunk + padding)
```

Listing 2: The protocol-proxy encoder.

```
#!/usr/bin/env python3
""" Decoder Module

These classes allow data to be decoded from  a given protocol.

 Author:
     Jon Oakley

 File:
     decoder.py

 Date:
     2017-06-22
     2019-06-15
 """

import sys
import os
import struct
import configparser
from Crypto.Cipher import AES

# pylint: disable=too-many-instance-attributes
class Decoder():
    """A class to encode a static network protocol

    This class can take arbitrary data and encode it using FTE as a
    static network protocol.

    Args:
        protocol_cfg (string): absolute path to the protocol config file
        keyfile (string): path to the AES public/private keypair
    """
    def __init__(self, protocol_cfg, keyfile):
        ## Config file
        config = configparser.ConfigParser()
        config.read(protocol_cfg)
        ## The size of the AES block after encryption
        self.aes_enc_size = int(config['DEFAULT']['aes_chunk_size'])

        with open(keyfile, 'rb') as key:
            ## AES cipher
            self.cipher = AES.new(key.read(), AES.MODE_ECB)

```

```
        ## The protocol instance
        protocol_path = os.path.dirname(protocol_cfg)
        sys.path.append(protocol_path)
        cls = getattr(__import__(config['DEFAULT']['mapper']),
                      config['DEFAULT']['class'])

        # Load protocols from here.
        if config['DEFAULT'].getboolean('load'):
            proto_base = os.path.dirname(protocol_cfg)
            proto_obj = os.path.join(proto_base,
                                     config['DEFAULT']['protocol_object'])
            self.proto = cls().load_protocol(proto_obj)
        else:
            self.proto = cls()
            self.proto.import_protocol(protocol_cfg)
        # self.proto.print_stats()

        # Initialize Session Variables
        ## The current AES chunk sequence number
        self.aes_chunk_seq = 0
        ## The current payload sequence number
        self.proto_chunk_seq = 0
        ## Data in the binary chunk
        self.str_encrypt_blk = ''
        ## The length of the packet
        self.packet_len = 0
        ## Data in the packet
        self.packet_data = bytes()

        print("Decoder Initialized")


    def reset(self):
        """Reset

        Called whenever a malformed packet is received or it's time to process
        a new packet
        """
        ## The current AES chunk sequence number
        self.aes_chunk_seq = 0
        ## The current payload sequence number
        self.proto_chunk_seq = 0
        ## Data in the binary chunk
        self.str_encrypt_blk = ''
        ## The length of the packet
        self.packet_len = 0
        ## Data in the packet
        self.packet_data = bytes()


    def decode(self, payload):
        """Decode

        Decode data using FTE and the defined protocol. Data is unmapped
        from the host protocol. The payload sequence number is stripped
        and checked to determine if the packet was a placeholder. The data
        is aggregated until an AES block is assembled. The AES block is
        decrypted and the AES sequence number is checked to determine if
        packets arrived out of order. If the AES sequence number is 0,
        the packet length is separated from the decrypted AES block. The
        remaining data is appended to the existing packet data and returned once
        the packet data is longer than the packet length. Before returning the
        packet, any extra data that was appended to make the AES block size is
```

```
        stripped.

        Args:
            payload (bytes): A binary payload from Protocol

        Returns:
            Binary data that was encoded in the payload
        """
        # Get the data encoded in the payload
        payload_seq, data = self.proto.unmap_data(payload)

        if payload_seq is None:
            return None

        # Check to see if the payload seq number is in order
        if not payload_seq == self.proto_chunk_seq:
            self.reset()
            return None

        self.proto_chunk_seq += 1

        # append the new binary data to the existing binary data
        self.str_encrypt_blk += data

        # Check to see if all the data for one AES block has arrived
        # There should be 64B/AES block, therefore, 512 bits
        if len(self.str_encrypt_blk) >= self.aes_enc_size*8:
            # Reset the payload sequence number
            self.proto_chunk_seq = 0
            # Strip any padding
            self.str_encrypt_blk = self.str_encrypt_blk[:self.aes_enc_size*8]
            # Convert the binary string to a byte string
            # Note that random data was sent in order to fill the last payload
            # this data is removed here
            bin_encrypt_blk = self.str2bin(self.str_encrypt_blk)
            # Reset the incoming binary data
            self.str_encrypt_blk = ''
            # Decrypt the data
            decrypt_blk = self.decrypt(bin_encrypt_blk)
            # Set the AES sequence number
            aes_seq = struct.unpack('>I', decrypt_blk[:4])[0]
            data = decrypt_blk[4:]

            # Check to see if the AES chunk is valid
            if not aes_seq == self.aes_chunk_seq:
                self.reset()
                return None

            self.aes_chunk_seq += 1

            # Parse the length of the packet
            if aes_seq == 0:
                self.packet_len = struct.unpack('i', data[:4])[0]
                data = data[4:]

            # Store the packet data
            self.packet_data += data

            # Check to see if all the packet data has arrived
            if len(self.packet_data) >= self.packet_len:
                # Extra data may have been added to fill the last
                # AES block.  This data is removed here
                packet = self.packet_data[:self.packet_len]
```

```
                self.reset()
                return packet

        return None


    def decrypt(self, block):
        """Decrypt

        Decrypt a binary AES block

        Args:
            block (bytes): Encrypted AES block

        Returns:
            Decrypted binary block
        """
        return self.cipher.decrypt(block)


    @staticmethod
    def str2bin(data):
        """str2bin

        This function converts  a string of ’1’s and ’0’s to binary data. This
        is necessary because the protocol mapping happens at the bit level, and
        this is the easiest way to track bits in Python.

        Args:
            data (string): ’1010..11’

        Returns:
            Binary data: b’\x01\x02...’
        """
        return bytes([int(data[i:i+8], 2) for i in range(0, len(data), 8)])
```

Listing 3: The protocol-proxy decoder.

```
#!/usr/bin/env python3
""" Protool Module

This class is used to model a static protocol. Implementaitons of the static
protocol should inherit this class and implement the ’map_data’ and ’unmap_data’
functions.

 Author:
     Jon Oakley

 File:
     protocol.py

 Date:
     2017-06-22 Version 0.0
     2019-06-16 Version 1.0
 """

import os
import math
import configparser
import pickle

class Protocol():
    """A class to represent a static network protocol
```

```

    This class contains the attributes of a protected static protocol.
    Data is transformed in to the protocol using Format Transforming
    Encryption (FTE).

    """
    def __init__(self):
        ## Number of encoded bits that can fit in each field
        self.enc_field_size = []
        ## Number of bytes in the protocol field
        self.field_size = []
        ## Total number of bytes in the protocol
        self.protocol_size = 0
        ## number of observations for each field
        self.num_obs = []
        ## Total number of bits that can be encoded in the payload
        self.enc_protocol_size = 0
        ## multidimensional array of observations for each field
        self.field_obs = []
        # print the protocol's statistics
        #self.print_stats()

    def map_data(self, data):
        """Template function to map data into a protocol's payload

        This function assumes data is a string of enc_protocol_size with '1'
        and '0' characters for example, 0x07 would be the string '00000111'.
        A template is used because the mapping will depend heavily on the specific
        protocol. An implementation of the mapping should be included in the
        protocol's directory.

        Assumptions:
            len(data) == enc_protocol_size

        Args:
            data (string): '1011..10'

        Returns:
            Binary payload
        """


    def unmap_data(self, data):
        """Template function to map a protocol's payload to data

        This function assumes data is a string of enc_protocol_size with '1'
        and '0' characters for example, 0x07 would be the string '00000111'.
        A template is used because the mapping will depend heavily on the specific
        protocol. An implementation of the mapping should be included in the
        protocol's directory.

        Assumptions:
            len(data) == sum(self.field_size)

        Args:
            data (bytes): b'\x01\x02'

        Returns:
            The string of '1's and '0's stored in the payload.
        """


    def get_enc_protocol_size(self):
```

```
        """Gets the information capacity of the protocol

        Returns:
            number of bits that can be encoded in the protocol

        """

        return self.enc_protocol_size


    def print_stats(self):
        """Print Statistics

        Display the various statistics about the protocol

        """

        print(f'Observations: {self.num_obs}')
        print(f'Number of bits: {self.enc_field_size}')
        print(f'Total size: {self.enc_protocol_size}')
        print(f'Number of bytes in each field: {self.field_size}')
        print(f'Total Bytes: {self.protocol_size}')


    def import_protocol(self, protocol_cfg):
        """Import Protocol

        Import the protocol based on it's configuration file

        Args:
            protocol_cfg (string): A path to the protocol config file

        """

        ## Configuration parser
        config = configparser.ConfigParser()
        config.read(protocol_cfg)
        base = os.path.dirname(os.path.abspath(protocol_cfg))

        #file_count = config['section']['value']
        file_count = int(config['DEFAULT']['fields'])

        for idx in range(1, file_count+1):
            field_file = os.path.join(base, f"fields/field{idx}")
            binary_obs = []

            with open(field_file, 'r') as in_file:
                # Read in observations
                obs = in_file.read().splitlines()
                # Number of payload bytes in the field
                num_bytes = len(obs[0].split('+'))
                self.field_size.append(num_bytes)
                # store the total protocol size
                self.protocol_size += num_bytes
                # store the length of the list in an array
                self.num_obs.append(len(obs))
                # Number of bits to represent a base10 number
                enc_field_size = int(math.floor(math.log(len(obs), 2)))
                # store the field capacity
                self.enc_field_size.append(enc_field_size)
                # total amount of data that can be stored with this payload
                self.enc_protocol_size += enc_field_size
                # create an array of all the binary observations
```

```
                for val in obs:
                    # Calculate the binary value of the observation
                    binary_value = bytes([int(x) for x in val.split('+')])
                    # Append observation to list of all observation for this
                    # field
                    binary_obs.append(binary_value)

            # Append the all the observations for this field to
            # list of all other observations
            self.field_obs.append(binary_obs)


    def save_protocol(self, savefile):
        """Save Protocol

        Saves the protocol as a pickle object

        Args:
            savefile (string): Location to save the file

        """

        with open(savefile, 'wb') as out_file:
            pickle.dump(self, out_file)


    @classmethod
    def load_protocol(cls, filename):
        """Load Protocol

        Load the protocol and return the object

        Args:
            filename (string): Path to the protocol's pickle file

        Return:
            A Protocol object

        Notes:
            Untested. May break things.

        """

        with open(filename, 'rb') as in_file:
            obj = pickle.load(in_file)
        return obj
```

Listing 4: The protocol-proxy protocol mapper.

```
#!/usr/bin/env python3


## \package hmm
#  \brief Everything needed to utilize an HMM FSA.
#  \author Jon Oakley
#  \date 06/22/2017
#
#  Read in data into an HMM, advance through states, and generate data
#  from the HMM.
#
#  The term 'Expression' or 'expr' is used to reference the output of a state
    transition.
# The need for this arises from the fact that states are collapsed and output can
```

```
    no
# longer be determined by the last letter of a given state

import random
import pickle
import itertools
import math
from graphviz import Digraph
import sys

class HMM():
    def __init__(self):
        self.hmm_symbols = ''
        self.L = 1
        self.collapse = True
        self.current_state = None
        self.states = {}
        self.alpha = 0.05
        self.observations = {}
        self.expressions = []

    ## Reset the graph for re-inferencing
    def reset(self):
        self.expressions = []
        self.current_state = None
        # Delete old states
        for k in self.states.keys():
            del self.states[k]
        self.states = {}
        self.alpha = 0.05


    ## Infer the HMM from a symbol file
    #
    # \param data_file The file that contains the string
    # \param L The length of the window to use
    # \param merged A dictionary of state subsitutions (when states are deamed
    equal)
    # \param Alpha The confidence value to use for state collapsing
    #
    # \bug May not work for multiple state substitutions ('aa,bb' -> 'aa,bb,cc')
    def infer(self, data_file, L, merged={}, alpha=.05,collapse=True):
        self.hmm_symbols = data_file
        self.L = L
        self.alpha = alpha
        self.collapse = collapse
        with open(self.hmm_symbols,'r') as f:
            for item in f.read().strip().split('\n'):
                self.add_observation(item,L,merged)

        self.print_txt_graph()
        if self.collapse:
            self.collapse_equal_states(merged)

    ## Use an observation to adjust the HMM
    #
    # \param item The observation
    # \param L The window length
    # \param merged The dictionary of state substitutions
    def add_observation(self, item, L, merged):
        # Break up the input sequence into chunks of length L
        str_states = [item[i:i+L] for i in range(0,len(item)-L)]
        for s in str_states:
```

```python
            # Substitute each state if applicable
            sub = s
            expr = s[-1]
            while sub in merged.keys():
                sub = merged[sub]

            # Add the occurence of the state
            self.update_state(sub,expr)

        # This state doesn't actually occur since
        # since it's the state after the last letter
        self.current_state.decrement_occurences()

    ## Incrementing the number of occurences for a given state
    #
    # \param state The state to increment
    # \param expr The label expressed between states
    def update_state(self, state, expr):
        if not expr in self.expressions:
            self.expressions.append(expr)

        # Check for start condition or new state
        # condition
        if self.current_state == None:
            self.states[state] = State(state)
            self.current_state = self.states[state]
            # Increment the number of occurences of the starting letter
            self.current_state.increment_occurences()
        else:
            next_state_keys = self.current_state.get_next_states()

            # Increment the occurences of the next state
            if state in next_state_keys:
                occ,s,expr = self.current_state.next_states[state]
                s.increment_occurences()
                self.current_state.next_states[state] = (occ+1,s,expr)
                self.current_state = s
                return
            elif not state in self.states.keys():
                # Create a new state
                self.states[state] = State(state)

            # Increment the overall occurences
            self.states[state].increment_occurences()
            # Create a new link to the new state
            self.current_state.next_states[state] = (1,self.states[state],expr)
            # Set the current state
            self.current_state = self.states[state]

    ## Merge all the equal states
    #
    # \param merged The dictionary of state substitutions
    def collapse_equal_states(self, merged):
        # Check every combination of states
        for s1,s2 in itertools.combinations(self.states.keys(),2):
            # Check to see if the states have the same distribution
            #if self.check_distribution(self.states[s1],self.states[s2]):
            if self.chi_square_test(self.states[s1],self.states[s2],self.alpha,'state'):
                print('Merge: ' + s1 + ' and ' + s2)
                # Create a new dictionary entry
                s_new= s1+','+s2
                merged[s1] = s_new
```

```
                merged[s2] = s_new
                # Reset the graph an re-infer using the new state substitutions
                self.reset()
                self.infer(self.hmm_symbols, self.L, merged, self.alpha)
                break
    ## Run a chi-squared test on two states
    #
    # \param s1 first state
    # \param s2 second state
    # \param alpha Alpha value to use
    # \param method Either 'expression' or 'state'.  By 'expression' compares the probabilities of a
    # tranisitions. By 'state' compares the two states for equality
    #
    # Uses the formula $\sum_{r,c} \frac{(O_{r,c} - E_{r,c})^2}{E_{r,c}}$
    # where E_{r,c} = \frac{n_r*n_c}{n}$
    # and n_r is total number of occurrences of a state (s1 or s2),
    # n_c is the total number occurences s1 and s2 transition to a
    # given state, and and n is the sum of the occurences of s1 and s2
    @staticmethod
    def chi_square_test(s1,s2,alpha,method):
        # Get a list of the number of occurrences of outgoing transitions
        s1_next = s1.get_next_states()
        s1_occ = [s1.next_states[x][0] for x in s1_next]
        s1_exp = [s1.next_states[x][2] for x in s1_next]

        s2_next = s2.get_next_states()
        s2_occ = [s2.next_states[x][0] for x in s2_next]
        s2_exp = [s2.next_states[x][2] for x in s2_next]

        # Set the identification method
        if method == 'expression':
            s1_id = s1_exp
            s2_id = s2_exp
        elif method == 'state':
            s1_id = s1_next
            s2_id = s2_next

        # List of unique transitions or states
        sym_diff = list(set(s1_id) ^ set(s2_id))

        # Add missing transitions or states
        for s in sym_diff:
            if not s in s1_id:
                s1_id.append(s)
                s1_occ.append(0)
            elif not s in s2_id:
                s2_id.append(s)
                s2_occ.append(0)

        # Sort by the identification method
        s1_z = sorted(zip(s1_id,s1_occ))
        s2_z = sorted(zip(s2_id,s2_occ))

        # Calculate the $X^2$ statistic
        df = len(s1_z)-1

        nr1 = s1.total_occurences
        nr2 = s2.total_occurences
        n = nr1+nr2

        X2 = 0
        for idx in range(len(s1_z)):
```

```
            occ1 = s1_z[idx][1]
            occ2 = s2_z[idx][1]
            nc = occ1 + occ2
            E1 = float(nr1*nc)/float(n)
            E2 = float(nr2*nc)/float(n)
            X2 += (pow(occ1-E1,2)/E1) + (pow(occ2-E2,2)/E2)

        p = chi2.sf(X2,df)

        # Accept the null hypothesis
        if p > alpha or X2 == 0:
            return True
        # Reject the null hypothesis
        else:
            return False

    ## Setup the HMM for proxy use
    #
    # Loads the HMM from a pickle file
    #
    # \param hmm_folder The directory containing the HMM files
    def import_observations(self, hmm_folder):
        # Choose a random state
        self.current_state = self.states[random.choice(list(self.states.keys()))]

        # Assumes observation file exists for each expression in HMM
        for expr in self.expressions:
            self.observations[expr] = []
            with open(hmm_folder + ’/’ + expr,’r’) as f:
                for line in f.readlines():
                    self.observations[expr].append(float(line.strip()))

    def get_observation(self):
        self.current_state,expr = self.current_state.random_state()
        return random.choice(self.observations[expr])

    def set_random_state(self):
        self.current_state = random.choice(self.states.values())

    def get_state(self, state):
        for k in self.states.keys():
            if state in k:
                return self.states[k]
        return None

    def save_hmm(self, savefile):
        with open(savefile, ’wb’) as f:
            pickle.dump(self,f)

    @classmethod
    def load_hmm(cls,filename):
        with open(filename, ’rb’) as f:
            g = pickle.load(f)
        return g

    def print_txt_graph(self):
        for cs in self.states.keys():
            print("State: " + cs + " | " + str(self.states[cs].total_occurences))
            for ns in self.states[cs].get_next_states():
                print(" --> " + ns + " : " + str(self.states[cs].get_prob(ns)))

    def print_dot_graph(self,name):
        dot = Digraph(comment=’HMM’,format=’pdf’)
```

```
        for k in self.states.keys():
            dot.node(k)
        for node in self.states.values():
            for next_node in node.get_next_states():
                occ,s,expr = node.next_states[next_node]
                lab = expr + '(' + str(round(node.get_prob(next_node),3)) + ')'
                #lab = expr + '(' + str(occ) + ')'
                dot.edge(node.name, next_node, label=lab)

        dot.render(name)


class State():
    def __init__(self, name):
        ## Name of the state
        self.name = name
        ## Total number of times the state occurs
        self.total_occurences = 0
        ## Tuple (occ, <state>, expr)
        self.next_states = {}

    def get_next_states(self):
        return self.next_states.keys()

    def get_prob(self, key):
        occ,s,expr = self.next_states[key]
        return float(occ)/float(self.total_occurences)

    def increment_occurences(self):
        self.total_occurences += 1

    def decrement_occurences(self):
        self.total_occurences -= 1

    ## Advance the HMM and get the associated timing
    #
    # \returns A timing value
    #
    # Dartboard approach to choosing a next state.  Create a probability list:
    #
    # [0 ... 0.X ... 0.Y ... 1]
    #
    # Choose a random value: V
    #
    # [0 ... 0.X .. V .. 0.Y ... 1]
    #
    # Choose state associated with probability 0.X
    def random_state(self):
        if self.next_states.keys() == []:
            return None
        # A list of transition probabilities
        prob_range = [0]
        # A list of choices
        choices = []
        # A list of next states
        n_state = []
        # Populate the transition probabilities, choices, and next states
        for k,s in self.next_states.items():
         prob_range.append(self.get_prob(k) + prob_range[-1])
         choices.append(k)
        prob_range[-1] = 1

        val = random.random()
```

```
        for idx in range(0,len(prob_range)):
         if prob_range[idx] <= val <= prob_range[idx + 1]:
             choose = choices[idx]
             return self.next_states[choose][1],self.next_states[choose][2]
```
Listing 5: The protocol-proxy hidden markov model.

Bibliography

[1] chrony. https://chrony.tuxfamily.org/.

[2] Cisco ios netflow. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[3] Freenet. https://freenetproject.org/author/freenet-project-inc.html.

[4] GNU's framework for secure peer-to-peer networking. https://gnunet.org/.

[5] The invisible internet project. https://geti2p.net/en/.

[6] Lantern. https://getlantern.org/en_US/.

[7] Moloch. https://molo.ch/.

[8] Multipath tcp - linux kernel implementation. https://www.multipath-tcp.org/.

[9] Ntp: The network time protocol. http://www.ntp.org/.

[10] obfs4. https://github.com/Yawning/obfs4.

[11] obfsproxy. https://github.com/NullHypothesis/obfsproxy.

[12] Pluggable Transports. https://www.pluggabletransports.info/.

[13] Psiphon. https://www.psiphon3.com/en/index.html.

[14] Snowflake. https://lists.torproject.org/pipermail/tor-dev/2016-January/010310.html.

[15] Suricata. https://suricata-ids.org/.

[16] Tcpdump. https://www.tcpdump.org/.

[17] Tor. https://www.torproject.org/.

[18] Ultrasurf. https://ultrasurf.us/.

[19] Zeek. https://zeek.org/.

[20] *Kolmogorov–Smirnov Test*, pages 283–287. Springer New York, New York, NY, 2008.

[21] Real-time power and intelliigent systems (rtpis) laboratory, 2019. http://rtpis.org/.

[22] Wireshark, 2019. https://www.wireshark.org/.

[23] Mashael AlSabah and Ian Goldberg. Performance and security improvements for Tor: A survey. *ACM Computing Surveys (CSUR)*, 49(2):32, 2016.

[24] Noah Apthorpe, Dillon Reisman, Srikanth Sundaresan, Arvind Narayanan, and Nick Feamster. Spying on the smart home: Privacy attacks and defenses on encrypted IoT traffic. *arXiv preprint arXiv:1708.05044*, 2017.

[25] Nazanin Asadi, Abdolreza Mirzaei, and Ehsan Haghshenas. Creating discriminative models for time series classification and clustering by hmm ensembles. *IEEE transactions on cybernetics*, 46(12):2899–2910, 2016.

[26] Philippe Biondi. Scapy, 2008. https://scapy.net/.

[27] Russell Brandom. Iran blocks encrypted messaging apps amid nationwide protests, 2018.

[28] Richard R Brooks, KC Wang, Lu Yu, Jon Oakley, Anthony Skjellum, Jihad S Obeid, Leslie Lenert, and Carl Worley. Scrybe: A blockchain ledger for clinical trials. In *IEEE Blockchain in Clinical Trials Forum: Whiteboard challenge winner*, 2018.

[29] RR Brooks, Kuang-Ching Wang, Lu Yu, G Barrineau, Q Wang, and Jonathan Oakley. Traffic analysis countermeasures using software-defined internet exchanges. In *2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*, pages 1–6. IEEE, 2018.

[30] RR Brooks, Lu Yu, Yu Fu, Guthrie Cordone, Jon Oakley, and Xingsi Zhong. Using markov models and statistics to learn, extract, fuse, and detect. In *Proceedings of International Symposium on Sensor Networks, Systems and Security: Advances in Computing and Networking with Applications*, page 265. Springer, 2018.

[31] Human Rights Council. Promotion and protection of all human rights, civil, political, economic, social and cultural rights, including the right to development, 2016.

[32] Roger Dingledine. Ten ways to discover Tor bridges. Technical report, Technical Report 2011-10-002, The Tor Project, October 2011. https://research. torproject. org/techreports/tenwaysdiscover-tor-bridges-2 11-1-31. pdf, Tech. Rep., 2011. 4, 10, 2011.

[33] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Format-transforming encryption: More than meets the dpi. *IACR Cryptology ePrint Archive*, 2012:494, 2012.

[34] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 61–72. ACM, 2013.

[35] Kevin P Dyer, Scott E Coull, and Thomas Shrimpton. Marionette: A programmable network traffic obfuscation system. In *USENIX Security Symposium*, pages 367–382, 2015.

[36] Sean R Eddy. Hidden markov models. *Current opinion in structural biology*, 6(3):361–365, 1996.

[37] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[38] Jessica Fridrich. Minimizing the embedding impact in steganography. In *Proceedings of the 8th workshop on Multimedia and security*, pages 2–10. ACM, 2006.

[39] Jessica Fridrich, Miroslav Goljan, and Rui Du. Detecting lsb steganography in color, and gray-scale images. *IEEE multimedia*, 8(4):22–28, 2001.

[40] Christopher Griffin, Richard R Brooks, and Jason Schwier. A hybrid statistical technique for modeling recurrent tracks in a compact set. *IEEE Transactions on Automatic Control*, 56(8):1926–1931, 2011.

[41] Oluwakemi Hambolu, Lu Yu, Jon Oakley, Richard R Brooks, Ujan Mukhopadhyay, and Anthony Skjellum. Provenance threat modeling. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 384–387. IEEE, 2016.

[42] Vahid Heydari, Sun-il Kim, and Seong-Moo Yoo. Scalable anti-censorship framework using moving target defense for web servers. *IEEE Transactions on Information Forensics and Security*, 12(5):1113–1124, 2017.

[43] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.

[44] Internews. New pluggable transport specification version 2.0, draft 2 is out. 2017.

[45] Negar Kiyavash, Farinaz Koushanfar, Todd P Coleman, and Mavis Rodrigues. A timing channel spyware for the csma/ca protocol. *IEEE Transactions on Information Forensics and Security*, 8(3):477–487, 2013.

[46] Hongda Li, Fuqiang Zhang, Lu Yu, Jon Oakley, Hongxin Hu, and Richard R Brooks. Towards efficient traffic monitoring for science dmz with side-channel based traffic winnowing. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 55–58. ACM, 2018.

[47] Chen Lu. Network traffic analysis using stochastic grammars. 2012.

[48] Chen Lu, Jason M Schwier, Ryan M Craven, Lu Yu, Richard R Brooks, and Christopher Griffin. A normalized statistical metric space for hidden markov models. *IEEE transactions on cybernetics*, 43(3):806–819, 2013.

[49] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[50] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 97–108. ACM, 2012.

[51] Alex Moshchuk, Steven D Gribble, and Henry M Levy. Flashproxy: transparently enabling rich web content via remote execution. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 81–93. ACM, 2008.

[52] Ujan Mukhopadhyay, Anthony Skjellum, Oluwakemi Hambolu, Jon Oakley, Lu Yu, and Richard Brooks. A brief survey of cryptocurrency systems. In *2016 14th annual conference on privacy, security and trust (PST)*, pages 745–752. IEEE, 2016.

[53] Jonathan Oakley, Carl Worley, Lu Yu, Richard Brooks, and Anthony Skjellum. Unmasking criminal enterprises: An analysis of bitcoin transactions. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 161–166. IEEE, 2018.

[54] Jonathan Oakley, Lu Yu, Xingsi Zhong, Ganesh Kumar Venayagamoorthy, and Richard Brooks. Protocol proxy: An fte-based covert channel. *Computers & Security*, page 101777, 2020.

[55] Michael O'Dwyer. Data privacy dead after fcc reversal legalizes isp data mining, 2017.

[56] John Palfrey, Harold Roberts, and Ethan Zuckerman. 2007 circumvention landscape report: Methods, uses, and tools. 2009.

[57] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[58] Ruben Rios, Jose A Onieva, and Javier Lopez. Covert communications through network configuration messages. *Computers & Security*, 39:34–46, 2013.

[59] Sabine Schmidt, Wojciech Mazurczyk, Radoslaw Kulesza, Jörg Keller, and Luca Caviglione. Exploiting ip telephony with silence suppression for hidden data transfers. *Computers & Security*, 79:17–32, 2018.

[60] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing around decoys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 85–96. ACM, 2012.

[61] Jason Schwier. Pattern recognition for command and control data systems. 2009.

[62] Jason M Schwier, Richard R Brooks, Christopher Griffin, and S Bukkapatnam. Zero knowledge hidden markov model inference. *Pattern Recognition Letters*, 30(14):1273–1280, 2009.

[63] Adrian Shahbaz. Freedom on the net 2018: The rise of digital authoritarianism. *Washington, DC: Freedom House. Retrieved February*, 28:2019, 2018.

[64] skivvies. Proxied sites configuration, 2013.

[65] Ronald W Smith and Scott G Knight. Predictable three-parameter design of network covert communication systems. *IEEE Transactions on Information Forensics and Security*, 6(1):1–13, 2010.

[66] Nathan Tusing, Jonathan Oakley, Geddings Barrineau, Lu Yu, Kuang-Ching Wang, and Richard R Brooks. Traffic analysis resistant network (tarn) anonymity analysis. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2019.

[67] Bart Vanluyten, Jan C Willems, and Bart De Moor. Equivalence of state representations for hidden markov models. *Systems & Control Letters*, 57(5):410–419, 2008.

[68] Kuang-Ching Wang, Richard R Brooks, Geddings Barrineau, Jonathan Oakley, Lu Yu, and Qing Wang. Internet security liberated via software defined exchanges. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 19–22. ACM, 2018.

[69] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 109–120. ACM, 2012.

[70] Ryan Whitwam. Supposedly non-existent VPN logs help FBI catch internet stalker, 2017.

[71] Brandon Wiley. Dust: A blocking-resistant internet transport protocol. *Technical rep ort. http://blanu. net/Dust. pdf*, 2011.

[72] Philipp Winter and Stefan Lindskog. *How the great firewall of china is blocking tor.* USENIX-The Advanced Computing Systems Association, 2012.

[73] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: A polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224. ACM, 2013.

[74] Carl Worley, Lu Yu, Richard Brooks, Jon Oakley, Anthony Skjellum, Amani Altarawneh, Sai Medury, and Ujan Mukhopadhyay. Scrybe: A second-generation blockchain technology with lightweight mining for secure provenance and related applications. In *Blockchain Cybersecurity, Trust and Privacy*, pages 51–67. Springer, 2020.

[75] Eric Wustrow, Colleen Swanson, and J Alex Halderman. Tapdance: End-to-middle anticensorship without flow blocking. In *USENIX Security Symposium*, pages 159–174, 2014.

[76] Lihong Yao, Xiaochao Zi, Li Pan, and Jianhua Li. A study of on/off timing channel based on packet delay distribution. *Computers & Security*, 28(8):785–794, 2009.

[77] Lu Yu, Oluwakemi Hambolu, Yu Fu, Jon Oakley, and Richard R Brooks. Privacy preserving count statistics. *arXiv preprint arXiv:1910.07020*, 2019.

[78] Lu Yu, Jason M Schwier, Ryan M Craven, Richard R Brooks, and Christopher Griffin. Inferring statistically significant hidden markov models. *IEEE Transactions on Knowledge and Data Engineering*, 25(7):1548–1558, 2013.

[79] Lu Yu, Qing Wang, Geddings Barrineau, Jon Oakley, Richard R Brooks, and Kuang-Ching Wang. Tarn: A sdn-based traffic analysis resistant network architecture. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 91–98. IEEE, 2017.

[80] Xingsi Zhong, Afshin Ahmadi, Richard Brooks, Ganesh Kumar Venayagamoorthy, Lu Yu, and Yu Fu. Side channel analysis of multiple PMU data in electric power systems. In *Power Systems Conference (PSC), 2015 Clemson University*, pages 1–6. IEEE, 2015.

[81] Xingsi Zhong, Yu Fu, Lu Yu, Richard Brooks, and G Kumar Venayagamoorthy. Stealthy malware traffic-not as innocent as it looks. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 110–116. IEEE, 2015.