Exploring the Use of Adaptive Covert Communication Channels

u2053390

Warwick Manufacturing Group
University of Warwick

Supervised by Peter Norris

Abstract

Adaptive covert communication systems are those that adapt to their surrounding environment to improve their covertness and reliability. The majority of current covert system implementations use a single covert channel to communicate. However, this approach leaves covert systems with a single point of failure, and prone to detection when they don't match the environment. In this paper, I propose a framework for an adaptive covert communication system, that can use a variety of covert channels to communicate, so that it can adapt to the environment and recover from failing channels. The framework is evaluated against a set of objectives and is shown to be effective at adapting to the environment and recovering from failing channels when tested in an isolated environment. Throughout the paper, I propose multiple standards for covert communication systems, such as a covert padding technique and a compression technique for covert protocols. While the framework provides a good foundation for adaptive covert communication systems and does make the detection of covert communications harder for an adversary, there is still plenty of work to be done before it can be used effectively in a real-world scenario.
This project aligns with the following CyBok skill **Network Security**.
Keywords: Adaptive Covert Communication Channels, Digital steganography, TCP/IP stack.

Contents

List of Figures

Chapter 1

Introduction

Covert channels are hidden communication methods that are "not intended for information transfer at all" Lampson (1973). They differ from overt channels in that uninvolved parties are unaware of their existence. While encrypted channels protect the content of communication, covert channels prevent the detection of the communication itself. This makes them suitable for exfiltrating data from secure environments, and censorship resistance (Yarochkin et al., 2008), where a warden exists between parties to prevent or monitor communication.

Covert channels can be classified into two main types: covert timing channels and covert storage channels. Covert timing channels allow a process to signal information to another by modulating its use of resources in a way that is observable and interpretable to another process (U.S. Department of Defense, 1985). For example, this may be altering the Inter-packet delay (IPD) between packets in the TCP/IP stack. However, Wendzel and Keller (2012) shows Active Wardens can mitigate these channels by normalizing the IPD. Covert storage channels, on the other hand, exploit storage locations accessible to the receiver. This paper focuses on covert storage channels in the protocols of the TCP/IP stack, due to the absence of blanket mitigation techniques.

Existing covert channel implementations typically target a single channel, such as embedding data into Skype traffic (Archibald and Ghosal, 2015) or reserved fields in the IEEE 802 family of protocols (Wolf, 1989). For optimal effectiveness, covert channels must be tailored to their operating environment. For instance, a covert channel in the IPv6 header would be ineffective in an IPv4-only environment, out-of-place protocols can be easily detected by an IDS (Yarochkin et al., 2008). Adaptive covert channels can address this issue by adapting to their surroundings and evading detection by adversaries, as covert systems are inherently more effective and secure when undetected (Fatayer, 2017). However, the non-stationary nature of network environments (Hood and Ji, 1997) complicates this task.

This paper proposes a framework for adaptive covert channels and evaluates its effectiveness in a simulated environment. The framework employs an algorithm to identify the most suitable covert channel for the current environment and a set of protocols for communication between the sender and receiver. A more comprehensive overview of the objectives of this paper can be found in Project objectives (4).

Chapter 2

Background

2.1 Steganography

Steganography is the practice of concealing information within other information.

An example of where this is used is in the prisoner's problem, outlined by Simmons (1984), where two prisoners, Alice and Bob, are being held in separate cells. The prison warden, Walter, knows that Alice and Bob are likely to try and plot an escape, but Walter cannot prevent their communication until he has proof. Walter tells the prisoners that he will allow them to communicate, but all messages will first be read by him. Alice and Bob must communicate in a way that Walter cannot understand, without raising suspicion of the existence of a hidden message. This is where steganography comes in, Alice and Bob could use a technique to make a "hidden" message, such as the first letter of each line being combined to make a message (stegotext), to ensure that the message read by Walter (cover text) is worded in a way that isn't suspicious.

Two types of Warden exist, a passive warden and an active warden. The passive warden will not alter the content of the message but will attempt to detect and prevent hidden communication in a message, whereas an active warden will alter the message in an attempt to spoil any unknown hidden communications. In the prison example, if Walter was an active warden he might reword the message so the meaning remains, but the implicit message is lost (Qi, 2013).

A good example of an active warden comes from a tale from World War I, where a cablegram was placed on a censors desk reading "Father is dead", the censor was suspicious and thus rewrote it to say "Father is deceased", shortly after a response was received saying "Is Father dead or deceased?" (Kahn, 1973) illustrating that the receiver was no longer certain of the implicit message that had attempted to be communicated. This shows the clear benefits of using an active warden, however, it does come at a cost, the warden must take the time to analyse the message and alter it, which is costly in real-time communication systems.

2.2 Digital Steganography

Digital steganography is a form of steganography that uses digital media as the cover text. The nature of digital media opens up many possibilities for cover texts, for example, a digital image can be used as the cover text, and the stegotext can be hidden in the least significant bits of the image. This is known as Least Significant Bit (LSB) steganography, and is a popular method of digital steganography (Dalia Nashat, 2019).

In a traditional image, this is simply not possible which highlights the effectiveness of using digital media as the cover text. This is not limited to images, it can be applied to any digital media, such as audio, video, and text. This is the focus of this paper, as the cover text will be the TCP/IP stack, and the stegotext will be hidden in the protocols of the stack.

Another benefit to using digital steganography is the ability to use more complex encoding techniques and permit symmetrical encryption to create secure steganographic systems. Hughes (2000) defines this as "A system where an opponent who understands the system, but does not know the key, can obtain no evidence (or even grounds for suspicion) that a communication has taken place. ie: no information about the embedded text can be obtained from knowledge of the stego-system", it is for this reason that encrypted covert channels are harder to detect (Rowland, 1996).

These secure systems are excellent when working in the TCP/IP stack, because of the nature of the protocols, often using random numbers to have "unique" identifiers, these unique fields can hold encrypted stegotext without raising suspicion, as the fields are expected to be random.

2.3 Covert Channels

The TCP/IP stack is a set of protocols that define how devices communicate over the internet.

The protocols have various fields that hold information about a packet and its contents, these fields can be manipulated to hold stegotext, and thus create a covert channel. The fields used in these covert communications are often required for legitimate communications, which makes the detection and prevention of these channels difficult.

In this paper, I will implement three existing "static" covert channels:

- IP Identification Field
- TCP Acknowledgement Field
- ARP Beaconing

2.3.1 IP Identification Field

The IP Identification stores "An identifying value assigned by the sender to aid in assembling the fragments of a datagram [(A unit of data transfer)]" (ISI, 1981), since this value is unique, an adversary cannot determine its validity, and this can be used to embed bits (Shehab et al., 2021), making it a good candidate for our secure steganographic system.

Another benefit is the volume of IPv4 traffic on the internet, as of 2022, 30-40% of end-user traffic is IPv6 (Wilhelm, 2022) which means that the majority of traffic is IPv4. This high frequency of traffic in combination with sixteen encodable bits of the identification field allows for a high bandwidth channel, while still maintaining a high quality of covertness.

The short-comings of this channel are outlined by Touch (2013), where they state "the [identification] field's value is [to be] defined only when fragmentation occurs", this means an active warden could set the identification field to a constant value, thus preventing the use of this channel. This could be avoided by fragmenting the packets, however, fragmented packets are less common and thus would raise suspicion. This is not a concern when dealing with passive wardens, as they will not alter the packets.

2.3.2 TCP Acknowledgement Field

The TCP Acknowledgement field can be exploited using the TCP protocol's three-way handshake. It works because a TCP server will respond to an initial connection request (SYN) that has defined an Initial Sequence Number (ISN), with an acknowledgement (SYN-ACK or SYN-RST (Depending on the status of the port)) that has an acknowledgement number equal to the ISN + 1.

This process can be abused by spoofing the sending address of the client (as the intended covert receiver) and sending a SYN packet with data encoded as the ISN. The server will then respond to the spoofed address (not the original sender) with a SYN-ACK packet that has the data encoded as the acknowledgement number (Rowland, 1996).

By spoofing the address to the recipient, an observing warden will only see communication between the recipient and the server, they will not know the packet originated from the sender. This makes it harder to determine if a covert channel is being used, and which systems are involved.

In addition to this, the TCP protocol is used in a large proportion of internet traffic, and its thirty-two-bit capacity gives this channel a very high bandwidth.

The ISN generator is bound to "a (possibly fictitious) 32-bit clock that increments roughly every 4 microseconds" (Anon, 1981), since the clock is a client-side clock, it is essentially random (and thus its validity is unverifiable) to an observer, for the first communication. However, a warden could track the allocation of ISNs and determine that a covert channel is likely.

This channel also leaves a trail of failed / incomplete handshakes, as a consequence of not performing the actual handshake. This can raise flags in wardens that are

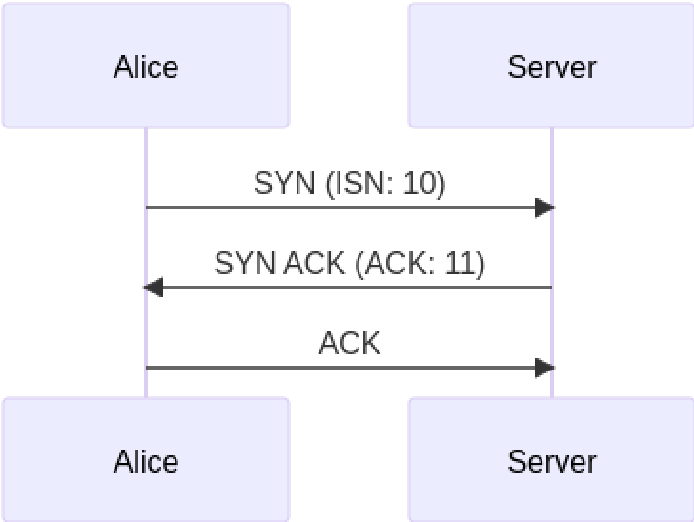

Figure 2.1: Normal TCP three-way handshake

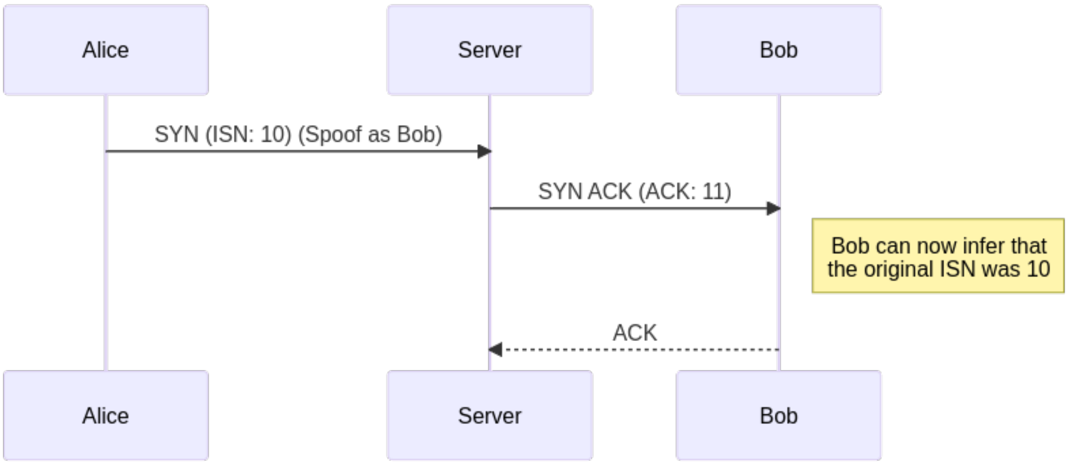

Figure 2.2: TCP three-way handshake with covert channel

monitoring the network, and thus make the channel less covert. As a further note, this channel is only possible when the spoofed address is in the same subnet as the recipient, otherwise, the packet will be dropped leaving the network.

2.3.3 ARP Beaconing

The Address Resolution Protocol (ARP) is used to map IP addresses to MAC addresses of devices, as outlined in Arf (1982). When trying to resolve an address, the resolver must broadcast the request to all hosts on the network, as it doesn't know what its physical address is, and await a response.

This leaves the address, that the resolver is trying to determine the physical address of, available for encoding. A limitation here is that the encoded address must be on the same network, in a /24 subnet, this leaves only a single byte of data available for encoding, reducing the original capacity by 75%.

Requests sent to 'dead' hosts, can raise suspicions in aware wardens, who understand the environment they are operating within (Dua et al., 2021). The redeeming quality of this channel is that, because of the broadcasting nature of the medium, the sender does not need to know the address of the recipient. ARP also play an integral role in network communication and thus is very unlikely to be blocked by a warden.

Chapter 3

Related Work

3.1 Towards Adaptive Covert Communication Systems

Most proposed covert systems are based on a single underlying channel, this is primarily due to the lack of covert systems in academic work, as opposed to proof-of-concepts. However, these channels do not work well in real-world scenarios because they are 'reliant on a particular method' (Yarochkin et al., 2008) which allows them to be easily disrupted and countered by an adversary.

There is a solution to this issue, a covert system that is built upon multiple underlying covert channels, that is adaptable and flexible to the dynamic changes in the network environment (Yarochkin et al., 2008). Yarochkin et al. (2008) proposes a framework that does this, it utilises a group of covert channels and operates in two phases. The first phase is a learning phase that identifies which of the channels apply to the current environment, this phase is continuous to allow the system to adapt to live changes in the environment. The second phase uses these channels to transfer data between the sender and receiver.

This approach to covert systems has a few benefits, as stated it provides redundancy & reliability (Yarochkin et al., 2008) to the communication over a static channel approach. This provides additional protection against communication failures that can arise from interference or preventative measures such as firewalls, active wardens, and protocol normalisation. Not only is this type of system harder to prevent, but it is also harder to detect, its adaption to the network means anomaly-based systems are rendered ineffective, and the irregular and undefined behaviour of the system makes it difficult to detect using signature-based systems (Yarochkin et al., 2008).

These additional benefits come at a cost, additional data to transmit, which can mean additional packets, and thus a lower quality of covertness. There is also however a covertness benefit that comes from the unpredictable nature of stego-objects, which makes it more difficult to distinguish from noise (Zhu et al., 2012).

Unfortunately, the link provided by the paper is dead and unarchived, so the exact details of the implemented system are unknown, meaning I will not be able to build on the codebase of this paper.

3.2 Dynamic Routing in Covert Channel Overlays Based on Control Protocols

In order to effectively communicate with reliability and assured integrity, a protocol is required. Implementing protocols however requires space, which is not in abundance in covert channels. Backs et al. (2012) proposes a dynamic-sized "micro-protocol", these MP's can be dynamic because of how information about packets and payloads is conveyed. Normal network headers, like those defined in ISI (1981) & Anon (1981), are designed to work in a "contextless" way, a single packet describes where it is coming to and from, and information about the payload. This is not the case for covert channels, since there are only two parties involved, they can use a "contextual" approach, where the information regarding a communication channel can be managed by the endpoints of the channel as opposed to the data. This allows the micro protocols to be reduced to context updates, and the majority of payloads can be data only (with a small header to declare that it is data). For static channels, there is no benefit to micro protocols (Backs et al., 2012), due to their unchanging context.

Backs et al. (2012) also proposes a dynamic underlying protocol change, that abstracts the underlying covert channel through the use of an API. this allows for different covert channels to be used with no change to the main logic of the program, this also allows channels to be implemented regardless of classification (Backs et al., 2012), permitting both covert timing channels and covert storage channels.

Chapter 4

Project objectives

This paper seeks to propose and evaluate a framework for an adaptive covert communication system, drawing on prior work in the field, outlined in Background (2) and Related Work (3). The framework will be evaluated against the following objectives, using the terms ***MUST***, ***SHOULD*** and ***MAY*** to describe the importance of each objective, as defined in Bradner (1997):

1. ***MUST*** be able to determine and switch to the best communication channel for the current situation.
2. ***MUST*** be able to detect and recover from failures in communication.
3. ***MUST*** be able to adapt to changes in the environment.
4. ***MUST*** Encrypt and decrypt messages using a shared secret key.
5. ***SHOULD*** be able to recover from the complete failure of a communication channel.
6. ***MAY*** Allow bidirectional communication.
7. ***MAY*** Handle multiple channels at a time.

I will also evaluate the framework against the following non-functional requirements:

1. The effect on the covertness of the framework that comes as a result of its adaptive nature.
2. The applicability of the framework to real-world scenarios.
3. The capability to add new communication channels with reasonable simplicity.

Chapter 5

Research Methodology

5.1 Research goals

The goal of this research is to determine whether an adaptive covert communication system is feasible, and if so, to what extent. This will be done by creating a framework for an adaptive covert communication system. In order to evaluate the framework against the objectives outlined in Project objectives (4), a testing environment is required.

5.2 The testing environment

The requirements of the testing environment are as follows:

1. The environment must allow the proposed framework to run.
2. The environment must be able to be monitored.
3. The environment must mimic a real-world network.
4. The environment must be repeatable.
5. The environment components must be static.
6. The environment must exist on a single machine.
7. The environment must be controlled, and completely isolated from the non-testing network.

Requirements 1-5 are required to ensure that testing is possible, and that test results are repeatable and comparable. The 6th requirement is a product of my limited resources, and the 7th is due to the limitations of my ethical approval.

These requirements can be met by creating a virtual network, many tools allow for this but in order for the framework to be possible the tool must allow for raw socket access.

5.3 Evaluating Docker

Docker (Merkel, 2014) is a tool that allows for the creation of containers inside virtual environments, these containers are lightweight and therefore many can be run on a single machine. this allows for an approach where each function has its container that can be altered as needed. For example, the sender and receiver exist as separate containers, and the traffic generator exists separately from them as well. This makes it easy to restructure and alter the environment as needed.

The shortcomings of Docker however was the lack of ability to capture the traffic of the entire network from inside the environment, it instead had to be done by the host machine. This is not ideal as I'd like to be able to have a completely closed system that can be run in multiple instances to collect data from multiple runs. This would be a useful feature in testing as the covert channels are slow and thus data takes a long time to collect.

Another issue with Docker was the framework didn't work while using it, whereas it did work on my host. Later on in the project, I discovered that the checksum calculations were incorrect, causing the packets to be discarded at the bridge, it is possible that this issue was occurring in Docker, but I had already settled on a different environment at this point.

5.3.1 Network namespaces

Network namespaces are a Linux kernel feature that allows for the creation of virtual network stacks, allowing the simulation of multiple network interfaces. The namespaces came with some other advantages:

- The framework runs on bare metal (without virtualisation), there is no added overhead of starting a virtual machine.
- They exist in the same filesystem as the host, allowing for easy access to the framework source code.
- Adding devices and connecting namespaces is easy and scriptable.

The structure of the namespaces is as follows:

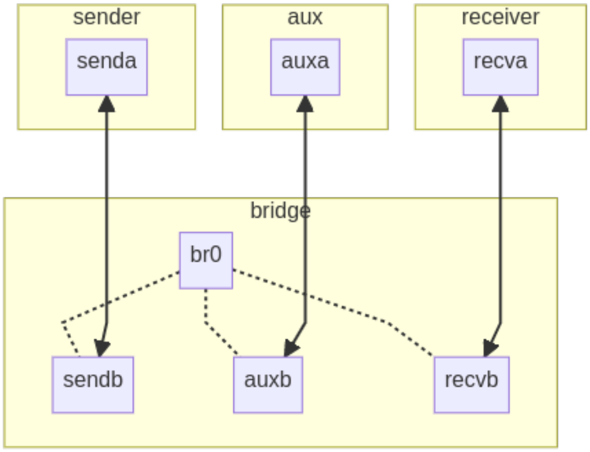

Figure 5.1: Diagram of the network namespaces

In the above diagram, the interfaces *a & *b are virtual ethernet pairs, these are used to connect the namespaces. Only the interfaces that are in their namespace are assigned addresses, the others do not require configuration to work, they are simply used to connect the namespaces.

The reason they are in separate namespaces is to remove ambiguity as to which is the correct interface and route to take, for example, if the sender and receiver were in the same namespace, they would try and route traffic through the loopback interface, which is undesirable behaviour. The use of bridging between the namespaces allows for the traffic to be routed via a central point, which allows for a simple warden to be implemented with access to all the traffic.

A shortcoming of the network namespace is how the traffic moves between namespaces, as a particular namespace can only see traffic that was sent from or to it. This means that the traffic generator must be in the same namespace as the sender, but this is not a problem for a single instance of the framework.

5.3.2 Traffic generation

The traffic generator is responsible for making the environment appear to be a real-world network, for this to also be repeatable the traffic must be generated in a deterministic way. This is done by using tcpdump, a tool that allows packet captures to be replayed via a network interface, allowing a single packet capture (stored as a "pcap" file) to be replayed multiple times.

The next step is then getting a capture of real-world traffic, as per my ethical approval I am not allowed to capture traffic that contains other people's data, so I instead found a public dataset `https://www.first.org/conference/2015/program#phands-on-network-forensics` that contains 24 hours of network traffic, that contains no personal information. It should be noted that the dataset is originally intended for use in forensics, but we aren't looking at the contents of the packets so this is not a concern.

There are a few issues with using this pcap, as the network range differs from the one used in the framework, and addresses of the sender and receiver may also have traffic directed at them, which can cause two responses into the network (one from the capture, and one form the read host), to mitigate these issues the traffic is rebased. Rebasing the pcap requires swapping the start of network addresses to the new range, I could have limited this to only the ethernet header but then some traffic, including ARP, would not work properly, so instead I replaced all instances of the old network address with the new one. This could cause some unusual artefacts in the traffic but it is not a concern for this project. The script used to rebase the pcap can be found in rebase_pcap.jl (B.1), although this script does as intended, it was an oversight to not factor in checksums and thus all the packets in the pcap have the incorrect checksum, so they are processed through a second script that recalculates the checksums using the python library scapy Rohith Raj et al. (2018), this script can be found in fix_checksum.py (B.2).

5.3.3 Version management

Both the framework and the testing environment must remain the same for testing to be repeatable, so maintaining version control is important.

Source control

To ensure testing is repeatable, the source code that is used must remain the same, and the versions of the dependencies must also remain the same. I achieved using git (the Git community, 2023), a version control system that allows for the tracking of changes to files, and the ability to revert to previous versions. Each commit of changes is uniquely identifiable so it is easy to identify which version of the code was used for a particular test.

Since the testing environment is a superset of the framework, the framework is included as a submodule of the testing environment, this points to a particular commit of the framework, so the framework can be updated independently of the testing environment. This also ensured that commits were atomic and incremental.

For a remote backup of code, I used a private repository on Github (GitHub, Inc, 2023), this allowed for the code to be accessed from anywhere, and will allow for the code to be open-sourced in the future.

Dependency management

The dependencies of the project are managed by the Julia package manager, it uses a two-file system, one for the dependencies and constraints of the framework `Project.toml` and one that maintains the exact version used `Manifest.toml`. By keeping the `Manifest.toml` file in the repository, the exact versions of the dependencies are known and can be used when the project is cloned, for a repeatable environment.

External dependencies

A unique situation arose during the writing of this paper, where a major update was released for Julia, however since I had already begun testing on the previous version, I chose to abstain from updating to the new version. There is not any reason that the code would be incompatible with the new version, but the performance improvements of the newer version would make some of the results incomparable.

The other external dependency used is libpcap (The Tcpdump Group, 2013), I could include a copy of the library in the repository, but The Tcpdump Group has a good record of compatibility of its library, and then a system-specific version can be installed. This is done using the system package manager of choice.

5.3.4 Covert detection bias

I am in a unique situation as the author of this paper, where I know the implementation of the covert channel, and thus detecting the channel is an easier task for me than it would be for an adversary. However, as it is a secure steganographic system (Hughes, 2000), my knowledge of the implementation should not allow the detection of the channel with reasonable certainty.

Chapter 6

Design

6.1 The Julia language

I chose to write the project in Julia (Bezanson et al., 2017), A dynamically typed, Just-In-Time compiled language, This allows it to have both the speed of a compiled language and the flexibility of an interpreted one. This made it an excellent choice for my framework as I could quickly iterate on the design, without compromising on the performance of computationally expensive operations like querying the queue (see The Packet Queue (6.4)).

Julia also has excellent integrations with the C language, which allowed me to use the libpcap library (The Tcpdump Group, 2013) inside of a Julia program, this can be seen in action with the use of `ccall` in figure 6.1.

The high-level, dynamic nature of Julia also lends itself to the ease of extending the framework with new covert channels, the intricacies of this process are discussed in Capability to add new communication channels (8.6.3).

6.2 Packet capture

For the framework to adapt to the network environment, it must be able to listen to the network traffic. I concluded that one of two options would be best suited to my needs:

- libpcap
- A raw socket

6.2.1 raw sockets

Raw sockets provide direct access to low-level protocols, allowing a program to read packets down to layer two, generally the Ethernet header.

This approach is standalone; it does not require external libraries to operate, so the program would not need pre-existing libraries installed on the system. This is particularly important in some of the applications of covert channels.

There is very little code complexity involved in setting up a raw socket:

```
# Capture the entire packet (AF -> Address Family)
const AF_PACKET = Cint(17)
# Raw listening socket
const SOCK_RAW = Cint(3)
# Capture all ethernet frames
const ETH_P_ALL = Cint(0x0300)

socket = fdio(ccall(:socket,
    Cint, # Return type
    (Cint, Cint, Cint), # Argument types
    AF_PACKET, SOCK_RAW, ETH_P_ALL # Arguments
))
packet = read(socket)
```

Figure 6.1: Raw socket listener implementation

Julia will read the socket pipe until an EOF is found, which marks the end of a packet; this lightweight and simplistic implementation means the construction of this listener has little cost to system resources, and thus it can be easily applied to a variety of use cases.

6.2.2 libpcap

libpcap is a cross-platform library for low-level network monitoring (The Tcpdump Group, 2013). There are several benefits to using libpcap, the first being the ease of implementation, libpcap provides a simple API for capturing packets, allowing for a callback function to be called when a packet is captured, this function is passed the capture header, packet pointer, and user data if required.

Because of Julia's ability to integrate seamlessly with C and its libraries, the callback function could be written in Julia, removing the need to write any C code. The benefit of using a library is that it is appropriately optimised. libpcap creates a memory-mapped ring buffer for asynchronous packet reception (Free Software Foundation, 1999), this allows the user space to read the packet data without the need for system calls, and the shared buffer also reduces the number of copies required.

While the platform-independent nature of libpcap is beneficial, the scope of this project is limited to Linux to manage the scope, and thus this will not be considered as part of my evaluation.

6.2.3 Conclusion

Both raw sockets and libpcap are viable options for packet capture, however, libpcap has a few advantages, the factor that swayed my decision was the support for the Berkeley Packet Filter (BPF). BPFs are register-based "filter machines" (McCanne and Jacobson, 1992) that make filtering packets incredibly efficient, this is particularly useful for this project as it allows for the filtering of packets to be done in the kernel, which reduces the number of packets that need to be processed by the program. Specifically, this means the receiver can filter traffic so it only has to process packets that are destined for it, reducing the overhead. In some cases, however, we will want to be able to capture packets without disrupting the main queue of packets, and the filters that are put on the queue. In these cases, I will opt to use the raw socket implementation, as its reduced code complexity (compared to searching the queue) will, in turn, reduce the complexity of the codebase. There are some other advantages to using the raw socket, but I will discuss them in context in 7.3.

6.3 Packet processing

When packets are captured, we want to process them into objects so we can idiomatically access their properties. Specifically, we want to process them into a `Packet` object, that holds the capture header, and a `Layer` object. The layer is made up of a `Layer_type` enum, the layer header, and the payload. The payload can be either another `Layer` or a `Vector{UInt8}`.

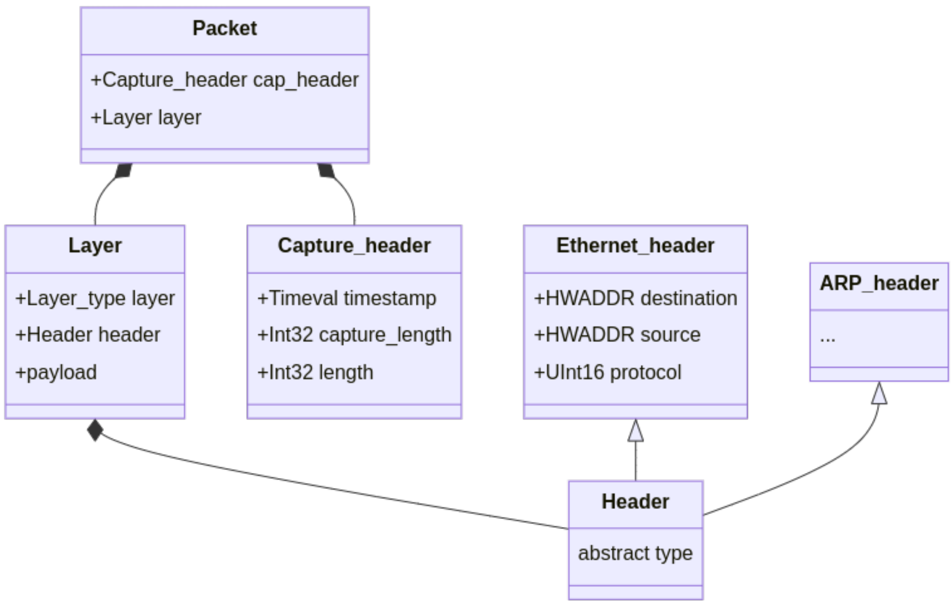

Figure 6.2: Packet structure

A function, `packet_from_pointer` takes the pointer given by libpcap, and the length of the packet. It iteratively processes the packet, creating an `Layer` struct of each layer, and appending to the previous `Layer`, incrementing the `Layer_type` enum as it goes. The headers defined have two helper functions to assist this process, `get_offset` which returns the size of the header, and `get_protocol` which returns the protocol embedded in the header, allowing the next layer to be processed.

The `Header` class is an abstract type, generalising all headers, it should also be noted the packet definition looks like this:

```
struct Packet
    cap_header::Capture_header
    layer::Layer{Ethernet_header}
end
```

To assist this process there is a tree defined that maps the protocol number to the header type, and the protocols in the layer above it. This makes the addition of protocols easy for the user, it is mostly just copying the structure of the protocol from the relevant header file, adding it to the tree, and two simple helper functions that just access fields.

6.4 The Packet Queue

The queue is essential to the operation of the framework, it is how packets that are captured by the listening thread, are retrieved by the main thread for processing.

Initially, I was using a `Channel` data structure, it is a thread-safe, waitable, First-in-First-out queue (Bezanson et al., 2017) that worked for the most part. However, it introduced a bug into my codebase which meant the listening thread would block the main thread while it was waiting for packets, thus the covert channel would not run in a silent environment.

While this wasn't a huge problem, since a covert channel is quite exposed on a silent network, it is not ideal. I eventually deduced that I was improperly accessing the properties of the structure, I was attempting to get the whole queue without removing the items, so I just took it from the underlying array. Unfortunately, the `Channel` is locked by default, and thus the main thread was blocked until the listening thread pushed something to the queue.

While I knew this action was not properly thread-safe, meaning I would likely be operating with stale data, it wasn't of great importance that my queue was truly up-to-date, as I was taking a rolling average of 150 packets anyway. I didn't realise that it would cause the main thread to be intermittently blocked, additionally, this unusual behaviour meant that it was difficult to debug, especially since my print statements were reliant on packets being received by the listening thread, which caused a lot of debugging cycles to be performed.

My solution to this was to implement a new, more idiomatic data structure, the `CircularChannel`. This is a thread-safe, waitable, First-in-First-out queue that behaves like a circular buffer so that it overwrites the oldest item when it is full.

By implementing the `put!` and `take!` functions for my new data structure, I was able to avoid changing the majority of my existing implementation, needing only to

```
packet = get_packet()
if isfull(queue)
    take!(queue)
end
put!(queue, packet)
```

Listing 1: Old "Channel" implementation

```
packet = get_packet()
put!(queue, packet)
```

Listing 2: New "CircularChannel" implementation

replace the type from `Channel` to `CircularChannel`. I also simplified the code for the producer (packet listener) as seen in the comparison above (figure 6.4). The `get_queue_data` function that I was previously using to illegally access the properties of the channel was called everywhere I needed to access the whole queue, so I only had to alter that function to access the `CircularChannel` in a thread-safe manner.

I efficiently achieved this waiting using `Conditions`, which are locks that implement `wait` and `notify` methods that block (reducing resource usage) and release threads, respectively, based on the status of the condition. This allows my main thread to wait for the listening thread to push a packet to the queue, and then continue processing the packet (for the receiver only).

6.4.1 Queriability

One of my goals for the project was to allow any type of covert channel to be used in the framework, regardless of classification. Ultimately, this meant I was going to have to take a dynamic approach to initialising and using channels. I concluded that it was best to be able to query the channel for certain properties in packets, like the destination port or the packet size.

I implemented a rudimentary query system, that allowed for querying of properties captured by the framework (known header values), these properties could be combined with simple logical operators (AND, OR) to sufficiently complex queries. An extract of Faucet's documentation gives an example of what these queries look like:

6.5 The arrangement

Covert communication is reliant on a preshared arrangement (Owens, 2002), this arrangement is a set of parameters that both parties will assume the other is using, hereon called the "target", it contains the following information:

- The IP Address of the target.
- The covert methods to use.
- The AES Pre-shared key.

`Faucet.Environment.query_queue` — Method

```
query_queue(queue, arguments)::Vector{Vector{Header}}
```

Returns a vector of all packet headers if a packet matches an argument

Arguments is a vector of dictionaries, each dictionary is a match case A match case is a dictionary of header types and required values in that header

Example

```
arguments = [
    {
        TCP_header => {
            :sport => 0x2f11 # 12049
            :dport => 0x0050 # 80
        }
    },
    {
        IPv4_header => {
            :ihl => [0x05, 0x06]
        }
    }
]
```

Note

will return all headers for packets that either have: (tcp header with a source port of `12049` *AND* destination port of `80`) *OR* (ipv4 header with an ihl of `5` *OR* `6`)

source

Figure 6.3: A snippet from the Faucet documentation

- The Initial Vector to use.

The IP address is where the sender will send the packets and the address that the receiver will listen on. In theory, this address does not have to be the address of the target, as long as the target can see that endpoint, but I cannot test this due to limitations in the testing environment. The covert methods are the channels that will be used for communication, they are referenced by their index in the array of covert methods, and thus the order of the covert methods is important. The AES pre-shared key is used to encrypt (and subsequently decrypt) the data, and the initial vector is used to initialise the AES encryption, both communication parties must know these values (Fatayer, 2017).

6.6 Sending packets

Sending packets is an important aspect of the framework, as outgoing packets must be completely composable to permit all covert channels. While Julia does have some socket functionality, it is not sufficient for the project as it is too high level, my implementation instead uses a lightweight TCP/IP stack design. The principle is the user can supply the protocols to use, and a dictionary of fields and values for each of the layers, The function will then construct the packet, filling in an informed manner.

For example, in an IP header, source and destination fields will default to the local IP and target's IP respectively, But in a TCP Header, the ISN will be random. If flags are set, but their respective fields are not filled, they will also be randomised. This allows for the user to specify the fields they want to be set, and the rest will be filled in a way that is consistent with the protocol, however, it should be noted that some of these randomised fields do not have uniform distributions, and thus may be more suspicious than others, but this is out of the scope of my project.

Since the filling of these additional fields is done at execution time, a "skeleton" packet can be created that outlines a basic packet, then this skeleton can be used multiple times without having random fields being static, it is also useful with packet checksums, as this burden is not passed to the user. I make use of this functionality in the use of covert modules, as outlined in Design of covert modules (6.8).

The packets themselves are sent over a raw socket, this allows for the packet to be sent without the kernel imposing checks on its validity (such as the checksum), this is important for the operation of some covert channels, as they create packets that would otherwise be dropped by the kernel (sometimes because of firewalls). For a packet to be sent over a raw socket at the link layer (ethernet) it needs to be sent with a `sockaddr_ll` struct, outlined in Free Software Foundation (1999) the majority of these fields are not required, or are unlikely to change (The physical layer protocol) and thus can be hardcoded, only two fields need to be populated dynamically, the interface index and the destination mac address. The interface used is the one that the queue listens on, and the destination MAC can be taken from the packet.

The packets can then be sent to the raw socket using the `sendto` function, taking the socket, packet and `sockaddr_ll` as arguments.

6.7 Providing context to the sender

Many of the sender's functions are reliant on the context of the environment, this environment is constructed as a dictionary of values that are passed to the sender. This dictionary contains the following values:

- The desired secrecy of the channel (User argument).
- The interface to use (see Sending packets (6.6))
- The socket to send packets over.
- Information about the route to the target.
- The queue

I will discuss the desired secrecy argument in Decision algorithm (6.9), and the interface has already been discussed in the construction of the `sockaddr_ll` struct (see

Sending packets (6.6)). The socket is simply a raw socket, it does not carry context and thus could be recreated at each use, but that requires calls to the kernel, and this approach has very little technical overhead.

The information about the target essentially outlines the default path to follow for a target, like the physical address of the target, or the gateway to use to reach the target. while this does not apply to all channels, it is used for many of them, and not worth recalculation for each packet.

The queue is also exposed here, since it accounts for most of the context of the sender, compiling these contextual values into a single dictionary makes it easier to pass around, and allows for the sender to be more idiomatic.

6.8 Design of covert modules

A covert module is a set of functions that are used to implement a covert channel. For the framework to work effectively, the addition of covert modules must be a process that is both simple and idiomatic, but sufficiently flexible to allow for any type of covert channel to be implemented. Since I cannot predict the nature (and thus the requirements) of the covert channels that will be implemented, A dynamic approach is required.

Covert modules are implemented using:

- An `init` function.
- An `encode` function.
- A `decode` function.
- An instance of the `covert_method` struct.

The `init` method is called when the module is loaded, it is used to initialise the channel, and it returns a set of keyword arguments that are to be passed to the `encode` functions. The most common of these arguments will be `template`, which outlines the skeleton and structure of a packet that the covert channel lives in, omitting the fields that are to be used in the channel, as outlined in Sending packets (6.6). This function is passed the queue (see The Packet Queue) as an argument, allowing it to adapt to the environment.

Covert channels can be adapted to the network on a sub-method level to improve covertness, a good example of this is using IPv4 vs IPv6 as the internet protocol, for covert channels that exist in all other layers this does not affect their operation, but does allow them to be better suited to the network. I specifically chose to implement the TCP Acknowledgement bounce covert channel to showcase this flexibility in the framework. By using a local TCP server to bounce the packets, the channel must be aware of the network to discover these hosts. For this to be possible, the `init` function must have the environment exposed to it, this is done by passing the environment dictionary outlined in Providing context to the sender (6.7) to the `init` function.

In the case of the TCP Acknowledgement bounce covert channel, the queue is taken from the environment dictionary, and the `init` function will use this to listen for the destination of TCP packets, and it will return a template that has the destination mac, ip and port set to the local server.

The `encode` function is called for each packet to be encoded, it is passed the keywords arguments from `init` and the payload to put into the packet. The function will then return a full packet template, that can be converted into a vector of bytes, and sent over the network.

The `decode` function is called to extract the payload from a packet, it doesn't require any other arguments as it just returns the field of the packet object that has the payload in it. This function does not need to perform any validity check, as the framework will verify that the packet could contain the payload before calling this function.

The `covert_method` struct holds metadata about a channel, and is comprised of five fields:

- The name of the covert channel.
- The TCP/IP layer it operates on.
- The protocol in that layer it operates on.
- How covert it is, on a scale of 1 - 10.
- The bit capacity of the channel.

These allow the framework to seamlessly integrate with the covert channel, for example, the bit capacity is used for crafting payloads for the channel, removing this burden from the module. The name of the channel is used in The arrangement (6.5) to ensure both parties are using the same channels.

The layer and protocol are used by the receiver side of the framework to filter out packets that are not relevant to the covert channel, so the `decode` function is only called on valid packets, so error handling can be omitted from covert modules unless they are particularly complex.

All of the fields are used to create an informed decision on which covert channel to use, as outlined in Decision algorithm (6.9).

6.8.1 Covert module validation

Covert modules need to be validated before they can be used in the framework, this is done to ensure they are compatible with the preshared arrangement (defined in The arrangement (6.5)). There are three checks to be performed:

- The smallest channel is not smaller than the minimum channel size.
- The number of channels is not too large to be represented by the minimum channel size.

- The channels are in the same order they appear in the arrangement.

The first check prevents very small channels from being used in the framework, as they are not large enough to carry the Microprotocols (6.10). The second check prevents too many channels from being used, this is because the channels are referenced by their index in the `covert_methods` array, and the micro protocols are limited by size on how many indexes they can represent.

Since the channels are referenced by their index, both the sender and receiver need to be using the same channels, in the same order, this is why the third check is required.

6.9 Decision algorithm

The decision algorithm is responsible for the adaptive nature of the framework, it is used to decide which covert channel is best suited to the current environment. The algorithm is an empirically derived function that takes the following arguments:

- E_s - The desired secrecy of the environment
- L - An array of dictionaries, describing the protocol usages in the environment
- E_r - The rate of packets in the environment
- E_h - The number of active hosts in the environment
- The covert methods to choose from.
- Penalised methods.
- Index of the current method.

While E_s is a user argument, L, E_r, and E_h are derived from The Packet Queue (6.4). Penalised methods have a penalty on their score, to discourage the algorithm from picking them again, why these methods are penalised is discussed in Penalising methods (6.13.1), and the index of the current method is used to discourage the constant switching of channels.

The output of this algorithm is two arrays, R and S where R_i is the rate of the i^{th} covert channel, and S_i is the score of the i^{th} covert channel.

For a deeper understanding of the algorithm, see Algorithm implementation (7.2).

6.9.1 Covertness

C_i is the covertness of the i^{th} covert method, it is calculated using the covertness of the channel defined in Design of covert modules (6.8), and the desired secrecy of the environment, a user argument. The covertness of the method is correlated to the desired secrecy of the environment, and thus it is not just the covertness of the channel.

This way, methods that are more covert than the desired are given a score boost and methods that are less covert than the desired are penalised.

The covertness of the channel is calculated using the following equation:

$$C_i = 1 - \frac{E_s - M_c}{10} \tag{6.1}$$

Where M_c is the covertness of the channel, and E_s is the desired secrecy of the environment.

If the desired secrecy was 5, and the channel had a covertness of 3, C_i would be 1.2 (a 20% boost to the score). Conversely, if the desired secrecy was 10 and the channel had a covertness of 3, C_i would be -0.7 (a 70% penalty to the score).

The reason to not just prevent the use of more covert channels is that the covertness of the channel is also correlated to its use in the environment, if a cover text is very prominent in the environment, the covertness of the method will be higher and thus a better choice than a slightly more covert, but less common method.

C_i is within the interval $[0.1, 1.9]$ (-90% to 90% penalty/boost).

6.9.2 Rate calculation

The rate of a channel determines how often packets should be sent over that channel while remaining covert. This is a function of the rate of packets in the environment, the number of active hosts in the environment, and the covertness of the channel.

The rate of the channel is calculated using the following equation:

$$R_i = \frac{E_h}{P_i * E_r * \frac{C_i}{2}} \tag{6.2}$$

Where P_i is the percentage of packets in the environment that use the protocol of the i^{th} channel, and E_r, E_h and C_i are as defined above. The reason for the $\frac{C_i}{2}$ term is to prevent the rate from being higher than the estimated protocol output per host per second ($\frac{E_h}{P_i * E_r}$) as this would cause the channel be using a protocol more than the average host, and thus be more suspicious.

R_i is within the interval $[0, \infty)$.

6.9.3 Score calculation

The score of a method is a function of its use in the environment, its capacity, and its covertness. The score is calculated using the following equation:

$$S_i = P_i * B_i * C_i \tag{6.3}$$

Where P_i is the percentage of packets in the environment that use the protocol of the i^{th} channel and B_i is the bit capacity of that channel. C_i is as defined above.

The $P_i * B_i$ term is the effective bandwidth of the channel, we'd like our covert communication to be as fast as possible, and thus the higher the effective bandwidth, the better. We again factor in the relative covertness (C_i) of the channel to the desired secrecy.

6.9.4 Preventing constant switching

The index of the current method is used to prevent the constant switching of channels, this is done by giving a bonus of 10% to its score, this percentage is arbitrary, but it is large enough to work and small enough to allow for the channel to be switched if it is not covert enough.

For a method to be chosen, it must be more than %10 better than the current channel, if it exceeds this threshold it will be chosen, and will gain a +10% bonus to its score, and the previous channel will lose its bonus, effectively putting a %20 gap between them. This encourages the channel not to change again unless it is significantly better than the current channel.

6.10 Microprotocols

The adaptive covert channel is reliant on the underlying protocols, these protocols are used to implement:

- Describing the nature of a payload
- Starting communication
- Ending communication
- Switching active channel (The one being used for communication)
- Verifying integrity of the communication
- Discarding chunks of data that have been corrupted
- Recovering from a disconnection in communication

A sentinel signal is sent to begin and end communication, this way they can both share the same micro protocol, once a sentinel is received, then the receiver starts to build the context of the communication, like the received data and the current chunk.

Data is sent in chunks, a chunk is effectively a buffer of data that has not had its integrity verified yet, once the verification of the chunk has passed, it can be committed to the main buffer, and the chunk can be emptied ready for the next set of data. Both the sender and receiver keep track of this chunk so the state of communication is atomic between the two parties.

In order to implement these protocols effectively, using the contextual design proposed in (Anon., 1990), packets need to describe their nature, I decided to implement this using a single bit, at the start of the payload. 0 or 1 indicate whether the packet is data or metadata, respectively. Protocols are implemented as metadata and are of adaptive length, the size of these protocols is a function of the number of underlying channels that may be used. The minimum channel size is $1 + \lfloor log_2 M_c + 1 \rfloor$ where M_c is the number of underlying covert methods.

For some protocols, it is beneficial to use more than the minimum channel size, for example, when verifying the integrity of a channel an offset can follow the protocol. The offset is assumed to be 0 if it is not present, but when it is present it improves the covertness of the channel, exactly why this is the case is discussed in Verifying communication integrity (6.12). For the chunk discarding protocol, the space following the protocol is filled with payload data, this is beneficial to the covertness of the channel as the sender has to resend the previous data segments, and without this offset the fields would be repeated, which would be suspicious to an observer.

For protocols that do not have auxiliary data to be sent with them, the following data bits are assigned at random, this has no benefit to the covertness of the channel since the payload is encrypted and thus appears random, however, the added complexity of the protocol for the tradeoff of bits did not seem worth it.

6.10.1 Protocol compression

It is important to minimise the size of micro protocol headers, so they are negligible to the size of the payload (Backs et al., 2012), we will achieve this by compressing the protocols.

The compression technique outlined in Dynamic Routing in Covert Channel Overlays Based on Control Protocols (3.2) is Huffman coding, which is a simple lossless compression technique that maps the protocols to a number, then the binary representation of that number can be used, however, I do not think this method of compression is best suited to the framework. The problem with Huffman coding arises when the distribution of protocol is not uniform, In my case the majority of traffic will be data, and thus by using only the first bit to declare it as data, the rest of them will be data. The consequence of this is the metaprotocols will be slightly larger than otherwise, but since by their nature they are less frequent than data, this is a worthwhile tradeoff.

6.10.2 Consequence of this approach

Unfortunately, using a single bit to denote data means that the original payload is not taken in whole bytes, A packet of n bits will carry n-1 bits of data. This creates an issue when reading the payload because the bits must be read out of alignment. This problem is exacerbated by the unknown nature of the packets pre-communication, as some channels have higher bit capacities, and along with the introduction of chunk retransmission, it is not feasible to precompute the possible payloads. The solution

I found for this problem was to store the payload as a bitstring, the base two representation of the number in string form, and then read the desired number of bits from the string. There is a time and space cost of storing the payload like this, and in hindsight using a `BitVector` would be more performant, but this cost is negligible and the code is easier to follow than using bit shifts to read the bits.

6.11 Transmission padding

The common approach to adding bits of padding is to append the data with a 1 and fill any remaining space with 0's, the cost of this (minimum number of bits added to the payload) is 1, which is the lowest possible. While this application works well for non-covert scenarios, it can create particularly suspicious-looking payloads, the worst-case scenario of this is an entire packet with a single 1 followed by 0's to fill capacity.

In this paper I will propose a solution that is better suited to covert channels, it does have a higher space cost, however, the benefit is more of the data is random, and sometimes the proficiency of a protocol must be sacrificed in favour of covertness (Saeb et al., 2007). The cost is logarithmically proportional to the size of the data, although if the data has fixed size increments (a common one would be bytes as most payloads will be byte data) then the length can be divided by this number to reduce its costs. In my case, the AES encryption means that the data is a multiple of the key size (128 bits) so the equation of my cost complexity would be $log_2(l/128)$ where l is the length of the encrypted data.

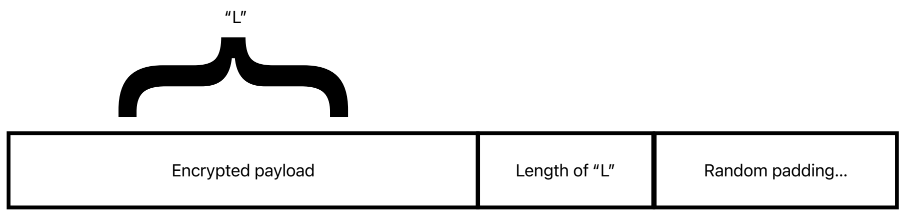

Figure 6.4: Padding diagram

The padding method works by storing the (reduced) size of the encrypted payload after the encrypted payload, and the remaining padding can almost be random. The data is read from right to left, and the sender must ensure that the first valid length is the one that describes the payload. For scenarios where the largest payload size is smaller than the smallest increment size, this is not necessary. In figure 6.4 L is the encrypted payload, and the length of L is a binary representation of the length divided by the increment size. The padding is random data.

Both of these padding methods will be tested and compared in Transmission padding (8.2).

6.12 Verifying communication integrity

Verifying the integrity of communication is essential to the operation of the framework, it allows the communication channel to remediate any errors that may occur and thus allows for the communication to be more reliable, while the framework does use encryption, encryption does not ensure integrity (Fatayer, 2017). The integrity of the communication is calculated using an 8-bit cyclic redundancy check (CRC-8). This allows errors to be recovered by using the discard chunk protocol (outlined in Microprotocols (6.10)).

This requires the receiver to be able to communicate the integrity of the message back to the sender, so the sender can decide whether to resend the chunk or not. In order to do this the medium of communication must be both undisruptable and broadcasted.

The communication must be undisruptable because if the sender does not receive the integrity of the message, it will assume the receiver did not receive the message, and thus resend it.

It must also be broadcastable because there is no guarantee that the receiver will be able to determine the address of the sender, this is particularly true in the case of the TCP Acknowledgement bounce covert channel, as the sender will be using a local server to bounce the packets, and thus the receiver will not know the address of the sender.

A covert channel based on the Address Resolution Protocol (ARP) (outlined in ARP Beaconing (2.3.3)) suited both of these requirements perfectly, as it is broadcasted, and it is required for the normal operation of a network, thus it is undisruptable. There are some applications where the ARP Beacon is not suitable for use:

- The sender is not on the same network as the receiver.
- The network is solely IPv6 (using Neighbor Discovery Protocol (NDP) instead of ARP).
- The nodes of the network are statically configured.

Where ARP is not applicable, a different means of communication must be used, however, that is out of the scope of this paper.

The verification process is a challenge and response protocol, the sender will send a challenge to the receiver, and the receiver will respond with the integrity of the message. Verification of data chunks is performed when the method change protocol is sent, this is when the communication medium changes and thus the framework seeks to verify that the previous channel was successful. This does delay the verification of the data until the start of the next method transmission, but it does mean that the verification can be can become an implicit part of the method change protocol, which keeps the minimum channel size low, and reduces the number of packets sent. This method still allows a the challenge to be performed without changing the channel, by sending the method change protocol to the current channel index, this is useful for continual verification of the channel, but it does increase the number of packets sent.

6.12.1 Improving the covertness of the response

To improve the covertness of the ARP Beacon, the sender will (if the channel is large enough) provide an offset to the receiver, and the receiver will XOR this with the result of the integrity check. Since the sender knows the correct integrity of the data, it can find an active host on the network and XOR the integrity with the host byte, the outcome of this is an offset that the receiver can use to make its ARP request look valid to an observer. This is only possible if the channel is large enough, but if it is not the receiver will assume no offset.

With the offset, the ARP Beacon is much more covert and less prone to detection from Aware active wardens as the adversary knows it is plausible that the channel is for legitimate use and not a channel (Fatayer, 2017). However, this is only true when the integrity of the data is correct, if the data is corrupted, then the request will point to the wrong address, which could raise suspicion. Unfortunately, the receiver cannot determine whether the integrity is correct without sending its response, so this is a tradeoff that must be made.

Some addresses are blacklisted from being used, for example, the address of the sender (as they are in communication, the receiver could send an ARP packet that is unrelated to the covert channel, which could cause issues). The broadcast and network addresses are also blacklisted, as they are not valid hosts on the network, and thus would be suspicious to an observer.

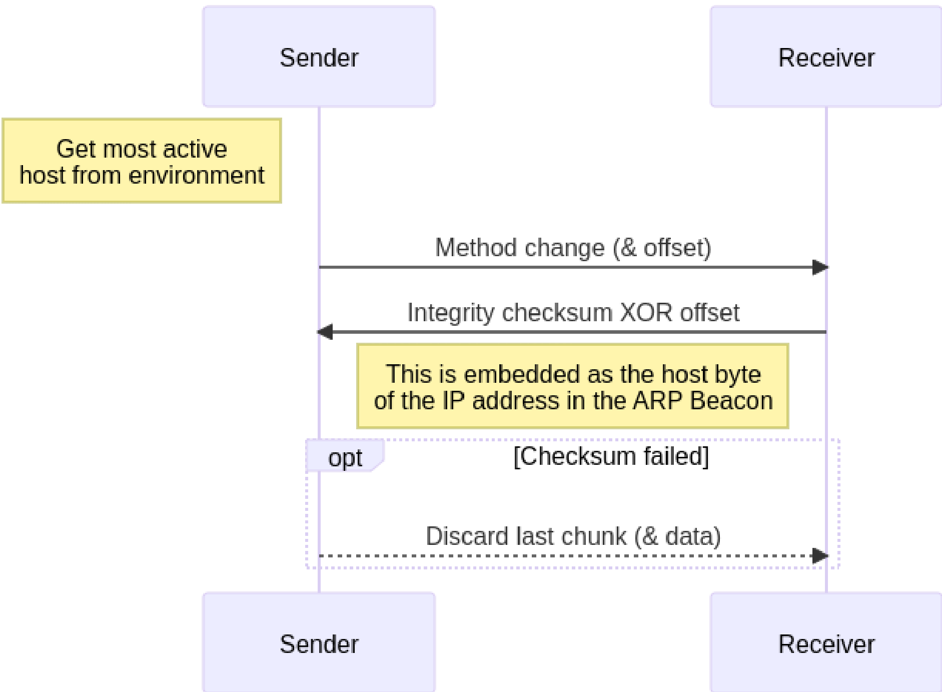

Figure 6.5: Integrity challenge and response (optional parts in brackets)

Since the sender knows what the response is supposed to be, it can await the response and succeed quickly (if it sees the expected response), otherwise, it will wait until the timeout expires (5 seconds by default). Unfortunately, the sender cannot accurately tell if the receiver has not received the packet, or if the integrity is incorrect since the host is assumed to be using ARP communication outside of the channel.

This challenge and response process can be performed irrespective of the underlying covert channel, but it might require limitations on the environment (just as ARP requires a local network).

6.12.2 The discard chunk protocol

The discard chunk protocol tells the receiver to revert the last chunk commit, as the sender determined it was not valid. The receiver assumes the integrity is correct because most of the time it is, and sending a packet to the sender to verify that the last verification was correct is unnecessary, so instead the receiver will assume the last verification was correct unless specifically informed otherwise by the sender, this reduces the total number of microprotocols which is important to reduce the number of packets sent (Backs et al., 2012).

The sender must then resend the previous chunk of data again, but this would cause an issue to arise as the fields of the packets will also be identical, in some cases this can cause an active warden to detect the communication, to prevent this we use utilise the remaining space (the space after the microprotocols) in the packet by filling it with data, this data offsets the bits being sent (by however much space is available) and thus the following packets will have different fields.

6.13 Channel failures

There are scenarios where the channel for communication no longer works, this could be of a change in the environment or a warden that has prevented the channel. In these scenarios the framework must be able to recover the communication effort, the sender and receiver need a fallback channel that they can use to restart communication. However, if the fallback channel was the one being blocked then the communication would have to end, even if other channels were viable.

To prevent this, a recovery mode is defined for the framework, in this mode the receiver is open to all channels, and the sender will iterate through possible channels until it finds one that works. To prevent this from happening unintentionally the recovery mode is only entered when:

- The receiver has failed two consecutive integrity checks.
- The receiver has not received any covert data for an interval.

Either one of the above conditions will send the sender/receiver into recovery mode, and the other one will eventually do the same. The sender will purposely wait until the interval has passed before attempting to recover, this is to prevent the sender from trying to switch channels prior to the receiver entering the recovery mode. This interval is based on the time and packets of the last verified chunk (the number of packets between verification, and the time between verification), this interval must be exceeded by 50% before the receiver will enter recovery, and the sender will wait for even longer before attempting to recover. It is important to note that since this time between verifications could naturally exceed the interval, the receiver will still act as if the current channel is viable, saving the data to the chunk (Although this chunk will be discarded implicitly if the channel is "recovered").

The recovery mode can be recovered with a specially crafted packet, this packet should begin with the sentinel signal (as if it was starting communication) followed by the length of verified data, and an offset. The offset is used for the same function here as it is in Verifying communication integrity (6.12), however, it is used in combination with the length of verified data, which is the length of the last sender-verified data. There are some cases where the receiver will have to assume its last verification was valid, but in reality, the discard chunk failed to reach it, but the sender will always know the length of the data at the last verification.

Since this packet requires more space than the minimum channel size, there are some cases where it does not fit in a channel, in these cases it simply cannot be used, and the alternative is to recover to a method with a larger bit capacity, then switch to the higher scoring channel afterwards. While this requirement is not perfect, it is the best way to prevent receivers from breaking communication due to interference from external sources.

When the receiver is recovered, it will send an integrity check back to the sender so it knows that the recovery was successful, the intervals between verifications here will not be recorded since they may not be consistent between the sender and receiver.

6.13.1 Penalising methods

This recovery process is not ideal for covert communication, and causes the sender to try multiple channels, if these other channels are blocked then it can raise suspicions, to discourage this we penalise methods.

When a method fails twice, causing the recovery process to occur, it receives a penalty of -90% to its score. This penalty lasts for 30 packets of valid data communication and prevents the method from being chosen again unless there are no better alternatives. The reason we do not permanently block the protocol is that it can fail for temporary reasons, but we cannot know that until we try again, It is undesirable to constantly retry failing methods as if they are being caught by wardens it is much more suspicious for many packets to have been blocked rather than just a few.

Chapter 7

Implementation

7.1 Implementation of covert modules

The following is an extract from the source code of the framework (`FaucetEnv/Faucet/src/covert_channels/covert_channels.jl`) that shows the implementation of a covert module.

```
ipv4_identifaction::covert_method{:IPv4_Identification} = covert_method(
     "IPv4_Identification",
     Layer_type(3), # network
     "IPv4_header",
     8,
     16, # 2 bytes / packet
)

# Init function for IPv4_Identification
function init(::covert_method{:IPv4_Identification}, net_env::Dict{Symbol, Any})::Dict{Symbol, Any}
     target_mac, target_ip = net_env[:dest_first_hop_mac], net_env[:dest_ip].host
     return Dict{Symbol, Any}(
          :payload => Vector{UInt8}("Covert packet!"), # not a real payload
          :env => net_env,
          :network_type => IPv4::Network_Type,
          :transport_type => TCP::Transport_Type,
          :EtherKWargs => Dict{Symbol, Any}(
              :dest_mac => target_mac,
          ),
          :NetworkKwargs => Dict{Symbol, Any}(
              :dest_ip => target_ip,
          )
    )
end

```

```julia
# Encode function for IPv4_Identification
function encode(::covert_method{:IPv4_Identification}, payload::UInt16; template::Dict{Symbol, Any})::Dict{Symbol, Any}
    template[:NetworkKwargs][:identification] = payload
    return template
end
encode(m::covert_method{:IPv4_Identification}, payload::String; template::Dict{Symbol, Any})::Dict{Symbol, Any} = encode(m, parse(UInt16, payload, base=2); template=template)

# Decode function for IPv4_Identification
decode(::covert_method{:IPv4_Identification}, pkt::Packet)::UInt16 = pkt.payload.payload.header.id
```

This implementation makes use of Julia's multiple dispatch system, allowing for different functions to be called depending on the type of arguments. The three defined functions `init`, `encode`, and `decode` are called with the covert channel as the first argument.

I designed the covert channels to be a composite parametric type, where the type parameter is the type of channel, this prevents ambiguity when the multiple dispatch system is used. The parametric type definition looks like this:

```julia
struct covert_method{Symbol}
    name::String
    layer::Layer_type
    type::String # What packet type are we aiming for?
    covertness::Int8 # 1 - 10
    payload_size::Int64 # bits / packet
    covert_method(name::String, layer::Layer_type, type::String,
    covertness::Int64, payload_size::Int64)::covert_method{Symbol} = new{Symbol(name)}(name, layer, type, Int8(covertness), payload_size)
end
```

This struct instantiation implicitly converts the name of the covert channel to a symbol, which is used as the type parameter.

7.2 Algorithm implementation

The input variables and output of the implementation can be found in Decision algorithm (6.9). This section will cover the entire algorithm, getting from the input variables to the output.

This function is written in Julia, making use of its ability to use Unicode characters in variable names, allowing me to use the same mathematical notation in the code as in the design documentation. The following is an extract from the source code of the framework (`FaucetEnv/Facuet/src/covert_channels/covert_channels.jl`)

```
function method_calculations(covert_methods::Vector{covert_method},
    env::Dict{Symbol, Any}, Eₚ::Vector{Int64}=[],
    current_method::Int64=0)::NTuple{2, Vector{Float64}}
    # Get the queue data
    q = get_queue_data(env[:queue])

    # Covert score, higher is better : Method i score = scores[i]
    S = zeros(Float64, length(covert_methods))
    # Rate at which to send covert packets : Method i rate =
    rates[i]
    R = zeros(Float64, length(covert_methods))

    if isempty(q)
        @error "No packets in queue, cannot determine method" q
        return S, R
    end

    #@warn "Hardcoded response to determine_method"
    L = [get_layer_stats(q, Layer_type(i)) for i ∈ 2:4]

    # Eₗ : Environment length : Number of packets in queue
    Eₗ = length(q)

    # Eᵣ : Environment rate : (Packets / second)
    Eᵣ = Eₗ / abs(last(q).cap_header.timestamp -
    first(q).cap_header.timestamp)

    # Eₛ : Environment desired secrecy : User supplied (Default: 5)
    Eₛ = env[:desired_secrecy]

    # Get the count of hosts local to the supplied ip (we don't
    want to consider external ones).
    Eₕ = get_local_host_count(q, env[:dest_ip])

    for (i, method) ∈ enumerate(covert_methods)
        Lᵢ_temp = filter(x -> method.type ∈ keys(x), L)
        if isempty(Lᵢ_temp)
            @warn "No packets with valid headers" method.type L
            continue
        end
        # Lᵢ : the layer that method i exists on
        Lᵢ = Lᵢ_temp[1]

        # Lₛ : The sum of packets that have a valid header in Lᵢ
        Lₛ = +(collect(values(Lᵢ))...)

        # Lₚ : Percentage of total traffic that this layer makes up
```

```
        Lₚ = Lₛ / Eₗ

        # Pᵢ is the percentage of traffic
        Pᵢ = Lₚ * (Lᵢ[method.type] / Lₛ)

        # Bᵢ is the bit capacity of method i
        Bᵢ = method.payload_size

        # Cᵢ is the penalty / bonus for the covertness
        #  has bounds [0, 2] -> 0% to 200% (± 100%)
        Cᵢ = 1 - ((method.covertness - Eₛ) / 10)

        # Score for method i
        #  Pᵢ * Bᵢ : Covert bits / Environment bits
        #  then weight by covertness
        #@info "S[i]" Pᵢ Bᵢ Cᵢ Pᵢ * Bᵢ * Cᵢ
        S[i] = Pᵢ * Bᵢ * Cᵢ

        # Rate for method i
        #  Eᵣ * Pᵢ : Usable header packets / second
        #  If we used this much it would be +100% of the environment rate, so we scale it down
        #  by dividing by hosts on the network, Eₕ.
        #  then weight by covertness
        #  We don't want to go over the environment rate, so reshape covertness is between [0, 1] (1 being 100% of env rate)
        #  (Eᵣ * Pᵢ * (Cᵢ / 2)) / Eₕ : Rate of covert packets / second
        #  ∴ 1 / Eᵣ * Pᵢ * (Cᵢ / 2) : Interval between covert packets
        #@info "R[i]" Eₕ Eᵣ Pᵢ Cᵢ/2 Eₕ / (Eᵣ * Pᵢ * (Cᵢ / 2))
        R[i] = Eₕ  / (Eᵣ * Pᵢ * (Cᵢ / 2))
    end

    # Eₚ (arg) : Environment penalty : Penalty for failing to work previously
    for i ∈ Eₚ
        S[i] *= 0.1 # 10% of original score
    end

    # Allow for no current method (as the case is when recovering)
    current_method != 0 && (S[current_method] *= 1.1) # Encourage current method (+10%)

    return S, R
end
```

The `method_calculations` function only does the calculations, not the selection of the best method, which is done by a function that wraps this one and returns an index i and a rate R_i for the highest scoring method (S_i).

7.3 Integrity check

The integrity check process has two main parts, the sender, and the receiver. For both of the processes, the integrity is calculated using the same function (`FaucetEnv/Facuet/src/utils.jl`):

```
function integrity_check(chunk::String)::UInt8
    padding = 8 - (length(chunk) % 8)
    # Payload may not be byte aligned, so pad it
    chunk *= padding == 8 ? "" : "0"^padding
    return crc(CRC_8)([parse(UInt8, chunk[i:i+7], base=2) for i in 1:8:length(chunk)])
end
```

Figure 7.1: `integrity_check` function

This function uses a CRC8 checksum, but in reality, its implementation is not important, the only requirements are that the function is deterministic and that a single byte is returned. A consideration for choosing such a function should be its behaviour when there is not a guarantee that the current chunk is a multiple of 8 bits, as discussed in Consequence of this approach (6.10.2), if the checksum function requires bytes, then the chunk must be padded in a deterministic way.

To prevent inconsistencies in the check, the same function is used by both the sender and receiver, by putting the function in the shared `utils.jl` file.

The sender needs to issue the challenge and receive the response. The challenge piggybacks the method change protocol, so it just needs to await the response from the receiver. It does this by opening a raw socket and listening for an arp packet from the target to the known host it used to create the offset. We can see the challenge and response process in the `send_covert_payload` function (found in `FaucetEnv/Facuet/src/outbound/packets.jl`) in figure 7.2.

Once the packet is sent, the sender enters the `await_arp_beacon` function, providing the IP address of the target (which we expect the source of the ARP request to be), the known host (which we extract from the local network (line 4), as discussed in Improving the covertness of the response (6.12.1)), and a timeout of how long to wait, in seconds. The timeout is largely unimportant as the function will return as soon as it receives the expected packet, which is normally a fraction of a second. The function can be seen in figure 7.3.

7.3.1 Final integrity check

An oversight of the integrity process outlined in the Verifying communication integrity (6.12) section is the last chunk does not get verified, this undermines the entire

```
1 # Get the current integrity, if we have used final padding, append it
2 integrity = integrity_check(bits[chunk_pointer:pointer-1] * final_padding)
3 # Get a known host for the integrity check
4 known_host = get_local_net_host(net_env[:queue], net_env[:src_ip], host_blacklist)
5 # Send the challenge
6 send_method_change_packet(method, method_index, integrity ⊻ known_host, net_env, method_kwargs)
7
8 ...
9
10
11 # Await the response
12 if await_arp_beacon(net_env[:dest_ip], known_host, check_timeout) # Success
13     # reset failures
14     protocol_failures = 0
15     @debug "Integrity check passed" method=method.name integrity known_host integrity ⊻ known_host
16     # reset chunk pointer
17     chunk_pointer = pointer
18     # update last verified integrity (+ length)
19     last_integrity = (chunk_pointer, integrity)
20 else
21     ...
```

Figure 7.2: Challenge and response process implementation

integrity of the entire process, the solution was to add a final integrity check. Because of the design of the packet-sending function, a loop that iterates the payload, verifying the last chunk of data becomes less trivial, as we want to perform the check after the loop, but if the check fails we would have to return to the loop, this would require a confusing nested loop structure and would be difficult to read, especially when the already complex function is considered. To solve this problem, I utilised the `@goto` and `@label` macros provided by Julia, this creates an unconditional jump to a label, allowing me to jump back to the integrity check when combined with a flag `finished` that indicates there is no more data to send if the integrity check succeeds with the flag set, it will break the loop, otherwise, it will resend the chunk as usual.

This worked, however, the final integrity check always failed. The reason for this was the sender was creating the integrity check using the payload, but the receiver was creating the integrity check using the payload and the padding, meaning that despite the data being the same, the integrity check was different. To solve this problem, I added a variable `final_padding` (see line 2 of figure 7.2) that is an empty string unless it is the final chunk, in which case it is the padding added to the payload to fill the channel capacity.

```julia
    function await_arp_beacon(ip::IPv4Addr, target::UInt8, timeout::Int64=5)
    # Get a fresh socket to listen on
    socket = get_socket(AF_PACKET, SOCK_RAW, ETH_P_ALL)
    start = time_ns()
    while (time_ns() - start) < timeout * 1e9
        # Read a packet
        raw = read(socket)
        # Confirm it is more than the minimum size of an ARP packet
        if length(raw) >= 42
            # Check it matches our boilerplate ARP
            if raw[ARP_SEQUENCE_SLICE] == ARP_SEQUENCE
                # Check it is from the source we are looking for
                if raw[ARP_SRC_SLICE] == _to_bytes(ip.host)
                    # Check it is to the target we are looking for
                    if raw[ARP_DEST_SLICE][4] == target
                        return true
                    end
                    push!(heard, raw[ARP_DEST_SLICE][4])
                end
            end
        end
    end
    return false
end
```

Figure 7.3: `await_arp_beacon` function

The implementation of this is spread across the entire `send_covert_packet` function and thus is difficult to show in a code snippet, but it can be seen in the source code (`FaucetEnv/Facuet/src/outbound/packets.jl`).

7.4 Getting a valid TCP server

For the TCP Acknowledgement bounce channel I wanted to use a locally accessible TCP server to showcase how the framework allows adaption to the environment at a channel level, As discussed in Design of covert modules (6.8). Unfortunately, because my test environment is a packet capture replay, the server found by the framework cannot be communicated with (as there is no host to communicate with). To solve this problem, I hardcoded the response to point at a valid server, this is not ideal, but it is a workaround for this project. We can still see the server that the framework finds in the packet capture.

The `get_tcp_server` function works by looking at the TCP traffic in the environment, It favours traffic that is going to, or from, a TCP service like HTTP (Port 80) or HTTPS (Port 443). If it cannot find any of those it will simply look for a TCP packet with the SYN Flag set, which indicated a server, and then return it.

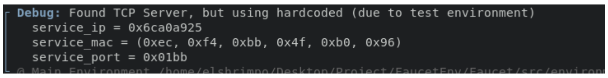

Figure 7.4: The TCP server found by the framework

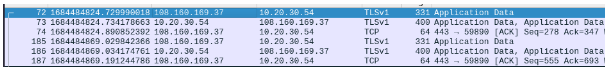

Figure 7.5: The TCP server found by the framework in the packet capture

```
function get_tcp_server(q::Vector{Packet})::Union{Tuple{NTuple{6, UInt8}, UInt32, UInt16}, NTuple{3, Nothing}}
    common_query = Vector{Dict{String, Dict{Symbol, Any}}}([
        Dict{String, Dict{Symbol, Any}}(
            "TCP_header" => Dict{Symbol, Any}(
                :dport => TCP_SERVICES
            )
        )
    ])
    # Clients connected to common TCP Services, TCP traffic to them is favourable (In terms of covertness)
    services = query_queue(q, common_query)

    ...

    service_mac, service_ip, service_port = nothing, nothing, nothing
    if !isempty(services)
        # Take the most recently active one
        service = pop!(services)
        # This is a packet going toward tcp service
        service_mac = getfield(service[layer2index["Ethernet_header"]], :source)
        # Ethernet_header -> tcp server mac (of next hop from local perspective)
        service_ip = getfield(service[layer2index["IPv4_header"]], :daddr)
        # IP_header.daddr -> tcp server ip
        service_port = getfield(service[layer2index["TCP_header"]], :dport)
        # TCP_header.dport -> tcp server port
    else
    ...
    return (mac, ip, port)
end
get_tcp_server(q::CircularChannel{Packet})::Union{Tuple{NTuple{6, UInt8}, UInt32, UInt16}, NTuple{3, Nothing}} = get_tcp_server(get_queue_data(q))
```

Chapter 8

Testing & Evaluation

This chapter tests and evaluates the produced framework against the objectives and non-functional requirements outlined in Project objectives (4) using the environment outlined in The testing environment (5.2).

8.1 Recovering from a communication channel failure

Communication failure can be caused by a variety of factors, but the most important to the covert system are caused by an adversary, especially in applications like censorship resistance.

I wrote a script to mimic an active warden (see filter.py (B.4) for implementation details), this script can block either one of the channels, or both. The script works by hooking into IPTables using a NetFilter queue, where it can drop and alter packets with the assistance of scapy (Rohith Raj et al., 2018). The script is rudimentary in its current state, as I do not have actual traffic to worry about disrupting, I can nullify header fields at my discretion. While the script sets the acknowledgement number to 0, in real-world application it would map these values in a fashion similar to that of Network Address Translation.

The clearest way to show this in action is to look at the script output during this process, to mimic this scenario I created the `FaucetEnv/Commblock` script, which calls the warden script (appendix B.4).

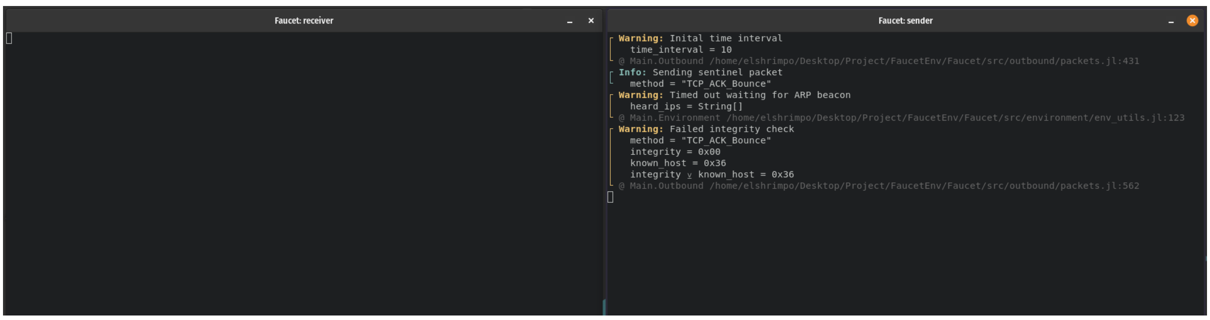

Figure 8.1: TCP Channel failure

Figure 8.2: TCP Channel fails again, starting the recovery process

Figure 8.3: The receiver "recovers" to this new channel

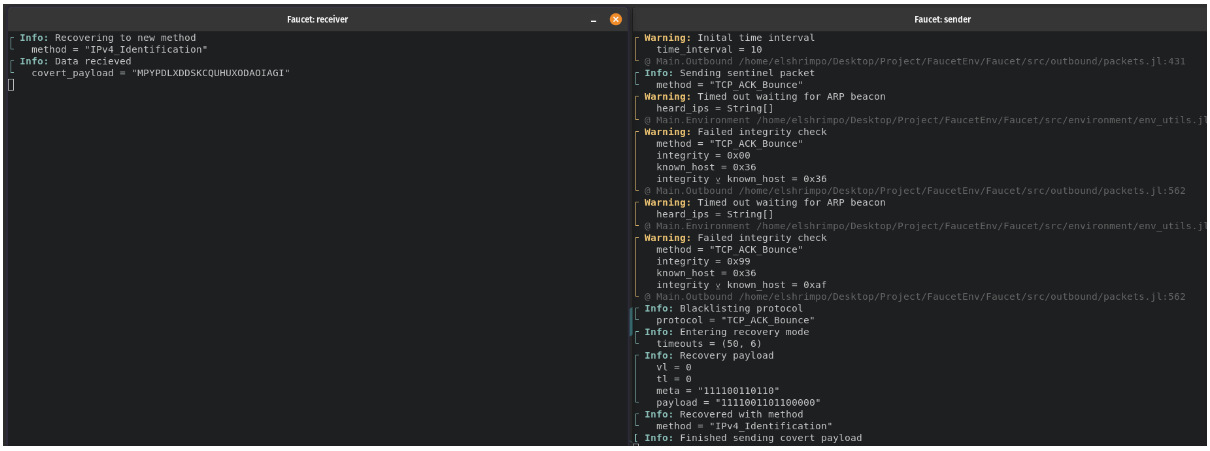

Figure 8.4: Communication is restored, and successful

Figure 8.5: The affect of the warden script on the packets

Running the testing script we can see that communication initially is unsuccessful (figure 8.1), however, once the sender fails twice in a row with the same protocol, it blacklists the protocol and enters recovery mode (figure 8.2). The sender waits before going into recovery mode to ensure the receiver is also in recovery mode. How these times are decided is outlined in Channel failures (6.13). Once it sends the recovery packet to the receiver, the receiver will "recover" onto this new channel, that it knows is valid (figure 8.3). Once the receiver has recovered, communication will continue as normal (although the original channel will remain blacklisted, and thus is unlikely to be used for a specified period). This can be seen in figure 8.4.

By enabling the debug flag on the warden script, we can see how the warden affects the packets (figure 8.5), where the acknowledgement field is set to 0.

This example shows that the framework can successfully recover from a communication channel failure, however, failures are not always as clear and simple as this, I doubt this approach is robust enough to handle all failures, but it does show it is possible. Managing the quality of channels is a complex task, but it would improve the availability of the system, especially when the environment is unreliable. This is an area that could be improved in future work.

8.2 Transmission padding

The types of padding, outlined in Transmission padding (6.11), are tested in this section.

The padding that works with a 1 followed by 0's for the remaining space will be referred to as "short" padding, and the padding proposed will be referred to as "covert" padding.

The drawback of covert padding over short padding is the additional space required to transmit the padding, however, due to the block size of AES encryption, the number of additional packets sent by this padding is negligible:

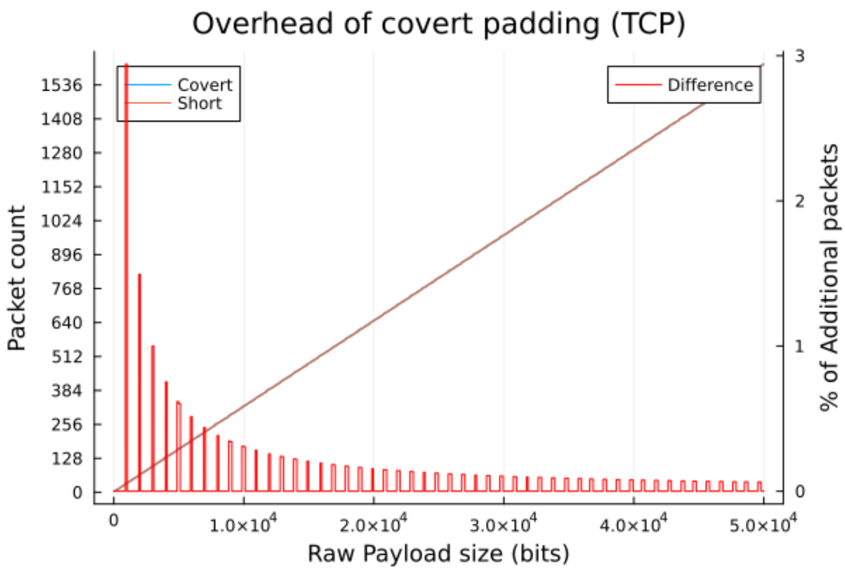

Figure 8.6: The size of the padding required for a given payload size (TCP)

We can see here that the use of covert padding has very little effect on the number of packets sent, with the maximum being 5% more packets sent for a payload (for IP). It is evident that the covert padding method is more effective for TCP, but for both protocols, they do not outperform the short padding method. By taking the number of payloads that require an extra packet to be sent, and dividing it by the total number of payloads, we get the percentage of packets that require an extra packet to be sent, multiplying that by the average percentage increase in packets sent, we can find the approximate overhead of the covert padding method:

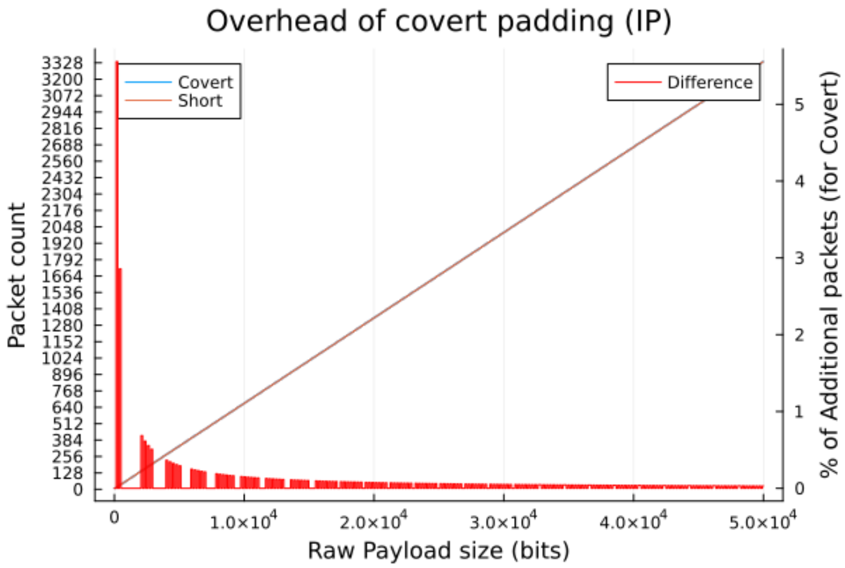

Figure 8.7: The size of the padding required for a given payload size (IP)

Where n is the number of payloads, d is the number of p that require an extra packet to be sent, p_s is the average percentage increase in packets sent, and o is the average overhead of the covert padding method, expressed as a percentage.

$$o = \frac{d}{n} * p_s * 100$$

The overhead for packets for TCP and IP is 1.02% and 2.71% respectively.

Where the covert padding method performs is in the distribution of the bits, where fields are supposed to be random, or unique, the percentage of 0's should be approximately 50%, however, with short padding, this is not the case:

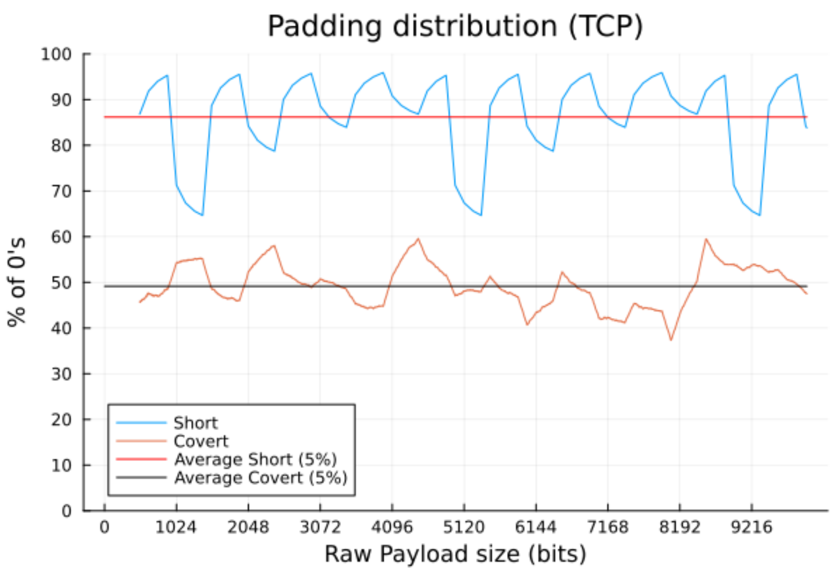

Figure 8.8: A comparison of bit distributions in the TCP header with short padding and covert padding

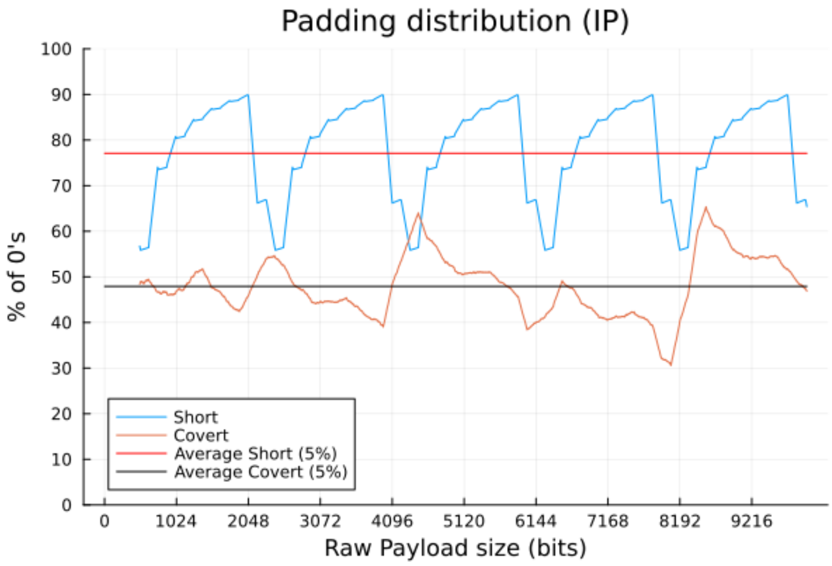

Figure 8.9: A comparison of bit distributions in the IP header with short padding and covert padding

We can see here that the covert padding method is much closer to the ideal distribution than the short padding method. We can also see that again, the larger field of the TCP header is better suited to the covert padding method, with a tighter distribution than the IP header. The covert headers are not perfect, because the size of the data always starts with a 0, which is required for the padding to be removed, however, this difference is negligible and would not be statistically significant.

8.3 Testing the covertness of communication

AES has passed the randomness test, and the padding method has a distribution that is much close to the ideal (with negligible variation), some other parts of communication can introduce patterns that reduce the covertness of the communication. These include the micro protocols and the random padding that follows them.

Extracting the IP Identification field from the captures of 256 communications, we can see that the distribution of the bits is not random and that there is a clear pattern to the data:

I am unsure what exactly causes the distributions to look like this, but I suspect it is something related to micro protocols, the sparse nature of the points at the top of the graph is likely an artefact of the first bit that denotes data (0 for data, 1 for meta protocol), since our communication is more data than meta protocol, this is to be expected. However, neither of these distributions matches the overt data (figure 8.12), which is undesirable. This could open the framework up to detection methods that look for these patterns, however, as I am unaware of the cause of these patterns, I am unsure how to detect them.

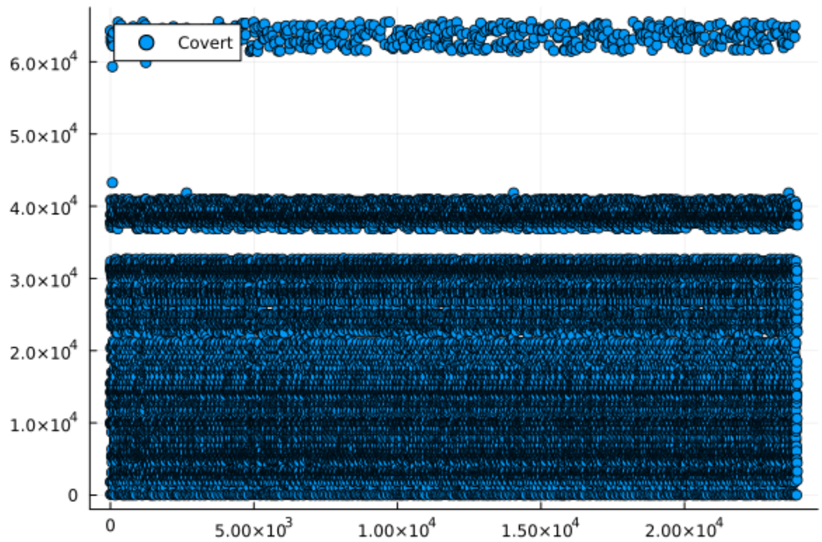

Figure 8.10: The distribution of the bits in the IP Identification field

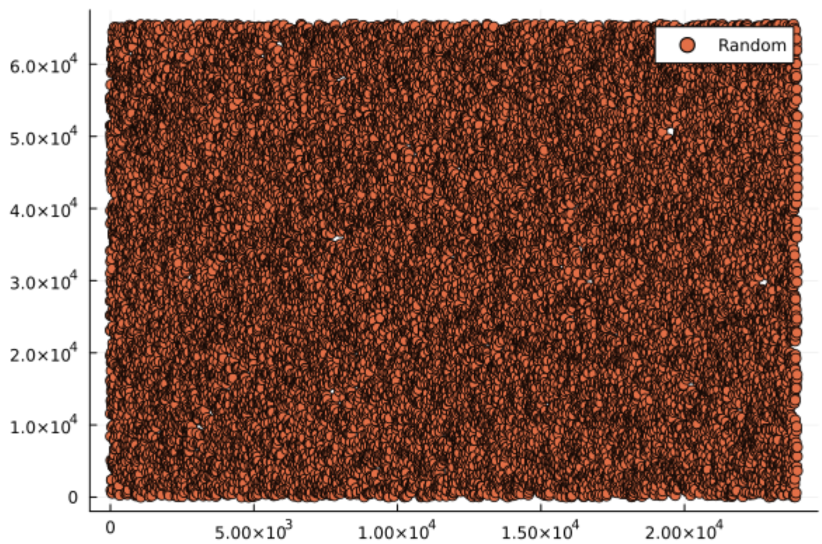

Figure 8.11: A Random distribution of a UInt16 field

8.4 Testing channel selection and adaption

The framework can effectively select the best channel for the current situation. It will attempt to choose the channel with the highest bandwidth, while still factoring in covertness. Unfortunately showing this in a figure is difficult. The first channel is always chosen to do the sentinel, but it is not always the best suited, in a TCP-biased environment (like the one seen in figure 8.13) the IP Identification is not out of place, but it does have half capacity of the TCP Acknowledgement bounce, so it will immediately switch (see figures 8.14 & 8.15).

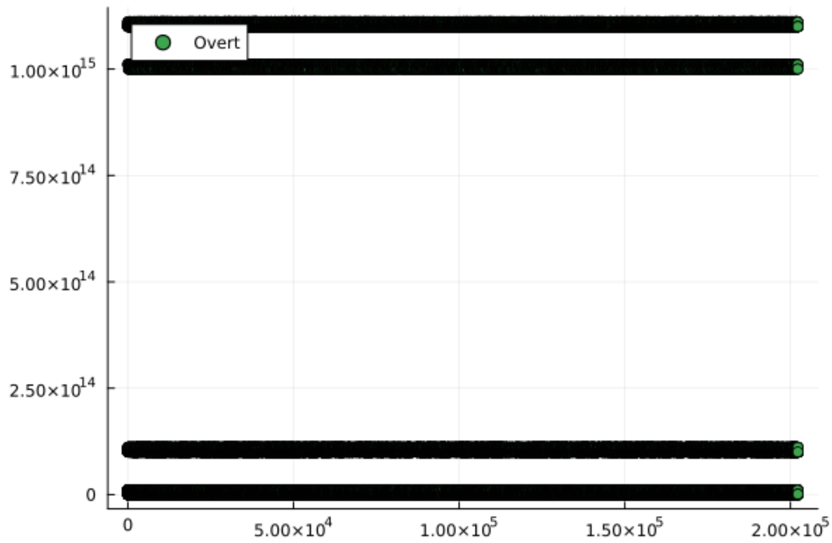

Figure 8.12: The distribution of the bits in overt traffic fields

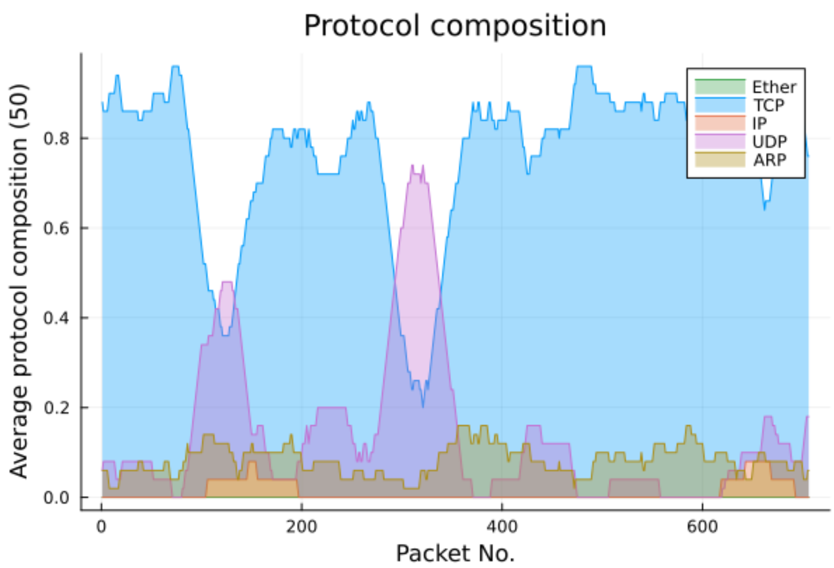

Figure 8.13: The composition of the environment

8.5 Testing detection and recovery from failures

The framework can detect failures in communication using the integrity challenge, under normal circumstances the integrity of the payload will not be compromised, however, by running the warden script (see filter.py (B.4)) we can compromise a section of communication, and see the framework recover from it.

Figure 8.16 shows the compromised payload (the middle one), where the field has been altered, when the sender next sends an integrity challenge, the receiver will respond with the incorrect value, and the sender will inform it to discard the last chunk of data, see figure 8.17. Conversely, if the receiver sends the correct value, the sender will continue as normal, see figures 8.18 & 8.19.

The framework successfully meets all of the ***MUST*** requirements outlined:

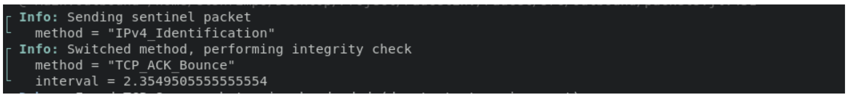

Figure 8.14: Sender adapting to the environment

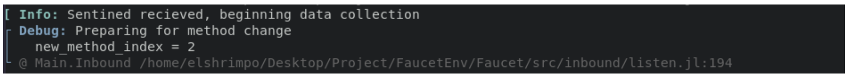

Figure 8.15: Receiver switching communication channel

- The framework can determine the best communication channel for the current situation, as shown in Testing channel selection and adaption (8.4).
- The framework can adapt to its environment, as shown in Testing channel selection and adaption (8.4).
- The framework can detect and recover from failures in communication, as shown in Testing detection and recovery from failures (8.5).
- The framework uses encrypted communication, with a pre-shared key, as outlined in The arrangement (6.5).

The framework also meets the ***SHOULD*** requirement:

"The framework can recover from the complete failure of a communication channel", as shown in Recovering from a communication channel failure (8.1).

However the requirement:

"Allow bidirectional communication" is not met, and I think is not possible for the framework to meet this requirement without large refactoring. While the framework does technically allow for bidirectional communication, this is simply limited to a response to a message, not a full conversation. For this system to be capable of bidirectional communication both parties would have to be able to agree on a channel that suits both of their environments, this would require more information to be shared between the parties and a more complex selection algorithm, additionally, this would impose a penalty on the response time of the system to a change in the environment.

The ***MAY*** requirement:

"[The framework can] handle multiple channels at a time" is also not met. The additional complexity of this requirement is not worth the benefit, although if only direct channels (Ones that go straight from the sender to the receiver) are used, the receiver could reasonably handle multiple channels by tracking the source of the packets. However, this would prevent the use of some channels, such as the TCP Acknowledgement bounce that I utilise in this paper.

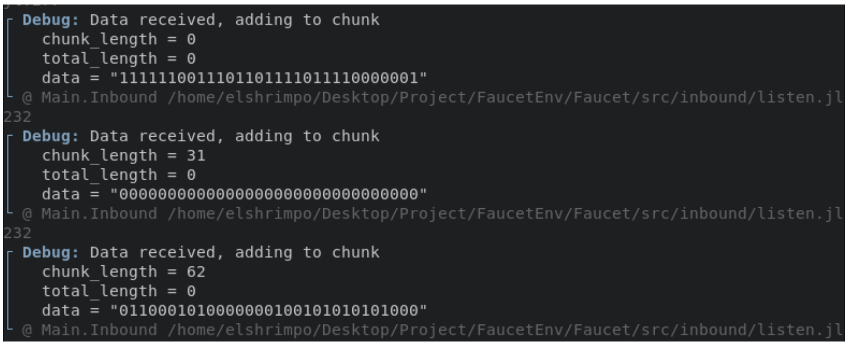

Figure 8.16: A compromised payload

Figure 8.17: The discarding of the compromised chunk

8.6 Evaluation of non-functional requirements

8.6.1 The effect on covertness of the framework

The use of multiple channels for communication allows for the framework to adapt to its environment, this is a great benefit to the covertness of the system, it also means that if a channel were to be discovered, for an adversary to extract the information they would have to work out the indexes of the relevant methods, which is a non-trivial task.

The downside to using multiple channels for communication is that they require protocols, and protocols require a certain degree of structure. Where an adversary is aware of these protocols they could feasibly use them to detect the presence of a covert communication system. This is a problem that is not unique to this framework, but to all covert systems that use protocols.

The reduced covertness of the protocol structure is negligible in comparison to the benefits of multiple channels, as the search domain increases with the addition of more channels, and structure is easily mistaken for noise.

Writing a script to identify the channel is simple, however, identifying only the channel is much more difficult. It works by searching for the sentinel values in possible

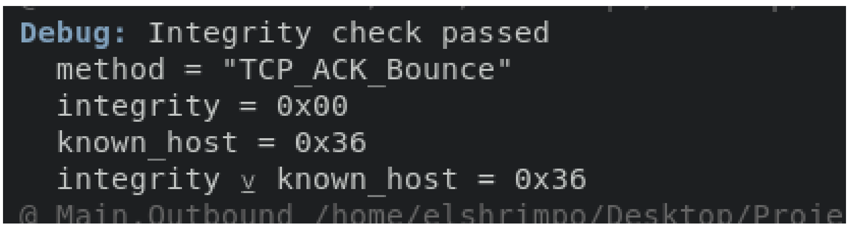

Figure 8.18: The sender sending an integrity challenge

```
Debug: Preparing for method change
  new_method_index = 2
@ Main.Inbound /home/elshrimpo/Desktop/Project/Fa
Debug: Sleeping before arp... (5)
  integrity = 0x00
  offset = 0x36
  integrity ⊻ integrity_offset = 0x36
@ Main.Inbound /home/elshrimpo/Desktop/Project/Fa
Debug: Sending ARP beacon
  encoded_byte = 0x36
```

Figure 8.19: The receiver responding to the integrity challenge

channels, in a clean environment (where the only traffic between the sender and receiver is the covert channel) this will detect the IP channel with 100% accuracy, however, its false positive rate is much too high to say with certainty that a covert channel exists. This script is available in the appendix at warden.py (B.3). The output of running this script on a pcap that only lasts for the duration of the covert channel can be seen in figure 8.20, where two of the identified sentinels are part of the covert channel.

```
With 763 packets:
 - Host (10.20.30.54): 119 sentinels found
 - Host (10.20.30.3): 23 sentinels found
 - Host (10.20.30.2): 21 sentinels found
False positive rate: 98.77%
```

Figure 8.20: The output of the rudimentary warden script

This script does not represent the true covertness of the channel well, as it assumes the sender knows the channels in the communication. it also assumes the sentinel value, but there is no reason why the sentinel value could not be changed. For example, one's complement could be applied to all the encoded data, flipping all the bits, this would mean that the sentinel value would be 0000, which would evade this form of detection with no performance loss (but it would have to be agreed beforehand).

This type of warden also fails to properly detect the TCP Acknowledgement bounce channel, as the script sees the acknowledgement from the bouncing server, as opposed to the sender, which makes correctly identifying the start/end of a channel incredibly difficult.

For a more accurate warden, it would have to be aware of the following:

- The covert methods that are being used.
- The sentinel value used.
- The index of each method (so it can "follow" the communication).

This is a difficult task, but perhaps could be done using a machine learning approach, where the warden is trained on a set of known channels, and then tested on an unknown channel. This would be a good area for future work.

To conclude this evaluation, it is easy to identify a possible covert channel, however, it is incredibly hard to identify the channel with certainty, even when the warden has knowledge of the implementation. Regardless of whether detection is possible retrospectively, it would require the channel to have already ended for communication to be identifiable. additionally, the solution for finding a channel is not of linear complexity, as the number of channels increases, the number of possible combinations increases exponentially. So the channel cannot be identified with certainty in real-time, or possibly at all, meaning it is incredibly covert.

8.6.2 Application of the framework to real-world scenarios

While the framework is proven to work inside a controlled environment, I doubt it would work in a real-world scenario. Given the complexity of the implementation, and my limited time, I have made shortcuts in the framework, the majority of my concern lies with the micro protocols, as I suspect they will be prone to interference from legitimate communication. While my channel can handle them being set to nothing, if they happened to be set to the right combination of values it could cause the channel to fail. There are also variable definitions, like the time to wait for a response to an integrity challenge, which has been arbitrarily set, and may not be suitable for all environments.

Another issue is the size of the framework, because of julias (current) lack of compilation, the entire standard library is required to run the framework, this is a large overhead and would be a problem in real-world applications where the sender is trying to remain undetected by the host it is running on. For scenarios where this is not an issue, like censorship resistance, this is not as much of an issue, although it does mean it requires more resources to run.

8.6.3 Capability to add new communication channels

Adding new channels to the framework is relatively simple, especially for channels that use already-supported protocols (such as IP or TCP), adding new protocols does however require additional work, as the receiver and sender will require a way to represent the protocol as a `Packet` (see Packet processing (6.3)), and a way to create the packet from a dictionary (see Sending packets (6.6)) respectively.

While these tasks are not particularly arduous, for somebody unfamiliar with the language they may struggle to understand the codebase, and the lack of documentation regarding the addition of new channels may also be a hindrance.

This is an area that could be improved in future work, by adding documentation and examples or perhaps creating a tool that parses a protocol from a header file and generates the required code.

However, the only way to avoid this problem would be to use a language with support for these packets, such as C, however, this would not come without its problems. For example, C has support for these headers, but it is not as easy to understand or add to the codebase as Julia, conversely, Python is easy to understand and add to, but comes with performance limitations and a lack of portability (Note that Julia does not yet have a stable compilation process, but it is under development).

An implementation of a covert channel can be seen in Implementation of covert modules (7.1).

Chapter 9

Conclusions

9.1 Conclusion

This project and the research related to it is a success. All of the ***MUST*** requirements have been met, and the ***SHOULD*** requirements have been met to a satisfactory level (although not quite perfect implementations). The remaining ***MAY*** requirements are not met, after weighing up the value of these additional benefits, I didn't think they would be a useful application of my limited time.

The framework was a complicated system to design and implement, and having encountered many problems on the way in addition to the wide scope of the project, I did not get to test the implementation to the extent I would have liked. Given extra time, I would want to test the environment in a wider range of environments, and with a wider range of protocols, and also test the framework against current warden solutions.

In hindsight, bidirectional communication would probably have proved to have been a useful feature, not only to increase the framework's use cases, but it would allow more robust error-checking mechanisms to be implemented.

Overall, I think my work has successfully shown that an adaptive covert communication framework is not only possible, but beneficial to the integrity, and availability of communication. The proposal of better micro protocols and a more covert padding implementation are also important contributions to the field.

9.2 Future work

Before the framework is used in a real-world scenario for covert communication, there is plenty of future work to extend the framework and improve its performance:

- No attempt at having a valid cover text:
  - The scope of this paper was the protocols involved in communication, however, the payload data is incredibly important to the application of the framework in the real world. The covertness of the communication is only as strong as the weakest link, in this case, it is the overt traffic.

- The current channel algorithm is naïve:
  - The algorithm only observes the number of possible cover texts, however, the nature of that traffic is equally important, If the majority of traffic is HTTPS but it all goes to a local proxy then HTTPS traffic to a different destination is suspicious.
  - This is not to say that the proposed algorithm is poor, it is still effective at evaluating protocols based on the environment.
- Managing the quality of channels:
  - The framework takes a "dumb" approach to managing channels, if it fails twice in a row it will be blacklisted for some time. This does not manage well with intermittently available channels, these intermittent channels cause the framework to keep resending messages, which is not ideal.
  - A smarter approach to this should factor in the history of a channel's quality, and penalise it appropriately for intermittent failures.

9.3 Ethical considerations

Whilst covert communication systems do have illicit uses, like espionage, In order for entities to be protected from these systems, they must be understood. This project is a step towards understanding these systems, and thus protecting against them.

Bibliography

An ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. RFC 826, November 1982. URL `https://www.rfc-editor.org/info/rfc826`. Last accessed 9th May 2023.

Anon, . Transmission control protocol. RFC 793, September 1981. URL `https://www.rfc-editor.org/info/rfc793`. Last accessed 9th May 2023.

Anon., . Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144, February 1990. URL `https://www.rfc-editor.org/info/rfc1144`. Last accessed 9th May 2023.

Archibald, Rennie & Ghosal, Dipak. Design and analysis of a model-based covert timing channel for skype traffic. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 236–244, 2015. doi: 10.1109/CNS.2015.7346833.

Backs, Peter & Wendzel, Steffen & Keller, Jörg. Dynamic routing in covert channel overlays based on control protocols. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 32–39, 2012.

Bezanson, Jeff & Edelman, Alan & Karpinski, Stefan & Shah, Viral B. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, September 2017. doi: 10.1137/141000671.

Bradner, Scott O. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997. URL `https://www.rfc-editor.org/info/rfc2119`. Last accessed 9th May 2023.

Dalia Nashat, Loay mamdouh. An efficient steganographic technique for hiding data. In *Journal of the Egyptian Mathematical Society*, 2019. doi: 10.1186/s42787-019-0061-6.

Dua, Arti & Jindal, Vinita & Bedi, Punam. Covert communication using address resolution protocol broadcast request messages. In *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 1–6, 2021. doi: 10.1109/ICRITO51393.2021.9596480.

Fatayer, Tamer S. A. Generated un-detectability covert channel algorithm for dynamic secure communication using encryption and authentication. In *2017 Palestinian International Conference on Information and Communication Technology (PICICT)*, pages 6–9, 2017. doi: 10.1109/PICICT.2017.13.

Free Software Foundation, . Packet(7). `https://www.github.com/wireshark/wireshark`, 1999. Last accessed 9th May 2023.

GitHub, Inc, . GitHub - About. `https://github.com/about`, 2023. Last accessed 9th May 2023.

Hood, C.S. & Ji, Chuanyi. Proactive network fault detection. In *Proceedings of INFOCOM '97*, volume 3, pages 1147–1155 vol.3, 1997. doi: 10.1109/INFCOM.1997.631137.

Hughes, Dave. *Steganography & Watermarking*. PhD thesis, University of New South Wales, 2000. URL `https://web.archive.org/web/20040330200849/http://www.cse.unsw.edu.au/~cs4012/hughes.ps`. Last accessed 9th May 2023.

ISI, . Internet protocol. RFC 791, September 1981. URL `https://www.rfc-editor.org/info/rfc791`. Last accessed 9th May 2023.

Kahn, David. *The Codebreakers: The Story of Secret Writing*. 1973. URL `https://doc.lagout.org/security/Crypto/The%20CodeBreakers%20-%20Kahn%20David.pdf`. Last accessed 9th May 2023.

Lampson, Butler. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 10 1973. doi: 10.1145/362375.362389.

McCanne, Steven & Jacobson, Van. The bsd packet filter: A new architecture for user-level packet capture. Technical report, Lawrence Berkeley National Laboratory, 1992. URL `https://www.tcpdump.org/papers/bpf-usenix93.pdf`. Last accessed 9th May 2023.

Merkel, Dirk. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

Owens, Mark. A discussion of covert channels and steganography. `https://sansorg.egnyte.com/dl/EmWUMdvoVP`, 2002. Last accessed 9th May 2023.

Qi, Qilin. *A Study on Countermeasures against Steganography: an Active Warden Approach*. PhD thesis, University of Nebraska - Lincoln, 2013.

Rohith Raj, S & Rohith, R & Minal, Moharir & Shobha, G. Scapy- a powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5, 2018. doi: 10.1109/ICNEWS.2018.8903954.

Rowland, Craig. Covert channels in the tcp/ip protocol suite, 1996. URL `https://firstmonday.org/ojs/index.php/fm/article/view/528/449`. Last accessed 9th May 2023.

Saeb, Magdy & El-Abd, Eman & El-Zanaty, Mohamed E. On covert data communication channels employing dna recombinant and mutagenesis-based steganographic techniques. In *International Conference on Computer Engineering and Applications*, pages 200–206, 2007.

Shehab, Manal & Korany, Noha & Sadek, Nayera. Evaluation of the ip identification covert channel anomalies using support vector machine. In *2021 IEEE 26th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–6, 2021. doi: 10.1109/CAMAD52502.2021.9617790.

Simmons, Gustavus J. *The Prisoners' Problem and the Subliminal Channel*, pages 51–67. Springer US, Boston, MA, 1984. ISBN 978-1-4684-4730-9. doi: 10.1007/978-1-4684-4730-9_5. URL `https://doi.org/10.1007/978-1-4684-4730-9_5`. Last accessed 9th May 2023.

the Git community, . Git. `https://git-scm.com/`, 2023. Last accessed 9th May 2023.

The Tcpdump Group, . libpcap. `https://www.github.com/the-tcpdump-group/libpcap`, 2013. Last accessed 9th May 2023.

Touch, Dr. Joseph D. Updated specification of the ipv4 id field. RFC 6864, February 2013. URL `https://www.rfc-editor.org/info/rfc6864`. Last accessed 9th May 2023.

U.S. Department of Defense, . Trusted computer system evaluation criteria, 1985. URL `https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/dod85.pdf`. Last accessed 9th May 2023.

Wendzel, Steffen & Keller, Jörg. Design and implementation of an active warden addressing protocol switching covert channels. 06 2012.

Wilhelm, Rene. Ipv6 10 years out: An analysis in users, tables, and traffic. 2022. URL `https://labs.ripe.net/author/wilhelm/ipv6-10-years-out-an-analysis-in-users-tables-and-traffic/`. Last accessed 9th May 2023.

Wolf, Manfred. Covert channels in lan protocols. In Berson, Thomas A. & Beth, Thomas, editors, *Local Area Network Security*, pages 89–101, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-46802-8.

Yarochkin, Fedor V. & Dai, Shih-Yao & Lin, Chih-Hung & Huang, Yennun & Kuo, Sy-Yen. Towards adaptive covert communication system. In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 153–159, 2008. doi: 10.1109/PRDC.2008.26.

Zhu, Yan & Yu, MengYang & Hu, HongXin & Ahn, Gail-Joon & Zhao, HongJia. Efficient construction of provably secure steganography under ordinary covert channels. In *Science China Information Sciences*, volume 55, 2012. doi: 10.1007/s11432-012-4598-3.

Appendix A

Ethics approvals

A.1 Cyber Risk Approval

Cyber Risk Approval

Student Name:	Oscar Cornish	Date:	23/11/2022
Student ID:	u2053390	Supervisor:	Peter Norris
Project Title:		Exploring adaptive covert communication channels	

Brief description of the proposed research activity and methodology

Creating a simulated network environment including miscellaneous background traffic and communication between two hosts, across multiple experiments the communication between these hosts will sometimes contain hidden covert traffic – I will then analyse the collected traffic and use various techniques to see if the covert traffic is detectable (e.g. snort rules of varying degrees of specificity).

Confirm that your project has taken account of/does not contravene the following:

Computer Misuse Act	Yes
GDPR and the Data Protection Act	Yes

Please specify what risks have you identified and list the mitigations you will put in place to reduce the risks to acceptable levels.

I will be monitoring network traffic, so I will set up a private network with a firewall separating my network from any traffic not directly related to my project.

Signature:

OCornish

A.2 WMG Supervisor Delegated Ethical Approval

Biomedical and Scientific Research Ethics Committee (BSREC):

WMG Supervisor Delegated Ethical Approval (WMG-SDA)

Students and supervisors can only make effective judgements about research ethics once the formal methods have been defined. The student should work with support from the supervisor to define a detailed methodology and once this is drafted they can complete an ethical application. This SDA form must be submitted to your approved project supervisor in conjunction with a draft of your Research Methodology chapter before any interaction with humans as research participants can take place. The supervisor must then submit this for processing by the relevant admin team and wait until final approval has been given (by the course management team) BEFORE data collection can take place. Please be aware that a supervisor sign-off does not always guarantee approval will be given.

Full instructions for ethical approval can be found on the project ethics website for your course; see links at bottom of page below.

Instructions for submission of this form:

This is a Word form; so please just click on the square tick boxes for the correct answer and they will automatically change to a 'tick'. To un-tick the box just click it a second time. There are some mandatory rectangular boxes that are highlighted in blue, with the optional boxes in grey. These should be double-clicked and the 'default text' box completed with relevant text, when required.

Once the form is complete, students should append it to a draft of their methodology/ proposal information for supervisors and the department to review. FTMSc and UG students should submit the required document via the relevant Tabula methodology/ proposal submission. All other students (overseas/ part-time PG) should email their completed form directly to their supervisor along with the relevant supporting documentation for approval. Hard copies will not be accepted and electronic copies of all documents should be kept by both the student and the supervisor. Data collection ***must not*** take place until ethical approval has been confirmed (or waived) by email. You must therefore wait until you receive an email from the relevant admin office (see below) to ensure your ethical approval has been fully processed before starting any formal data collection. You will then either receive either an ethical approval reference number, or an ethical approval waiver email, either of which must be kept and produced at the time of dissertation submission. Any dissertation found to contain data that has been collected without gaining appropriate ethical approval may be subject to penalties, usually including failure of that dissertation.

Students should not send ethical approval forms directly to the admin team, as this must come via your research supervisor (or Tabula for FTMSc and UG students). Admin teams will only process forms that have already been signed by your research supervisor, or uploaded to an intranet-based web-form (which takes the place of on e-signature). Instructions for project supervisors about where to submit forms can be found here: **https://warwick.ac.uk/fac/sci/wmg/intranet/student/deptguidelines/academic/ethics/**

For further guidance about the ethics approval process, please refer to your specific ethics pages on the student intranet by following the links below. Admin office contact details for support are also given below.

- **Full-time Masters students**: **wmg-FTMasters@warwick.ac.uk**
- **Part-time Masters students**: **WMGPTProgrammesTeam@warwick.ac.uk**
- **Undergraduate students**: **wmgUGoffice@warwick.ac.uk**

- **[Overseas students (any centre)](#): wmg-overseas@warwick.ac.uk**

SECTION 1. APPLICANT AND COLLABORATION DETAILS

1.1 RESEARCHER

Researcher's Title (optional): Mr **Researcher's Forename:** Oscar
Researcher's Surname: Cornish **Researcher's Student ID number:** u2053390
Warwick e-mail address: u2053390@live.warwick.ac.uk

Researcher's Status:

Undergraduate Student ☒ **Name of course/qualification:** Cyber Security
Taught Postgraduate Student ☐

Research Training

Has the researcher has completed the compulsory Information Security Smart training course (OR completed both the GDPR AND the Information Security Essentials courses which were running up until Feb 2022)? Please append evidence of course completion to this application.

Yes ☒ **No** ☐ **If yes, insert date of completion: 23/11/2022**

Has the researcher has completed the Epigeum online research integrity training course? The short course is compulsory for all researchers, and the full course is strongly recommended for any researchers collecting data from or about human participants. The 'export control' additional module is also strongly recommended.

Yes ☒ **No** ☐ **If yes, insert date of completion: 23/11/2022**

1.2 SUPERVISOR – MUST BE COMPLETED FOR ALL STUDENT PROJECTS

Supervisor's Forename: Peter **Supervisor's Surname:** Norris
Supervisor's Post: Associate Professor
Supervisor's Faculty/School and Department: WMG Cyber Security
Supervisor's Warwick e-mail address: pn@warwick.ac.uk

1.3 OTHER INVESTIGATORS/COLLABORATORS (INTERNAL & EXTERNAL)

Please list all other known collaborators, internal and external to Warwick, including the name of the company/organisation or Investigator's Warwick department/school and their role in the project. If none, please write 'none': none

1.4 RESEARCH CONDUCTED OUTSIDE OF THE UK (or student's main study location).

When collecting data in countries other than the student's main study location, there is added risk that the researcher may overlook research-related laws in that local country (or state) which govern the collection of research data in that country. The responsibility for finding, understanding and adhering to these local laws and research governance regulations (in addition to the usual UK and University policies on conducting research) lies on the researcher and their supervisor. Please see the overseas research webpage for more information: https://warwick.ac.uk/fac/sci/wmg/ftmsc/project/ethics/overseas/

Will this project…	Yes	No
1.4a ... involve the researcher collecting any primary/ new data from participants located overseas from the student's main study location (whether they travel there in person or not)? **If YES, insert countries where data will be collected:** **If YES, please confirm here that the researcher has read, understood and will adhere to ALL local research laws/ policies**	☐ ☐	☒
1.4b ... Will this project involve the researcher (or any collaborators) travelling overseas from the student's main study location (i.e. outside of the UK for UK-based students, or your country of study for overseas students)? Please be aware that the University currently advises against this. **If YES to 1.4b, continue to Section 1.5; if NO then please proceed to Section 2.**	☐	☒

1.5 OVERSEAS TRAVEL DECLARATIONS

The University currently does not advise that taught students travel overseas from their main study location to collect research data, as there are now many virtual/ online ways to achieve this. In exceptional circumstances this MAY be allowed, but various conditions must first be met.

Before you go any further with this form please email a detailed rationale for needing to travel overseas, signed off by the project supervisor, to wmg-ft-projects@warwick.ac.uk and ensure that this has been approved before going any further with this ethical approval application.

Insert countries to be visited:

When travelling overseas to conduct research ALL travellers MUST adhere to the **Travel Risk Management** policy carry out a risk assessment and have this signed off by their supervisor prior to booking any travel. It is the traveller's responsibility to ensure that this form is completed, that they are covered by appropriate insurance, and MUST submit the evidence that they have approval to conduct overseas research as part of this ethical approval application. This is likely to delay your ethical approval whilst the forms are being checked. Please append to this application your full rationale and completed risk assessment for needing to travel overseas.

Please confirm here that the researcher has read, understood and will adhere to the University Travel Risk Management policy **Yes** ☐

Please confirm here that you have read and comply with the Export Controls Policy **Yes** ☐

Please confirm here that you have travel and/ or research-related insurance to cover your research activities, approved by your project supervisor **Yes** ☐

SECTION 2. PROJECT SUMMARY

2.1 Proposed Project Title:	Exploring adaptive covert communication channels
2.2 Suggested Data Collection Start Date for Project (insert N/A if not collecting any new data):	23/12/2022
2.3 Likely Project Completion Date:	1/06/2023

2.4 Type of project (see more info here):

Audit/ Clinical Audit	☐	**Primary data collection (*including the use of social media*)**	☒
Service evaluation or development in health or social care	☐	**Secondary analysis of previously anonymised data**	☐
Literature review only	☐	**Secondary analysis of publicly available data**	☐
		Other (please specify):	

2.5 If primary data collection was ticked above, what is the proposed methodology (tick all relevant methods):

Interview	☐	**Experiment (*with participants*)**	☐
Qualtrics Online Survey	☐	**Experiment (*no participants*)**	☒
Paper-based survey	☐	**Use of social media**	☐
Focus group	☐	**Software evaluation/ software testing** (NB: *if only using software to analyse data collected in other ways, do not tick here*)	☒
Action research/ an intervention	☐	**Other (please specify):**	

SECTION 3. IS ETHICAL APPROVAL NECESSARY?

Does this project…	Yes	No
3.1 … involve collecting any new/ primary data, from or about living participants (including yourself), i.e. data other than that which is already available in the public domain? *NB: This will include all projects using interview or survey data, or any other data containing personally identifiable information*	☐	☒
3.2 … involve analysing primary or unpublished data from, or about, living human beings?	☐	☒
3.3 … involve collecting or analysing primary or unpublished data about people who have recently died *(NB: most projects would not normally do this)*?	☐	☒
3.4 … involve collecting or analysing primary or potentially sensitive unpublished	☐	☒

data about or from companies, organisations or agencies (*e.g. company strategies/ policies/ finance/ marketing/ other data*) other than data that is already available in the public domain (*i.e. if the data is available on a company website then tick 'no'*)?		
3.5 … involve analysing secondary data (data you haven't collected yourself) from, or about, living participants that could include personally identifiable information/ data unless other than data that is already available in the public domain?	☐	☒
3.6 … involve using or accessing data from social media (*e.g. to recruit participants, as a data source, as a data collection tool, for communication into focus groups, chat rooms, or interviews)?*	☐	☒

If you have answered YES to ANY of these questions in Section 3, please proceed to Section 5.

If you have answered NO to ALL of these questions:

- **Please complete Section 4 by signing on p4 and then send Sections 1- 4 only to your supervisor for counter-signing**
- **Keep a copy of pages 1- 4 of this form for your records, but once your supervisor has sent this form to the course office, you will both receive an email confirming that ethical approval is not needed. This email can later be used as proof that you have completed this process (and must be included as an appendix in the dissertation).**
- **You do NOT need to complete the rest of this document**

SECTION 4. DECLARATION FOR PROJECTS BASED ON NON-HUMAN RESEARCH OR SECONDARY DATA ONLY

4.1 RESEARCHER/APPLICANT DECLARATION (for projects based on secondary data only)

I undertake to abide by the University of Warwick's Research Code of Practice and other relevant professional and University policies, regulations, procedures and guidelines in undertaking this study; and I understand that to not adhere to such codes may be grounds for disciplinary action.

I respect the University's ethical requirement for students to abide by the General Data Protection Regulation (GDPR) for the storage of data.

I confirm that in carrying out this project no primary data will be collected about or from human participants.

I will immediately suspend research and request a new ethical approval if the project subsequently changes from the information I have declared in this form.

I understand that the BSREC review system grants ethical approval for projects, and that the seeking and obtaining of all other necessary approvals (e.g. any health and safety requirements, travel risk assessments) and permissions prior to starting the project is my responsibility.

Name of Researcher: Oscar Cornish

Signature: **Oscar Cornish** **Date: 23/11/2022**

4.2 SUPERVISOR DECLARATION FOR NON-HUMAN OR SECONDARY RESEARCH PROJECTS

I confirm that the project does not require ethical review as it does not involve human participants, their data or tissue.

I confirm that the research project is viable and the student should have appropriate skills to undertake the project.

I understand that the BSREC review system grants ethical approval for projects, and that the seeking and obtaining of all other necessary approvals (e.g. any health and safety requirements, travel risk assessments, overseas approvals, etc.) and permissions prior to the starting of a project is the responsibility of the student and their supervisor.

Name of Supervisor: Peter Norris

Signature: (*if not submitting on webform*) ………………………………………………… Date:

Research Training Declaration

I confirm that I have undertaken any mandatory ethics training as provided by WMG for Project Supervisors and I understand that Epigeum Research Integrity online training is also strongly recommended for completion by all research supervisors.

WMG Supervisor Ethics Training (*where available*)– Date of completion:

University GDPR/Information Security training – Date of completion:

Have you completed the strongly recommended Epigeum online research integrity training course? Yes ☐ No ☐ If yes, insert date of completion:

SECTION 5. EXTERNAL ETHICAL REVIEW		
NB: Most projects should acquire University approval and should not attempt external review as an alternate route of approval	**Yes**	**No**
5.1 Has the collection of data for this research project already been given ethical approval by any other (internal or external) research ethics committee (e.g. social care, NHS, other University, company ethical process)?	☐	☐
5.2 Are you intending to submit this project for ethical approval to another external research ethics committee?	☐	☐

If you have answered NO to BOTH of these questions, please proceed to Section 6.

If you have answered YES to either of these questions:

- **Please attach any prior ethical approval documentation to these application forms before submitting the forms to your supervisor**
- **Please give full details about which ethics committee is involved in approving this research and the date of the committee approval (or future meeting) here:**
- **Ensure you have notified your supervisor of seeking/ gaining this alternative ethical approval and then contact the relevant course office for further advice (see p1)**

…. then proceed to Section 6

SECTION 6. RESEARCH PARTICIPANTS

6.1 NUMBER OF PARTICIPANTS

Please state the estimated number of research participants:

(i.e. people you are collecting data on, e.g. 1-10, 20-30, 40-60, 100+ etc.

If using company data, give the persons providing that data as the participants. If no participants write 'none')

BREAKDOWN OF PARTICIPANTS

Where applicable, state the breakdown of participants by type and give estimated numbers of each type, e.g. participant type (lecturer, student, company staff, etc.), job title/ course/ level (manager, director, MSc PPM, etc.):

Participant Type:	**Job Title/ Course/ Level**	**Number of participants of this type:**

6.2 RECRUITMENT STRATEGY AND PROJECT CONTENT

Please explain here, in brief, how you intend to find and recruit the participants to the research study (*e.g. using a webgroup, website, forum, social media, email list, family contacts, etc.*): **Please explain here** the nature of the contact required (i.e. what contact, how regular, what format) with the participants (or other people) before, during and after the research project:		
Could the nature of this project recruit (or could the project involve direct contact with) …	Yes	No
6.3 …. children or young people under 18 years of age?	☐	☐
6.4 …. adults who have learning difficulties?	☐	☐
6.5 …. adults who are significantly disadvantaged by an infirmity or disability?	☐	☐
6.6 …. adults who have mental health problems or other medical problems that may impair their cognitive abilities or ability to consent to taking part?	☐	☐
6.7 …. adults who are resident in social care or any medical establishment, or who could reasonably be classed as vulnerable?	☐	☐
6.8 …. adults who are in the custody of the criminal justice system?	☐	☐
6.9 …. any participants with communication difficulties (including difficulties arising from limited facility with the language you will use to ask the questions to the participant)?	☐	☐
6.10 Could this project involve NHS service users, NHS professionals or volunteers, medical data/ tissue, NHS facilities, any medical facility with NHS contracts, or access to past/ present medical records?	☐	☐
6.11 Could this project involve ethnography, observation of participants, or use of video or photos containing identifiable participants, or making any kind of video or photographic recording of participants (for audio recording <u>only</u>, select 'No' and complete Section 7.4)?	☐	☐
6.12 Could this project involve any kind of deception or covert operations by the researchers (i.e. data is collected without the participant's knowledge)?	☐	☐
6.13 Could this project involve data collection/ questions about physical or mental health/ wellbeing/ medical data or other sensitive topics (e.g. criminality, political opinions, religious beliefs, racial origins, sexual life, professional or academic misconduct, trade union membership, etc.)?	☐	☐
6.14 Could this project involve putting participants through any kind of research activity other than a survey, interview, focus group or software evaluation (e.g. an experiment/ intervention, analysis of their movements, gait, an AV/VR experience, measurement of daily activity, taking part in a simulation, etc.)?	☐	☐
6.15 Could this project potentially place participants or the researchers in a dangerous environment, or at risk of physical, psychological or emotional distress or lead to any negative consequences beyond the risks of normal life?	☐	☐
6.16 Could this project involve the researcher or participants breaching any data protection, contractual arrangements (*e.g. terms and conditions for use of software, website, etc*.) or other relevant law (*within the UK or country which data is being collected*) or breach any other terms the researcher or participant has agreed to?	☐	☐
6.17 Could the nature of this project potentially place the participant / researchers in a situation where they are at risk of investigation by the police or security services; or cause them to be subject to any other legal investigation/ obligation?	☐	☐
6.18 Is your research funded by or are you collaborating with a non-UK military organisation?	☐	☐
6.19 Are you transferring (physically, electronically or verbally) any technologies, material, equipment or know-how, to any non-UK organisation, that could be used to	☐	☐

support the design, development, production, stockpiling or use of nuclear, chemical or biological weapons?		
6.20 Are you using a third party (e.g. a friend, family member, company, etc.) to collect or analyse the data on your behalf?	☐	☐

If you have answered NO to ALL of these questions, please proceed to Section 7.

If you have answered YES to ANY of these questions, please do NOT go any further with this ethical form as you may need to go through FULL ethical approval via the University Research Ethics Committee (BSREC), rather than using this student Supervisor Delegated Approval form. Please contact someone senior on your course for further advice to determine who can review your project and which ethical approvals route you will need. If you are a FT PGT student, or you are unsure who to contact, then please email the FT MSc projects team at: wmg-ft-projects@warwick.ac.uk.

SECTION 7. INFORMED CONSENT

NB: Please see full guidance explaining informed consent and your project ethics webpages (see p1 for details) about participant information leaflets and consent forms before completing this section	**Yes**	**No**
7.1 Will evidencable informed consent (written or verbal agreement) be given by participants/ companies before the project data collection begins?	☐	☐
7.2 Will participants/ companies be given a participant information leaflet (PIL) to inform them about the type of data being collected and what will happen to this data during and after the project?	☐	☐
7.3 Does the PIL explain that participants/ companies have the right not to take part, and/ or may withdraw themselves and their data from the study, and at which point that withdrawing data from the study might not be possible, e.g. once the data has already been analysed/ anonymised?	☐	☐
7.4 Will informed consent be obtained for any recording of participants (e.g. audio recording of interviews). NB: Studies involving video or photographic recording cannot be approved under CDA and prior approval must given by BSREC (see section 6).	☐	☐
7.5 Are you able to give the participants/ companies at least 24 hrs notice after provision of a PIL to them giving consent to participate in this research?	☐	☐

If you have answered YES to ALL of these questions, please briefly explain here how you will implement the informed consent procedure for your project:

Now ensure you have included copies of both your Consent Form and Participant Information Leaflet/ PIL (or debriefing leaflet) along with this form and your methodology/ proposal document.

…. then proceed to Section 8.

If you have answered NO to ANY of these questions, please explain here:

- **Why it is essential for the project to be conducted in this way such that is may not follow usual procedures for obtaining participant consent (e.g. this could be an online survey where consent is given at the same time as survey completion)?**
- **How you propose to address any ethical issues arising from any absence of transparency in your data collection method?**

Include copies of any Consent Form, Participant Information Leaflet (PIL), etc. to your methodology/ proposal document, AND

….. then proceed to Section 8.

SECTION 8. RISK OF HARM		
Is there a risk that…	Yes	No
8.1 … your project may lead to physical harm to any participants or researchers?	☐	☐
8.2 … your project may lead to psychological, emotional distress or embarrassment to any participants or researchers (however minor)?	☐	☐
8.3 … your project may place any participants or researchers in potentially dangerous situations or environments?	☐	☐
8.4 … your project may result in harm to the reputation or future employability of any participants, researchers, their employers, or other persons or organisations?	☐	☐

If you have answered NO to ALL of these questions, please proceed to Section 9.

If you have answered YES to ANY of these questions, please explain here:

- **The nature of the risks involved and why these are necessary**
- **How you propose to assess, manage and mitigate any risks**
- **The procedures for arranging that participants understand and consent to the risks and the sources of help they may refer to if they are seriously distressed or harmed as a result of taking part in this project**
- **The arrangements for recording and reporting any adverse consequences of the research**
- **Which country/countries, and locations where the project will be undertaken, e.g. public place, school, company, hospital, University, researcher's office, etc.**

….. then proceed to Section 9

SECTION 9. RISK OF DISCLOSURE		
Is there a risk that…	Yes	No
9.1 … your project may lead participants to disclose evidence of previous criminal convictions or a potential to commit criminal offences?	☐	☐
9.2 … your project may collect information that is likely to be useful to a person committing or preparing an act of terrorism?	☐	☐
9.3 … your project may lead participants to disclose evidence that children or vulnerable adults have or are being harmed or at risk of harm?	☐	☐
9.4 … your project may lead participants to disclose facts about themselves or others that may later lead to distress or harm?	☐	☐
9.5 … your participant(s) may disclose material that could put the researcher at risk of committing an offence with regard to failing to report a suspected crime; such that anonymity of the participants cannot be guaranteed?	☐	☐
9.6 Have you been asked to sign any non-disclosure agreements (NDAs) to conduct this research? NB: please contact the relevant admin team immediately in this case (see p1 for contact details)?	☐	☐

If you have answered NO to ALL of these questions, please proceed to Section 10.

If you have answered YES to ANY of these questions, please explain here:

- **Why it is necessary to take risks of potential or actual disclosure**
- **What actions you would take if such disclosures were to occur**
- **What advice you will take and from whom before taking these actions**
- **What specific information is likely to be collected**
- **What information you will give participants about the possible consequences of disclosing information about information that may lead to risk of harm**

……then proceed to Section 10

SECTION 10. PAYMENT OF PARTICIPANTS		
	Yes	No
10.1 Do you intend to offer participants cash payments or any other kind of incentive or compensation for taking part in your project?	☐	☐
10.2 Is there any possibility that such inducements may cause participants to consent to risks that they might not otherwise find acceptable?	☐	☐
10.3 Is there a possibility that payment or incentive of any kind may skew or bias the data provided by participants?	☐	☐
10.4 Will you inform participants that accepting compensation or incentives does not invalidate their right to withdraw from the project?	☐	☐
If you have answered NO to ALL of these questions, please <u>proceed to Section 11</u>. **If you have answered YES to ANY of these questions, please explain here:** • **The nature of the incentives or amount of payment that will be offered** • **The reasons why it is necessary to offer such incentives** • **Why you consider it ethically and methodologically acceptable to offer incentives** **….. then proceed to Section 11**		

SECTION 11. INTERNET OR SOCIAL MEDIA RESEARCH		
	Yes	No
11.1 Will you use the internet or social media (e.g. WeChat, WhatsApp, Facebook, LinkedIn or similar) to share the link to your Qualtrics survey?	☐	☐
11.2 Will any part of your project involve collecting personal information using the internet, social media (or similar), whether on a public forum or within an application/ app' or social media site?	☐	☐
11.3 Is any of the data you propose to use in this research from within a 'closed group', password protected website/ forum, or other non-public area of the internet?	☐	☐
11.4 Is there a possibility that any information collected using websites/ social media may be from or about '*vulnerable*' participants (see section 6)?	☐	☐
11.5 Is there a possibility that any information collected using websites/ social media may be from or about children (people aged under 18)?	☐	☐
11.6 Is there a possibility that any of the information collected using websites/ social media might be deemed as personally 'sensitive'?	☐	☐
11.7 Could your data collection method involve breaching any application's (or app's) terms and conditions or breach a participant's confidentiality or anonymity arising from the use of electronic media?	☐	☐
If you have answered NO to ALL of these questions, please <u>proceed to Section 12</u>. **If you have answered YES to ANY of these questions, please explain here:** • **How you propose to collect this data on the internet** • **How you propose to get 'consent' from participants for use of their data, or from internet companies to use such data in a research study?** • **How do you propose to ensure that you do not collect data from any participants under 18 years of age accidentally?** • **The terms and conditions of the software/ platform you are using and how you meet those terms (NB: please append the terms and conditions of the software tool/ company to this application after going through them with your supervisor to ensure compliance)** • **Any significant statements within the terms of conditions of the browser/ application/ site you are using to collate your data** • **How you propose to address the risks associated with internet/ social media research, e.g. data is not collected from unintended participants (*please first review your answers to Section 6.1*)**		

….. then proceed to Section 12		
SECTION 12. PROTECTED CHARACTERISTICS		
	Yes	**No**
12.1 Will your project involve collecting information from participants regarding ANY of the nine 'protected characteristics', covered by the UK [Equality Act 2010](), i.e. age, sex/gender, sexual orientation, gender reassignment, disability, marriage and civil partnership, pregnancy and maternity, race, religion or belief?	☐	☐
Have you appended your intended participant-facing questions to this approval form?	☐	☐
If you have answered NO to this question, please <u>proceed to Section 13</u>. **If you have answered YES to this question,** please refer to the [ethics website about protected characteristics]() (that includes best practice for asking these types of questions)**, append ALL questions asked to participants to this application form** and then explain here: • **Why it is necessary to collect information on one of more of these protected characteristics** • **Why it is not possible to avoid asking these questions of your participants** • **How you intend to analyse the data collected using these characteristics (as it is not ethical to collect information like this if you do not need it)** **….. then proceed to Section 13**		
SECTION 13. DATA COLLECTION, USE, STORAGE, CONFIDENTIALITY, SECURITY AND RETENTION		
Please explain whether you intend to fully anonymise (or pseudonymise) participant/ company data and how you expect to secure the confidentiality of the research data collected: **Please give brief details here about data security; before, during and after the data collection (e.g. what security arrangements will be used e.g. passwords on computer files or locked paper cabinets to ensure that data cannot be accessed by any parties other than members of the research team):** **Please give brief details here about your retention of any data (how long will data be retained, in what format, where will it physically be stored, when will it be deleted. Also consider signed consent forms here, they should be kept separately from research data):** **Will any individuals other than the researcher and supervisor be given access to any raw or non-anonymised data? Yes ☐ No ☐** **If YES give the names* and reasons* why these people will need access to this data:** ******Please note that you will need to hold a University approved data sharing/ processing agreement with each party that is external to the University whom data will be shared*		
	Yes	**No**
13.1 Are there any reasons why you cannot make reasonable steps to ensure the full security and confidentiality of any personal, sensitive or confidential data collected for the project?	☐	☐
13.2 Are you intending to directly or indirectly identify any of your participants (or their associated companies/ organisations) within the dissertation (or any other outputs from this project)?	☐	☐
13.3 Is there possibility that confidential information could be traced back to any specific organisation as a result of the way you present your results from this	☐	☐

research?		
13.4 Will any members of the research team retain any personal, sensitive or confidential data after the end of the project, other than fully anonymised data?	☐	☐
13.5 Do you (or any other member of the research team) intend to make use of any confidential information, knowledge, or trade secrets for purposes other than described in this document (i.e. for company reporting, publication, conferences, etc.) as this must be very clear on the PIL and consent forms?	☐	☐
13.6 During the project will any research data be stored or hosted on any non-approved University platforms, for example apps/tools other than Qualtrics, OneDrive, Outlook, Teams, Files.Warwick (this could include Apps, other online survey tools, recruitment tools, cloud hosting tools, etc.)?	☐	☐
13.7 Will data be shared with any person or organisation outside of WMG/ University for processing, e.g. external transcription services, external statistics support, publishing, etc.?	☐	☐

If you have answered NO to all of these questions, proceed to the declaration in Section 14

If you have answered YES to ANY questions from 13.1 to 13.5, please explain the reasons why it is essential to breach normal protocol regarding data integrity, confidentiality, security and retention of research data:

If you have answered YES to 13.6, please provide details of the systems and how they operate:

NB: If you are not using a University approved tool/ software then you may need to contact the **Information Security** team (informationsecurity@warwick.ac.uk) regarding this technology as it may need to go through the Information Assurance workbook approved approval process, see: https://warwick.ac.uk/services/its/servicessupport/software/purchasing

If you have answered YES to 13.7, please give details of sharing arrangements clarifying whether the data will be identifiable, the external organisation to which it will be sent, and what contracts/ arrangements are in place to safeguard the data and ensure the data processors/ controllers will comply with data protection requirements (see the GDPR training module for more information in unsure):

….. then proceed to Section 14

SECTION 14. SIGNATURES AND AUTHORISATION FOR ETHICAL APPROVAL

14.1 RESEARCHER/APPLICANT DECLARATION

I undertake to abide by the University of Warwick's Research Code of Practice and other relevant professional and University policies, regulations, procedures and guidelines in undertaking this study; and I understand that to not adhere to such codes may be grounds for disciplinary action;

I respect the University's ethical requirement "to cause no harm to the participants by collecting or publishing data";

I respect the University's ethical requirement for students "to abide by the UK General Data Protection Regulation (GDPR) for the collection and storage of any personal data";

I confirm that I will carry out the project in the ways described in this form (and associated research methodology submission). I will immediately suspend research and request a new ethical approval if the project subsequently changes from the information I have declared in this form;

I understand that the BSREC review system grants ethical approval for projects, and that the seeking and obtaining of all other necessary approvals and permissions approvals (e.g. any health and safety requirements, travel risk assessments) prior to starting the project is my responsibility.

Name of Researcher:

Signature: ………………………………………………………………….

Date:

NB: Some of kind of Research Methodology or protocol document explaining this research study must be submitted with this form. In addition all participant facing documentation/ information/ questions must be included with this application (see below for examples). There may be more than one type of document for each study, e.g. there may be multiple PIL's for different groups of participants/ types of research.

Please specify below which documents have been submitted with this application:

☐ **Research Methods/ Protocol** (this is now mandatory to include for all student projects)
☐ **GDPR/ Information Security Smart course completion evidence** (mandatory)

☐ Consent form(s)

☐ Participant Information Leaflet(s)	☐ Participant invitation email(s)
☐ Questionnaire/ survey question(s)	☐ Interview schedule/ topic guide (for unstructured interviews)
☐ Poster/ advertisement for study	☐ Data flow map
☐ Data collection form	☐ Risk assessment(s) (Travel/ Health and Safety)
☐ Data management plan	☐ Other, please specify:

14.2 SUPERVISOR DECLARATION AND AUTHORISATION FOR STUDENT PROJECTS

I confirm that I have read this application and will be acting as the ethical reviewer for this project.

I confirm that the project meets the BSREC Criteria for Supervisor Delegated Ethical Approval, in that the project will be undertaken by an undergraduate, or taught postgraduate, student, AND:

- ***the research project involves human participants only via their participation in an interview, focus group, questionnaire, audit/ clinical audit, service evaluation/ development, or the evaluation of software and e-Learning materials***
- ***participants could not be classified as vulnerable or dependent (e.g. they are not receiving health or social care, primary or secondary education, or criminal justice services, etc.),***
- ***the research does not investigate sensitive or intrusive matters (e.g. health status, wellbeing, criminal activity, sexual history, criminality, political opinions, religious beliefs, racial origins, sexual life, trade union membership, etc.);***

I confirm that the project is viable and the student has the appropriate skills to undertake the research. Participant recruitment procedures, including the Participant Information Leaflet(s) (appended to this form) and the process for obtaining informed consent, are appropriate, and the ethical issues arising from the project have been sufficiently addressed in this form (or associated research methodology submission).

I understand that the BSREC review system grants ethical approval for projects, and that the seeking and obtaining of all other necessary approvals and permissions approvals (e.g. any health and safety requirements, travel risk assessments) prior to the starting of a project is the responsibility of the student and their supervisor.

Name of Supervisor:

Signature: (*if not submitting on webform*) ………………………………………………..   Date:

Research Training Declaration

I confirm that I have undertaken any mandatory ethics training as provided by WMG for Project Supervisors and I understand that the concise Epigeum Research Integrity online training is also mandatory for all research supervisors. The 'Export Control' additional module may also be required.

Epigeum online research integrity training course – Date of completion:

WMG Supervisor Ethics Training (*where available*) – Date of completion:

University GDPR/Information Security training – Date of completion:

Appendix B

Code snippets

B.1 rebase_pcap.jl

```
# Take pcap file, and the address range to rebase.
# Args:
#   1 - Pcap file : String
#   2 - Address range to rebase : String
#   3 - New address range : String
#   4 - Excluded addresses : String (csv)

# Example:
#  rebase_pcap.jl test.pcap 192.168.0.0 10.0.0.0
   10.20.30.1,10.20.30.2

# !!! only does /24 ranges at the moment !!!

struct IpAddr
    octet1::UInt8
    octet2::UInt8
    octet3::UInt8
    octet4::UInt8
    IpAddr(s::String) = new(parse.(UInt8, split(s, "."))...)
end

addr(o::Vector{UInt8}) = join(string.(Int64.(o)), ".")

function pcap_addr_stats(pcapf::String,
   range::Vector{UInt8})::Vector{UInt8} # Vector of endpoints
    ref_dict = Dict{UInt8, Int64}()
    # Read pcap as bytes
    pcap = Vector{UInt8}()
    open(pcapf, "r") do f
        readbytes!(f, pcap, typemax(Int64))
    end
    println("Pcap length: $(length(pcap))")
    # search for range in Pcap
```

```
    for i in 1:length(pcap)-length(range)-1
        if pcap[i:i+length(range)-1] == range
            last_octet = pcap[i+length(range)]
            if haskey(ref_dict, last_octet)
                ref_dict[last_octet] += 1
            else
                ref_dict[last_octet] = 1
            end
        end
    end
    # Print stats
    for (oct, count) ∈ ref_dict
        println("Octet: $(addr(vcat(range, oct))), Count: $count")
    end
    return Vector{UInt8}([k for k ∈ keys(ref_dict)])
end

excluded_octects = parse.(UInt8, last.(split.(split(ARGS[4], ","), ".")))
println("Excluded octets: $excluded_octects")

pcapf = ARGS[1]
from = IpAddr(ARGS[2])
to = IpAddr(ARGS[3])
from_range = [from.octet1, from.octet2, from.octet3]
to_range = [to.octet1, to.octet2, to.octet3]

println("\n")

octs = pcap_addr_stats(pcapf, from_range)

remap = Dict{UInt8, UInt8}()

for o ∈ excluded_octects
    while true
        choice = rand(UInt8)
        if choice ∉ octs && choice ∉ values(remap) && choice ∉ excluded_octects
            remap[o] = choice
            break
        end
    end
end


println("\n")

function rebase(pcapf::String, fromrange::Vector{UInt8}, torange::Vector{UInt8}, remap::Dict{UInt8, UInt8})::NTuple{2, String}
```

```
    # Read pcap as bytes
    pcap = Vector{UInt8}()
    open(pcapf, "r") do f
        readbytes!(f, pcap, typemax(Int64))
    end
    range_length = length(fromrange)
    # search for range in Pcap
    for i in 1:length(pcap)-range_length-1
        if pcap[i:i+range_length-1] == fromrange
            last_octet = pcap[i+range_length]
            if last_octet ∈ excluded_octects
                pcap[i+range_length] = remap[last_octet]
            end
            pcap[i:i+range_length-1] = torange
        end
    end
    # Write new pcap
    name = join(vcat("rebased", split.(pcapf, ".")[2:end]), ".")
    rebased_name = "Dirty/" * name
    open(rebased_name, "w") do f
        write(f, pcap)
    end
    pcap_addr_stats(rebased_name, torange)
    return (rebased_name, "Rebased/"*name)
end

(broken_checksum, fixed) = rebase(pcapf, from_range, to_range,
   remap)

run(Cmd(["python3", "fix_checksum.py", broken_checksum, fixed]))
```

B.2 fix_checksum.py

```python
# Fix checksums in a pcap file
# Arg 1: Input pcap
# Arg 2: Output pcap

import sys
import scapy.all as scapy

def null_checksum(packet):
    for layer in (scapy.IP, scapy.TCP, scapy.UDP, scapy.ICMP):
        if packet.haslayer(layer):
            print(layer)
            packet[layer].chksum = None
    return packet

if __name__ == "__main__":
    if len(sys.argv) != 3:
        exit(-1)
    _packets = scapy.rdpcap(sys.argv[1])
    packets = map(null_checksum, _packets)
    scapy.wrpcap(sys.argv[2], packets)
    exit(0)
```

B.3 warden.py

```
import scapy.all as scapy
import sys

# Args 1 : pcap file

def bitstring(d):
    return "{0:b}".format(d)

def tcp_ack_bounce_data(packet):
    if not packet.haslayer(scapy.TCP):
        return None
    else:
        return bitstring(packet[scapy.TCP].ack)

def ip_identification(packet):
    if not packet.haslayer(scapy.IP):
        return None
    else:
        return bitstring(packet[scapy.IP].id)

def get_hosts(packet):
    if not packet.haslayer(scapy.IP):
        return None
    else:
        return (packet[scapy.IP].src, packet[scapy.IP].dst)

def linear(pcap_data):
    channels = {}
    for i, packet in enumerate(pcap_data):
        x = get_hosts(packet)
        if x:
            src, _ = x
            if src[:9] == "10.20.30.":
                t, i = tcp_ack_bounce_data(packet), ip_identification(pa
                if (t and t[:4] == "1111") or (i and i[:4] == "1111"):
                    if src in channels:
                        channels[src].append(i)
                    else:
                        channels[src] = [i]
    return channels

def main():
    packets = scapy.rdpcap(sys.argv[1])
    channels = linear(packets)
    s = 0
    print(f"With {len(packets)} packets:")
```

B.4 filter.py

```
from netfilterqueue import NetfilterQueue
import scapy.all as scapy
import os, sys

os.system("iptables -F")
os.system("iptables -F -t nat")
os.system("iptables -A FORWARD -j NFQUEUE —queue-num 0")

def null_IP_Identification(_packet):
    packet = scapy.IP(_packet.get_payload())
    if packet.haslayer(scapy.IP) and packet[scapy.IP].id != 0:
        print("IP ID: " + str(packet[scapy.IP].id) + " -> 0")
        packet[scapy.IP].id = 0
        packet[scapy.IP].chksum = None
        _packet.set_payload(bytes(packet))
    _packet.accept()

def map_TCP_ACK(_packet):
    packet = scapy.IP(_packet.get_payload())
    if packet.haslayer(scapy.TCP):
        packet[scapy.TCP].ack = 0
        packet[scapy.TCP].chksum = None
        _packet.set_payload(bytes(packet))
    _packet.accept()

if __name__ == "__main__":
    queue = NetfilterQueue()
    # Args can be "map_TCP_ACK" or "null_IP_Identification"
    queue.bind(0, globals()[sys.argv[1]])
    try:
        print("Using filter: '" + sys.argv[1] + "'")
        queue.run()
    except KeyboardInterrupt:
        print("\nFlushing iptables...")
        os.system("iptables -F")
        os.system("iptables -F -t nat")
        print("Exiting...")
        exit(0)
```

Appendix C

Codebase

C.1 FaucetEnv/namespaces.jl

```

FAUCET_PREFIX = "FaucetNS"

cmdify(s::String) = Cmd(String.(split(s, " ")))

function run_str(s::String)::Nothing
    run(cmdify(s))
    return nothing
end

function get_namespaces()::Vector{String}
    return readlines(`ip netns ls`)
end

get_faucet_namespaces() = filter(x -> startswith(x, FAUCET_PREFIX),
   get_namespaces())

ns(name::String) = name == "" ? "" : "ip netns exec
   $FAUCET_PREFIX$name "

function create_namespace(name::String, netns::String="")::Nothing
    run_str("ip netns add $FAUCET_PREFIX$name")
end

function delete_namespace(name::String)::Nothing
    run_str("ip netns del $FAUCET_PREFIX$name")
end

function create_veth_pair(name1::String, name2::String,
   netns::String="")::Nothing
    run_str("$(ns(netns))ip link add $FAUCET_PREFIX$name1 type veth
   peer name $FAUCET_PREFIX$name2")
end
```

```
create_veth_pair(name::String, netns::String="") = create_veth_pair("$(name)a", "$(name)b", netns)

function move_dev(name::String, dest_ns::String, source_ns::String="")::Nothing
    run_str("$(ns(source_ns))ip link set $FAUCET_PREFIX$name netns $FAUCET_PREFIX$dest_ns")
end

function set_dev_address(dev::String, cidr::String, netns::String="")::Nothing
    run_str("$(ns(netns))ip a add $cidr dev $FAUCET_PREFIX$dev")
end

function set_dev_up(dev::String, netns::String="")::Nothing
    run_str("$(ns(netns))ip link set $FAUCET_PREFIX$dev up")
end

function create_bridge(name::String, netns::String="")::Nothing
    run_str("$(ns(netns))ip link add $FAUCET_PREFIX$name type bridge")
end

function add_bridge_device(name::String, bridge::String, netns::String="")::Nothing
    run_str("$(ns(netns))ip link set $FAUCET_PREFIX$name master $FAUCET_PREFIX$bridge")
end

get_devs(netns::String="")::String = read(cmdify("$(ns(netns))ip a"), String)


# +----------------------------+
# | ns: bridge                 |
# |                            |
# |           br0              |
# |          /   \             |
# |     sendb      recvb       |
# |                            |
# +----------------------------+

# +----------------------------+
# | ns: sender                 |
# |                            |
# |          senda             |
# |                            |
# +----------------------------+

# +----------------------------+
```

```
# | ns: receiver                |
# |                             |
# |         recva               |
# |                             |
# +-----------------------------+

# All have FAUCET_PREFIX prepended to them


NAMESPACES = ["bridge", "sender", "receiver", "aux"]

function create_Faucet_env()::Nothing
    @info "Created namespaces." NAMESPACES
    create_namespace.(NAMESPACES)
    @info "Faucet namespaces" get_faucet_namespaces()

    # Create sender veth pairs
    create_veth_pair("senda", "sendb", "sender")
    # Move senderb to bridge namespace
    move_dev("sendb", "bridge", "sender")
    set_dev_address("senda", "10.20.30.3/24", "sender")

    # Create receiver veth pairs
    create_veth_pair("recva", "recvb", "receiver")
    # Move receiverb to bridge namespace
    move_dev("recvb", "bridge", "receiver")
    set_dev_address("recva", "10.20.30.2/24", "receiver")

    # Create aux veth pairs
    create_veth_pair("auxa", "auxb", "aux")
    # Move auxb to receiver namespace
    move_dev("auxb", "bridge", "aux")
    set_dev_address("auxa", "10.20.30.201/24", "aux")

    # Create bridge
    create_bridge("br0", "bridge")

    # Add senderb and receiverb to bridge
    add_bridge_device("sendb", "br0", "bridge")
    add_bridge_device("recvb", "br0", "bridge")
    add_bridge_device("auxb", "br0", "bridge")

    bridge_devs = get_devs("bridge")
    sender_devs = get_devs("sender")
    receiver_devs = get_devs("receiver")

    # Bring devices up
    set_dev_up("senda", "sender")
    set_dev_up("sendb", "bridge")

```

```
    set_dev_up("recva", "receiver")
    set_dev_up("recvb", "bridge")

    set_dev_up("auxa", "aux")
    set_dev_up("auxb", "bridge")

    set_dev_up("br0", "bridge")

    @debug "Devs created and setup" bridge_devs sender_devs
      receiver_devs
    return nothing
end

function teardown()::Nothing
    @info "Deleting namespaces." NAMESPACES
    delete_namespace.(NAMESPACES)
    @info "Faucet namespaces" get_faucet_namespaces()
    return nothing
end
```

C.2 FaucetEnv/Test.jl

```
# sudo -E julia Test.jl

for i=1:256
    run(`bash Test $i`)
end
```

C.3 FaucetEnv/Traffic/rebase_pcap.jl

```
# Take pcap file, and the address range to rebase.
# Args:
#   1 - Pcap file : String
#   2 - Address range to rebase : String
#   3 - New address range : String
#   4 - Excluded addresses : String (csv)

# Example:
#  rebase_pcap.jl test.pcap 192.168.0.0 10.0.0.0
   10.20.30.1,10.20.30.2

# !!! only does /24 ranges at the moment !!!

struct IpAddr
```

```
    octet1::UInt8
    octet2::UInt8
    octet3::UInt8
    octet4::UInt8
    IpAddr(s::String) = new(parse.(UInt8, split(s, "."))...)
end

addr(o::Vector{UInt8}) = join(string.(Int64.(o)), ".")

function pcap_addr_stats(pcapf::String, range::Vector{UInt8})::Vector{UInt8} # Vector of endpoints
    ref_dict = Dict{UInt8, Int64}()
    # Read pcap as bytes
    pcap = Vector{UInt8}()
    open(pcapf, "r") do f
        readbytes!(f, pcap, typemax(Int64))
    end
    println("Pcap length: $(length(pcap))")
    # search for range in Pcap
    for i in 1:length(pcap)-length(range)-1
        if pcap[i:i+length(range)-1] == range
            last_octet = pcap[i+length(range)]
            if haskey(ref_dict, last_octet)
                ref_dict[last_octet] += 1
            else
                ref_dict[last_octet] = 1
            end
        end
    end
    # Print stats
    for (oct, count) ∈ ref_dict
        println("Octet: $(addr(vcat(range, oct))), Count: $count")
    end
    return Vector{UInt8}([k for k ∈ keys(ref_dict)])
end

excluded_octects = parse.(UInt8, last.(split.(split(ARGS[4], ","), ".")))
println("Excluded octets: $excluded_octects")

pcapf = ARGS[1]
from = IpAddr(ARGS[2])
to = IpAddr(ARGS[3])
from_range = [from.octet1, from.octet2, from.octet3]
to_range = [to.octet1, to.octet2, to.octet3]

println("\n")

octs = pcap_addr_stats(pcapf, from_range)

```

```
remap = Dict{UInt8, UInt8}()

for o ∈ excluded_octects
    while true
        choice = rand(UInt8)
        if choice ∉ octs && choice ∉ values(remap) && choice ∉ excluded_octects
            remap[o] = choice
            break
        end
    end
end


println("\n")

function rebase(pcapf::String, fromrange::Vector{UInt8}, torange::Vector{UInt8}, remap::Dict{UInt8, UInt8})::NTuple{2, String}
    # Read pcap as bytes
    pcap = Vector{UInt8}()
    open(pcapf, "r") do f
        readbytes!(f, pcap, typemax(Int64))
    end
    range_length = length(fromrange)
    # search for range in Pcap
    for i in 1:length(pcap)-range_length-1
        if pcap[i:i+range_length-1] == fromrange
            last_octet = pcap[i+range_length]
            if last_octet ∈ excluded_octects
                pcap[i+range_length] = remap[last_octet]
            end
            pcap[i:i+range_length-1] = torange
        end
    end
    # Write new pcap
    name = join(vcat("rebased", split.(pcapf, ".")[2:end]), ".")
    rebased_name = "Dirty/" * name
    open(rebased_name, "w") do f
        write(f, pcap)
    end
    pcap_addr_stats(rebased_name, torange)
    return (rebased_name, "Rebased/"*name)
end

(broken_checksum, fixed) = rebase(pcapf, from_range, to_range, remap)

run(Cmd(["python3", "fix_checksum.py", broken_checksum, fixed]))
```

C.4 FaucetEnv/Faucet/test/t_utils.jl

```
include("../src/utils.jl")

@testset "Utils" begin
    @test IPv4Addr("192.168.0.1").host == 0xc0a80001
    @test string(IPv4Addr("192.168.0.1")) == "192.168.0.1"
    @test IPv4Addr([0xc0, 0xa8, 0x00, 0x01]).host == 0xc0a80001
end
```

C.5 FaucetEnv/Faucet/test/runtests.jl

```
using Test

include("t_utils.jl")


@testset "Whole program" begin
    # Setup
    include("../src/main.jl")
    using ..Inbound: init_receiver, listen
    using ..CovertChannels: covert_methods
    using .Outbound: init_environment, send_covert_payload
    using .Environment: init_queue

    # Init receiver queue first, so it doesn't miss anything
    queue = init_receiver(:local)

    # Then setup the sender
    net_env = init_environment(target, init_queue())
    covert_payload = Vector{UInt8}("Hello covert world!")
    send_covert_payload(covert_payload, covert_methods, net_env)

    # Then listen for the response
    data = listen(queue, covert_methods)

    @test data == covert_payload

end
```

C.6 FaucetEnv/Faucet/docs/make.jl

```
using Documenter, DocumenterMarkdown
using Faucet
```

```

makedocs(
    format = Documenter.HTML(),
    sitename = "Faucet",
    modules = [Faucet]
)

# Documenter can also automatically deploy documentation to gh-pages.
# See "Hosting Documentation" and deploydocs() in the Documenter manual
# for more information.
#=deploydocs(
    repo = "<repository url>"
)=#
```

C.7 FaucetEnv/Faucet/src/Faucet.jl

```
module Faucet

    """
        WARNING!!

        The module structure here is incorrect, and simply for the benefit of autodoc creation
    """

    PADDING_METHOD=:covert

    include("CircularChannel.jl")
    include("constants.jl")
    include("utils.jl")

    module Environment

        #import Layer_type, get_ip_from_dev, IPv4Addr, _to_bytes, CircularChannel
        import ..Layer_type, ..CircularChannel, ..get_ip_from_dev, ..IPv4Addr, .._to_bytes, ..ENVIRONMENT_QUEUE_SIZE

        export init_queue

        include("environment/headers.jl")
        include("environment/query.jl")
        include("environment/bpf.jl")
        include("environment/queue.jl")
        include("environment/env_utils.jl")
```

```

    end

    module CovertChannels

        import ..Layer_type, ..IPv4, ..Network_Type, ..TCP,
    ..Transport_Type, ..CircularChannel, ..MINIMUM_CHANNEL_SIZE
        using ..Environment: Packet, get_tcp_server,
    get_queue_data, get_layer_stats, get_header, get_local_host_count

        export covert_methods

        include("covert_channels/covert_channels.jl")
        include("covert_channels/microprotocols.jl")

    end

    module Outbound

        struct Target end
        target = Target()

        import ..IPv4Addr, ..Network_Type, ..Transport_Type,
    ..Link_Type, ..Ethernet, ..IPv4, ..TCP, ..UDP, ..ARP,
    ..to_bytes, ..ip_address_regex, ..ip_route_regex,
    ..ip_neigh_regex, ..mac, ..to_net, ..._to_bytes,
    ..integrity_check, ..PADDING_METHOD, ..remove_padding,
    ..CircularChannel
        using ..CovertChannels: craft_change_method_payload,
    craft_discard_chunk_payload, craft_sentinel_payload,
    craft_recovery_payload, method_calculations, determine_method,
    covert_method, init, encode
        using ..Environment: Packet, get_socket, sendto,
    await_arp_beacon, get_local_net_host, AF_PACKET, SOCK_RAW,
    ETH_P_ALL, IPPROTO_RAW

        include("outbound/environment.jl")
        include("outbound/packets.jl")

    end

    module Inbound

        import ..MINIMUM_CHANNEL_SIZE, ..integrity_check,
    ..IPv4Addr, ..PADDING_METHOD, ..remove_padding, ..CircularChannel
        using ..Environment: init_queue, local_bound_traffic,
    Packet, get_local_ip
        using ..CovertChannels: SENTINEL, DISCARD_CHUNK,
    couldContainMethod, decode, covert_method, extract_method
        using ..Outbound: ARP_Beacon
```

```

        include("inbound/listen.jl")

    end

end # module Faucet
```

C.8 FaucetEnv/Faucet/src/CircularChannel.jl

```
# Define a "CircularChannel", a thread-safe channel, that will
  overwrite the oldest data when full
# Heavily inspired by:
#
  https://github.com/JuliaCollections/DataStructures.jl/blob/master/src/circul

"""
    CircularChannel{T}(sz::Int) where T
```
A thread-safe channel, that will overwrite the oldest data when full
Implements:
    - put!(cc::CircularChannel{T}, data::T)
    - take!(cc::CircularChannel{T})::T
    - length(cc::CircularChannel{T})::Int
    - size(cc::CircularChannel{T})::Tuple{Int}
    - isempty(cc::CircularChannel{T})::Bool
    - convert(::Vector{T}, cc::CircularChannel{T})::Vector{T}
```
"""
mutable struct CircularChannel{T} <: AbstractVector{T}
    capacity::Int
    @atomic first::Int
    @atomic length::Int
    buffer::Vector{T}
    lock::Threads.Condition

    function CircularChannel{T}(first::Int, len::Int,
  buf::Vector{T}) where {T}
        first <= length(buf) && len <= length(buf) || error("Value
  of 'length' and 'first' must be in the buffers bounds")
        return new{T}(length(buf), first, len, buf,
  Threads.Condition())
    end
end

# Short hand
CircularChannel{T}(sz::Int) where T = CircularChannel{T}(1, 0,
  Vector{T}(undef, sz))
```

```julia

# Check bounds
Base.@propagate_inbounds function _buffer_index_checked(cc::CircularChannel, i::Int)
    @boundscheck if i < 1 || i > cc.length
        throw(BoundError(cc, i))
    end
    _buffer_index(cc, i)
end

# Implement the "circular" functionality
@inline function _buffer_index(cc::CircularChannel, i::Int)
    n = cc.capacity
    idx = cc.first + i - 1
    return ifelse(idx > n, idx - n, idx)
end

# Get index and set index are only intended for internal use, only "supported" functions are put! and coverting to a vector

# Override getindex to use our circular functionality
@inline Base.@propagate_inbounds function Base.getindex(cc::CircularChannel, i::Int)
    @lock cc.lock return cc.buffer[_buffer_index_checked(cc, i)]
end

# Override setindex to use our circular functionality
@inline Base.@propagate_inbounds function Base.setindex(cc::CircularChannel, data, i::Int)
    @lock cc.lock cc.buffer[_buffer_index_checked(cc, i)] = data && return cc
end

@inline function Base.put!(cc::CircularChannel{T}, data) where T
    lock(cc.lock)
    try
        data_converted = convert(T, data)
        if cc.length == cc.capacity
            @atomic cc.first = (cc.first == cc.capacity ? 1 : cc.first + 1)
        else
            @atomic cc.length += 1
        end
        @inbounds cc.buffer[_buffer_index(cc, cc.length)] = data_converted
        notify(cc.lock)
        return cc
    finally
        unlock(cc.lock)
    end
```

```julia
end

@inline function Base.take!(cc::CircularChannel{T}) where T
    lock(cc.lock)
    try
        if cc.length == 0
            wait(cc.lock)
        end
        @atomic cc.length -= 1
        return cc.buffer[_buffer_index(cc, cc.length + 1)]
    finally
        unlock(cc.lock)
    end
end

# Define some generic functions
Base.length(cc::CircularChannel) = @atomic cc.length
Base.size(cc::CircularChannel) = (length(cc), )
Base.isempty(cc::CircularChannel) = length(cc) == 0

# Extract the data from the channel, in order
function Base.convert(::Vector{T}, cc::CircularChannel{T}) where T
    lock(cc.lock)
    try
        first = cc.buffer[cc.first:cc.length]
        second = cc.buffer[1:cc.first-1]
        return Vector{T}[first; second]
    finally
        unlock(cc.lock)
    end
end
```

C.9 FaucetEnv/Faucet/src/constants.jl

```julia
#=

    Define program constants

=#

# Program related constants
const ENVIRONMENT_QUEUE_SIZE = 150
const MINIMUM_CHANNEL_SIZE   = 4
```

C.10 FaucetEnv/Faucet/src/covert_channels/microprotocols.jl

```julia
#=

    Micro protcol definitions

    *Protocols can be larger than the smallest channel*
    - But must not rely on it
        - Exception here is for recovering the channel, which is a special case
    - DISCARD_CHUNK for example is padded with payload bits, but if there is not the capacity for it, then it isnt.

    Protocols are passed through the covert channel,
     the first channel used will be the most covert for the time being.

    Main protocols:

        first bit == 0b0 (MP_DATA)
            => Following bits are all data
        first bit == 0b1 (MP_META)
            => Following bits are meta

    Meta protocols:

        Taking smallest as Y bits (including MP_META)
        X = Y - 1 (Ignore the META flag)

        X bits -> 2^x permutations
            1 -> sentinel value
            2^x- 1 -> DISCARD_CHUNK signal
            2^x- 2 -> protocols

        first X bits == "1" ^ X
            => Sentinel value (Start / end communication)
        otherwise
            => index in covert_channels / DISCARD_CHUNK

=#

const SENTINEL = parse(UInt16, "1"^(MINIMUM_CHANNEL_SIZE), base=2)
const DISCARD_CHUNK = SENTINEL - 1

#=

    Payload crafting functions

=#
```

```
"""
Take a unsigned integer (META_PROTOCOL) and pad it as required to fit a given capacity
"""
resize_payload(payload::Integer, capacity::Int)::String = resize_payload(UInt64(payload), capacity)
function resize_payload(payload::Unsigned, capacity::Int)::String
    content = bitstring(payload)[end-MINIMUM_CHANNEL_SIZE+1:end] # Get the last 5 bits
    padding = join([rand(("1","0")) for i ∈ 1:(capacity - length(content))])
    return content * padding # Pad with random bits
end

"""
The DISCARD_CHUNK protocol is the meta_protocol,
but padded with the more payload bits
"""
function craft_discard_chunk_payload(capacity::Int, bits::String, pointer::Int64)::Tuple{Int64, String}
    chunk = bitstring(DISCARD_CHUNK)[end-MINIMUM_CHANNEL_SIZE+1:end]
    pointer_offset = capacity - length(chunk)
    payload = chunk * bits[pointer:pointer+pointer_offset-1]
    @assert length(payload) == capacity
    @assert pointer_offset == length(payload) - length(chunk)
    return pointer_offset, payload
end

"""
SENTINEL payload has no additional data just random padding
"""
craft_sentinel_payload(capacity::Int)::String = resize_payload(SENTINEL, capacity)

"""
Recovery payload has a specific format,
to prevent false positives:
```
| MINIMUM_CHANNEL_SIZE | UInt8 | 4 bits | ...
| verification_length | integrity ⊻ known_host | verification_length % 0x10 | padding
```
REQUIRES `method.capacity >= MINIMUM_CHANNEL_SIZE + 8 + 4`
"""
function craft_recovery_payload(capacity::Int, (verification_length, integrity)::Tuple{Int64, UInt8}, known_host::UInt8)::String
    if capacity < MINIMUM_CHANNEL_SIZE + 8 + 4
```

```julia
        @error "Capacity too small for recovery payload" capacity=capacity
        return ""
    end
    meta = "1" ^ MINIMUM_CHANNEL_SIZE
    meta *= bitstring(integrity ⊻ known_host)
    payload = meta * bitstring(verification_length % 0x10)[end-3:end]
    padding = join([rand(("1","0")) for i ∈ 1:(capacity - length(payload))])
    @info "Recovery payload" vl=verification_length tl=verification_length % 0x10 meta payload
    @assert length(payload) + length(padding) == capacity
    return payload * padding
end

"""
payload structure:
```
| MINIMUM_CHANNEL_SIZE | UInt8 | ...
| Method_index | Offset | padding
```
If the capacity is too small, the offset is assumed to be 0.
This loses the benefit of the offset but allows usability in smaller channels.
"""
function craft_change_method_payload(method_index::Int, offset::UInt8, capacity::Int)::String
    meta = "1" * bitstring(method_index)[end-MINIMUM_CHANNEL_SIZE+2:end]
    meta *= bitstring(offset)
    padding = join([rand(("1","0")) for i ∈ 1:(capacity - length(meta))])
    payload = (meta * padding)[1:capacity]
    midx, key = extract_method(payload)
    @assert midx == method_index
    @assert key == offset
    return payload
end

"""
Remove the method index and offset from the method change payload.
"""
function extract_method(payload::String)::Tuple{Int, UInt8} # Method_index, key
    key = parse(UInt8, lpad(payload[MINIMUM_CHANNEL_SIZE+1:min(MINIMUM_CHANNEL_SIZE+8, end)], 8, "0")[1:8], base=2)
    method_index = parse(Int, payload[2:MINIMUM_CHANNEL_SIZE], base=2)
```

```
    return method_index, key
end

#=

    Micro protocol verification functions

=#

"""
    channel_capacity_check(methods::Vector{covert_method})::Bool

Check that the channel sizes are large enough to fit the
   microprotocols
"""
function
   channel_capacity_check(methods::Vector{covert_method})::Bool
    for method ∈ methods
        if method.payload_size < MINIMUM_CHANNEL_SIZE
            @error "Channel size too small for protocol"
   channel=method.name size=method.payload_size
            return false
        end
    end
    return true
end

"""
    registered_channel_check(methods::Vector{covert_method})::Bool

Check that the number of channels is less than the sentinel value,
   and therefore can be addressed in our microprotocols
"""
function
   registered_channel_check(methods::Vector{covert_method})::Bool
    if length(methods) >= SENTINEL - 1 # -1 for DISCARD_CHUNK
        @error "Too many registered channels"
   channels=length(methods) max=SENTINEL-1
        return false
    end
    return true
end

"""
    channel_match_check(methods::Vector{covert_method})::Bool

Check that the channels in the target match the channels in the
   microprotocols
"""
function channel_match_check(methods::Vector{covert_method})::Bool
```

```
    names = [method.name for method ∈ methods]::Vector{String}
    if length(names) != length(target.channels)
        @error "Channels in target and microprotocols do not match"
    target=target.channels microprotocols=names
        return false
    else
        for i=1:lastindex(names)
            if names[i] != target.channels[i]
                @error "Channels in target and microprotocols do
    not match" target=target.channels microprotocols=names index=i
                return false
            end
        end
    end
    return true
end

"""
Check that the microprotocols pass all the channel checks.
see also: [`channel_capacity_check`](@ref),
    [`registered_channel_check`](@ref),
    [`channel_match_check`](@ref).
"""
function check_channels(methods::Vector{covert_method})::Bool
    return all([
        channel_capacity_check(methods),
        registered_channel_check(methods),
        channel_match_check(methods)
    ])
end
```

C.11 FaucetEnv/Faucet/src/covert_channels/covert_channels.jl

```
#=

    Constant definitions

=#

const TCP_ACK           = 0x0010
const TCP_SYN           = 0x0002
const TCP_SYN_ACK       = 0x0012
const TCP_PSH_ACK       = 0x0018

#=

    Covert channels
```

```

=#

struct covert_method{Symbol}
    name::String # Readable name
    layer::Layer_type # Layer it exists on
    type::String # What packet type are we aiming for? (Packet will live at .layer)
    covertness::Int8 # 1 - 10
    payload_size::Int64 # bits / packet
    covert_method(name::String, layer::Layer_type, type::String, covertness::Int64, payload_size::Int64)::covert_method{Symbol} = new{Symbol(name)}(name, layer, type, Int8(covertness), payload_size)
end

"""
Checks if a packet has the structure to contain a covert packet of the given type.
"""
function couldContainMethod(packet::Packet, m::covert_method)::Any
    # Verify the packet could be a part of the covert channel
    return split(string(typeof(get_header(packet, m.layer))), ".")[end] == m.type
end

#=
    TCP_ACK_Bounce abuses the TCP handshake,
    by spoofing the source to the destination and sending a request to a server,
    the server responds with an ACK# of the original packets ISN+1,
    the reciver can then ISN-1 and decode using a predefined technique

    - Target IP (In env)
    - Target Mac (in env)
    - TCP Server Mac (or first gw mac)
    - TCP Server IP
    - TCP Server Port

=#

tcp_ack_bounce::covert_method{:TCP_ACK_Bounce} = covert_method(
    "TCP_ACK_Bounce",
    Layer_type(4), # transport
    "TCP_header",
    3,
    32 # 4 bytes / packet
)

```

```
# Init function for TCP_ACK_Bounce
function init(::covert_method{:TCP_ACK_Bounce},
   net_env::Dict{Symbol, Any})::Dict{Symbol, Any}
    dest_mac, dest_ip, dport = get_tcp_server(net_env[:queue])

    return Dict{Symbol, Any}(
        :payload => Vector{UInt8}(),# ("Covert packet!"), #
   Obviously not a real payload
        :env => net_env,
        :network_type => IPv4::Network_Type,
        :transport_type => TCP::Transport_Type,
        :EtherKWargs => Dict{Symbol, Any}(
            :dest_mac => dest_mac,
            :source_mac =>
   net_env[:src_mac]#net_env[:dest_first_hop_mac]
        ),
        :NetworkKwargs => Dict{Symbol, Any}(
            :source_ip => net_env[:dest_ip].host,
            :dest_ip => dest_ip
        ),
        :TransportKwargs => Dict{Symbol, Any}(
            :flags => TCP_SYN::UInt16,
            :dport => dport
        )
    )
end

# Encode function for TCP_ACK_Bounce
function encode(::covert_method{:TCP_ACK_Bounce}, payload::UInt32;
   template::Dict{Symbol, Any})::Dict{Symbol, Any}
    template[:TransportKwargs][:seq] = payload - 0x1
    return template
end
encode(m::covert_method{:TCP_ACK_Bounce}, payload::String;
   template::Dict{Symbol, Any})::Dict{Symbol, Any} = encode(m,
   parse(UInt32, payload, base=2); template=template)

# Decode function for TCP_ACK_Bounce
decode(::covert_method{:TCP_ACK_Bounce}, pkt::Packet)::UInt32 =
   pkt.payload.payload.payload.header.ack_num

#=
    IPv4_identification utilises the 'random' identification header,
    this can be replaced with encrypted (so essentially random)
   data.
=#

ipv4_identifaction::covert_method{:IPv4_Identification} =
   covert_method(
    "IPv4_Identification",
```

```
    Layer_type(3), # network
    "IPv4_header",
    8,
    16, # 2 bytes / packet
)

# Init function for IPv4_Identification
function init(::covert_method{:IPv4_Identification}, net_env::Dict{Symbol, Any})::Dict{Symbol, Any}
    target_mac, target_ip = net_env[:dest_first_hop_mac], net_env[:dest_ip].host
    return Dict{Symbol, Any}(
        :payload => Vector{UInt8}("Covert packet!"), # Obviously not a real payload
        :env => net_env,
        :network_type => IPv4::Network_Type,
        :transport_type => TCP::Transport_Type,
        :EtherKWargs => Dict{Symbol, Any}(
            :dest_mac => target_mac,
        ),
        :NetworkKwargs => Dict{Symbol, Any}(
            :dest_ip => target_ip,
        )
    )
end

# Encode function for IPv4_Identification
function encode(::covert_method{:IPv4_Identification}, payload::UInt16; template::Dict{Symbol, Any})::Dict{Symbol, Any}
    template[:NetworkKwargs][:identification] = payload
    return template
end
encode(m::covert_method{:IPv4_Identification}, payload::String; template::Dict{Symbol, Any})::Dict{Symbol, Any} = encode(m, parse(UInt16, payload, base=2); template=template)

# Decode function for IPv4_Identification
decode(::covert_method{:IPv4_Identification}, pkt::Packet)::UInt16 = pkt.payload.payload.header.id

"""
Array of all covert methods, the order of these must be the same between sender & target
"""
covert_methods = Vector{covert_method}([
    tcp_ack_bounce,
    ipv4_identifaction,
])

```

```
"""
```markdown
Perform calculations for each covert_method based on the
   environment.

Note:
 - Blacklisted methods are penalised (-90% score)
 - Current method is encouraged (+10% score)
```
"""
function method_calculations(covert_methods::Vector{covert_method},
   env::Dict{Symbol, Any}, Eₚ::Vector{Int64}=[],
   current_method::Int64=0)::NTuple{2, Vector{Float64}}
    # Get the queue data
    q = get_queue_data(env[:queue])

    # Covert score, higher is better : Method i score = scores[i]
    S = zeros(Float64, length(covert_methods))
    # Rate at which to send covert packets : Method i rate =
   rates[i]
    R = zeros(Float64, length(covert_methods))

    if isempty(q)
        @error "No packets in queue, cannot determine method" q
        return S, R
    end

    #@warn "Hardcoded response to determine_method"
    L = [get_layer_stats(q, Layer_type(i)) for i ∈ 2:4]

    # Eₗ : Environment length : Number of packets in queue
    Eₗ = length(q)

    # Eᵣ : Environment rate : (Packets / second)
    Eᵣ = Eₗ / abs(last(q).cap_header.timestamp -
   first(q).cap_header.timestamp)

    # Eₛ : Environment desired secrecy : User supplied (Default: 5)
    Eₛ = env[:desired_secrecy]

    Eₕ = get_local_host_count(q, env[:dest_ip])

    for (i, method) ∈ enumerate(covert_methods)
        Lᵢ_temp = filter(x -> method.type ∈ keys(x), L)
        if isempty(Lᵢ_temp)
            @warn "No packets with valid headers" method.type L
            continue
        end
        # Lᵢ : the layer that method i exists on
        Lᵢ = Lᵢ_temp[1]
```

```
        # Lₛ : The sum of packets that have a valid header in Lᵢ
        Lₛ = +(collect(values(Lᵢ))...)

        # Lₚ : Percentage of total traffic that this layer makes up
        Lₚ = Lₛ / Eₗ

        # Pᵢ is the percentage of traffic
        Pᵢ = Lₚ * (Lᵢ[method.type] / Lₛ)

        # Bᵢ is the bit capacity of method i
        Bᵢ = method.payload_size

        # Cᵢ is the penalty / bonus for the covertness
        #  has bounds [0, 2] -> 0% to 200% (± 100%)
        Cᵢ = 1 - ((method.covertness - Eₛ) / 10)

        # Score for method i
        #  Pᵢ * Bᵢ : Covert bits / Environment bits
        #  then weight by covertness
        #@info "S[i]" Pᵢ Bᵢ Cᵢ Pᵢ * Bᵢ * Cᵢ
        S[i] = Pᵢ * Bᵢ * Cᵢ

        # Rate for method i
        #  Eᵣ * Pᵢ : Usable header packets / second
        #  If we used this much it would be +100% of the environment rate, so we scale it down
        #  by dividing by hosts on the network, Eₕ.
        #  then weight by covertness
        #  We don't want to go over the environment rate, so reshape covertness is between [0, 1] (1 being 100% of env rate)
        #  (Eᵣ * Pᵢ * (Cᵢ / 2)) / Eₕ : Rate of covert packets / second
        #  ∴ 1 / Eᵣ * Pᵢ * (Cᵢ / 2) : Interval between covert packets
        #@info "R[i]" Eₕ Eᵣ Pᵢ Cᵢ/2 Eₕ / (Eᵣ * Pᵢ * (Cᵢ / 2))
        R[i] = Eₕ  / (Eᵣ * Pᵢ * (Cᵢ / 2))
    end

    # Eₚ (arg) : Environment penalty : Penalty for failing to work previously
    for i ∈ Eₚ
        S[i] *= 0.1 # 10% of original score
    end

    current_method != 0 && (S[current_method] *= 1.1) # Encourage current method (+10%)

    return S, R
end
```

```julia

"""
Return the index of the method with the highest score, and the interval to send packets at.

The calculations are done in [`method_calculations`](@ref)
"""
function determine_method(covert_methods::Vector{covert_method}, env::Dict{Symbol, Any}, penalities::Vector{Int64}=[], current_method::Int64=0)::Tuple{Int64, Float64}
    # Determine the best method to use
    S, R = method_calculations(covert_methods, env, penalities, current_method)

    # i : index of best method
    # Sᵢ : score of best method
    Sᵢ, i = findmax(S)
    # Rᵢ : rate of best method
    Rᵢ = R[i]

    if allequal([S..., 0.0])
        # If all scores are 0 then we have no valid methods, default to 1, with a large time interval
        return 1, 100.0
    end
    # @debug "Determined covert method" covert_methods[i].name score=Sᵢ rate=Rᵢ

    # Sort scores by second value in pair (score) and return highest
    return i, Rᵢ
end
```

C.12 FaucetEnv/Faucet/src/inbound/listen.jl

```julia
using AES
using Dates
import Base: -, >

# These functions are to remove bulky lines from the listen function
function -(a::Tuple{Float64, Int64}, b::Tuple{Float64, Int64})::Tuple{Float64, Int64}
    # (a[1] - b[1], a[2] - b[2]) => a - b
    return (a[1] - b[1], a[2] - b[2])
end
function >(a::Tuple{Float64, Int64}, b::Tuple{Float64, Int64})::NTuple{2, Bool}
    # (a[1] > b[1], a[2] > b[2]) => a > b
```

```
    return (a[1] > b[1], a[2] > b[2])
end

"""
    dec(data::bitstring)::Vector{UInt8}

Decode a packet using Pre-shared key
 & removing padding
"""
function dec(_data::String)::Vector{UInt8}
    data = remove_padding(_data)

    # Convert to bytes
    bytes = Vector{UInt8}()
    if length(data) % 8 != 0
        @error "Data is not a multiple of 8, either recieved additional packets, or missing some"
        error("Data length not a multiple of 8")
    end
    bytes = [parse(UInt8, data[i:i+7], base=2) for i ∈ 1:8:length(data)]

    # Recreate cipher text & cipher
    ct = AES.CipherText(
        bytes,
        target.AES_IV,
        length(target.AES_PSK) * 8,
        AES.CBC
    )
    cipher = AES.AESCipher(;key_length=128, mode=AES.CBC, key=target.AES_PSK)

    decrypted = decrypt(ct, cipher).parent

    padding = decrypted[end]
    if decrypted[end-padding+1:end] == [padding for i in 1:padding]
        return decrypted[1:end-padding]
    else
        return decrypted
    end
end

# Get queue with filter

"""
```markdown
Initialise our receiver:
- `:local` => Filter to only local traffic
- `:all` => Filter to all traffic
```
```

```
"""
function init_receiver(bpf_filter::Union{String, Symbol})::CircularChannel{Packet}
    if bpf_filter == :local
        bpf_filter = local_bound_traffic()
    end
    if bpf_filter == :all
        bpf_filter = ""
    end
    @debug "Initializing receiver" filter=bpf_filter
    if typeof(bpf_filter) != String
        throw(ArgumentError("bpf_filter must be a string, :local, or :all"))
    end
    return init_queue(;bpf_filter_string=bpf_filter)
end

"""
Check a packet against all methods, if it matches the format of a method, return the index of the method, else return -1
This check uses the last know verification point, something that both sides know.
"""
function try_recover(packet::Packet, integrities::Vector{Tuple{Int, UInt8}}, methods::Vector{covert_method})::Tuple{Int64, Int64}
    for (i, method) ∈ enumerate(methods)
        if couldContainMethod(packet, method)
            data = bitstring(decode(method, packet))
            if length(data) >= MINIMUM_CHANNEL_SIZE + 12 && data[1:MINIMUM_CHANNEL_SIZE] == bitstring(SENTINEL)[end-MINIMUM_CHANNEL_SIZE+1:end]
                offset = parse(UInt8, data[MINIMUM_CHANNEL_SIZE+1:MINIMUM_CHANNEL_SIZE+8], base=2)
                transmission_length = parse(Int, data[MINIMUM_CHANNEL_SIZE+9:MINIMUM_CHANNEL_SIZE+12], base=2)
                for (len, integrity) ∈ reverse(integrities)[1:min(end, 4)] # Go back max 4 integrities, to be safe
                    @debug "Checking integrity of recovery packet" len=len integrity=integrity
                    if len % 0x10 == transmission_length # The -1 is an artefact of the pointer on the sender side
                        ARP_Beacon(integrity ⊻ offset, IPv4Addr(get_local_ip()))
                        return i, len
                    end
                end
            end
        end
    end
```

```
    return -1, 0 # Not a recovery packet
end

"""
Process a packet contain meta protocols
"""
function process_meta(data::String)::Tuple{Symbol, Any}
    meta = data[1:MINIMUM_CHANNEL_SIZE]
    if meta == bitstring(SENTINEL)[end-MINIMUM_CHANNEL_SIZE+1:end]
        return (:sentinel, nothing)
    elseif meta ==
   bitstring(DISCARD_CHUNK)[end-MINIMUM_CHANNEL_SIZE+1:end]
        return (:integrity_fail, data[MINIMUM_CHANNEL_SIZE+1:end])
    else # Return
        return (:method_change, extract_method(data))
    end
end

"""
Process a packet, returning the type of packet, and the data
"""
function process_packet(current_method::covert_method,
   packet::Packet)::Tuple{Symbol, Any}
    if couldContainMethod(packet, current_method)
        data = bitstring(decode(current_method, packet))
        # check if data or meta
        if data[1] == '0'
            return (:data, data[2:end])
        else
            return process_meta(data)
        end
    end
    return (:pass, nothing)
end

"""
Await a covert communication, and return the decrypted data
"""
function listen(queue::CircularChannel{Packet},
   methods::Vector{covert_method})::Vector{UInt8}
    local_ip = get_local_ip()
    # previous is essentially a revert, if a commited chunks
   integrity fails, we can revert to the previous state
    data, previous, chunk = "", "", ""
    # Track if we have recieved a sentinel
    sentinel_recieved = false
    # Default to the first method
    current_method = methods[begin]
    # Keep track of the number of packets recieved
    packets = 0
```

```

    # Recovery variables
    # (transimission_length, integrity)
    integrities = Vector{Tuple{Int, UInt8}}([(0, 0x0)])
    last_interval_size = Tuple{Float64, Int64}[(20.0, 5)]
    last_interval_point = Tuple{Float64, Int64}[(0.0, 0)]

    # Await sentinel
    @debug "Listening for sentinel" current_method.name
    while true
        current_point = Tuple{Float64, Int64}[(time(), packets)]
        # Take a packet from the queue, to process it
        packet = take!(queue)

        # If we have exceeded the size of the last interval (by packets or time), we should
        recovery = any(((current_point .- last_interval_point) .> last_interval_size)[1])

        # Process the packet, using our current method
        type, kwargs = process_packet(current_method, packet)

        # If we are in recovery mode, and the packet is not a method change (verification)
        if recovery && type != :method_change
            # Try to recover to a new method
            (index, len) = try_recover(packet, integrities, methods)
            if index != -1
                # Recovery successful, reset the current chunk
                chunk = ""
                # Remove data past the last valid recovery point (senders POV)
                data = data[1:len]
                @info "Recovering to new method" method=methods[index].name

                # Incase we go straight into recovery mode
                sentinel_recieved = true
                # Update current method
                current_method = methods[index]
                # Change time of last interval, but don't update the size (recovery)
                last_interval_point = current_point
                # Don't process this packet any further (it could poison our chunk)
                continue
            end
        end

        # Check for sentinel
```

```
        if type == :sentinel
            if sentinel_recieved # If we have already recieved a sentinel, we have finished the data
                break
            end
            @info "Sentined recieved, beginning data collection"
            sentinel_recieved = true

        # We put sentinel_recieved check first to fail fast
        elseif sentinel_recieved && type == :method_change
            (new_method_index, integrity_offset) = kwargs
            @debug "Preparing for method change" new_method_index

            # On method change we confirm the integrity of the chunk, so get it
            integrity = integrity_check(chunk)

            # Beacon out integrity of chunk ⊻'d against the offset we received
            @debug "Sleeping before arp... (5)" integrity=integrity offset=integrity_offset integrity⊻integrity_offset
            sleep(5)
            ARP_Beacon(integrity ⊻ integrity_offset, IPv4Addr(local_ip))

            # Update current method
            current_method = methods[new_method_index]

            # Save our old data, incase this integrity is wrong
            previous = data

            # Append chunk to data
            data *= chunk
            # Reset chunk
            chunk = ""

            # Update integrity list
            push!(integrities, (length(data), integrity))
            # Update last interval size
            last_interval_size = current_point .- last_interval_point
            # Update last interval point
            last_interval_point = current_point

        elseif sentinel_recieved && type == :integrity_fail
            @warn "Integrity check failed of last chunk, reverting..."
            # Reset the chunk, and add the data that follows this metaprotocol
            chunk = kwargs
```

```julia
            # Remove last integrity, it was wrong...
            pop!(integrities)
            # Revert 'commit' of chunk
            data = previous

        elseif sentinel_recieved && type == :data
            @debug "Data received, adding to chunk" chunk_length=length(chunk) total_length=length(data) data=kwargs
            # Append data to chunk
            chunk *= kwargs
            # Increment packets
            packets += 1
        end
    end
    # Append remaining chunk to data (Should be empty due to post-payload verification)
    data *= chunk
    @debug "Data collection complete, decrypting..."
    return dec(data)
end

"""
Listen forever will repeatedly call [`listen`](@ref) on the given queue, and write the data to a unique file
"""
function listen_forever(queue::Channel{Packet}, methods::Vector{covert_method})
    while true
        data = listen(queue, methods)
        file = "comms/$(now().instant.periods.value).bytes"
        @info "Communication stream finished, writing to file" file=file
        open(file, "w") do io
            write(io, data)
        end
    end
end
```

C.13 FaucetEnv/Faucet/src/receiver.jl

```julia

include("main.jl")

using ..Inbound: init_receiver, listen # , listen_forever
using ..CovertChannels: covert_methods

queue = init_receiver(:local)
```

```

@debug "Listening..."

data = listen(queue, covert_methods)

@info "Data recieved" covert_payload=String(data)

sleep(10)
exit(0)
```

C.14 FaucetEnv/Faucet/src/outbound/environment.jl

```
"""
Get the mac address of the interface with the given ip

This is a unstable wrapper around `ip a` and `ip neigh`
"""
function mac_from_ip(ip::String, type::Symbol=:local)::NTuple{6, UInt8}
    if type == :local
        for match ∈ eachmatch(ip_address_regex, readchomp(`ip a`))
            if match[:ip] == ip
                return mac(match[:mac])
            end
        end
    elseif type == :remote
        cmd_output = readchomp(`ip neigh`)
        for match ∈ eachmatch(ip_neigh_regex, cmd_output)
            if match[:ip] == ip
                return mac(match[:mac])
            end
        end
    else
        error("Invalid type: $type")
    end
    error("Unable to find MAC address for IP: $ip")
end
mac_from_ip(ip::IPv4Addr, type::Symbol=:local)::NTuple{6, UInt8} = mac_from_ip(string(ip), type)

"""
Return the interface, gateway, and source ip for the given destination ip
"""
function get_ip_addr(dest_ip::String)::Tuple{String, Union{IPv4Addr, Nothing}, IPv4Addr} # Interface, Gateway, Source
```

```
    for match ∈ eachmatch(ip_route_regex, readchomp(`ip r get
    $dest_ip`))
        if match[:dest_ip] == dest_ip
            iface = string(match[:iface])
            gw = isnothing(match[:gw]) ? nothing :
    IPv4Addr(match[:gw])
            src = IPv4Addr(match[:src_ip])
            return iface, gw, src
        end
    end
end
get_ip_addr(dest_ip::IPv4Addr)::Tuple{String, Union{IPv4Addr,
    Nothing}, IPv4Addr} = get_ip_addr(string(dest_ip))

const arping_regex = r"^Unicast reply from (?:\d{1,3}\.){3}\d{1,3}
    \[(?<mac>(?:[A-F\d]{2}:){5}[A-F\d]{2})\]"m

"""
Get the first hop mac address for the given target ip
"""
function first_hop_mac(target::String, iface::String)::NTuple{6,
    UInt8}
    try
        return mac_from_ip(target, :remote)
    catch e
        x = match(arping_regex, readchomp(`arping -c 1 $target`))
        if !isnothing(x)
            return mac(x[:mac])
        end
        @warn "Unable to find MAC address of $target using arping"
    end
    return nothing
end
first_hop_mac(target::IPv4Addr, iface::String)::NTuple{6, UInt8} =
    first_hop_mac(string(target), iface)

"""
Get the interface for the given ip
"""
function get_dev_from_ip(ip::String)::String
    for match ∈ eachmatch(ip_address_regex, readchomp(`ip a`))
        if match[:ip] == ip
            return string(match[:iface])
        end
    end
    error("Unable to find device for IP: $ip")
end
get_dev_from_ip(ip::IPv4Addr)::String = get_dev_from_ip(string(ip))

"""
```

```
Initialise an environment "context" for the given target
"""
function init_environment(target::Target,
    q::CircularChannel{Packet}, covertness::Int=5)::Dict{Symbol, Any}
     @assert isa(target.ip, IPv4Addr)
     @assert 1 ≤ covertness ≤ 10

     env = Dict{Symbol, Any}()
     # What is the "covertness" we are aiming to achieve?
     env[:desired_secrecy] = covertness
     # Get dest ip as UInt32
     env[:dest_ip] = target.ip
     # Get src ip from sending interface
     iface, gw, src_ip = get_ip_addr(target.ip)
     # Get sending interface + address
     env[:interface] = length(ARGS) ≥ 2 ? ARGS[2] : iface # Override
    interface if specified in args
     env[:src_ip] = src_ip
     # Get mac address from sending interface
     env[:src_mac] = mac_from_ip(env[:src_ip])
     # Get first hop mac address
     env[:dest_first_hop_mac] = isnothing(gw) ?
    first_hop_mac(env[:dest_ip], iface) : first_hop_mac(gw, iface)
     # Target object
     env[:target] = target
     # Queue
     env[:queue] = q
     # Get socket
     env[:sock] = get_socket(AF_PACKET, SOCK_RAW, IPPROTO_RAW)
     return env
end
```

C.15 FaucetEnv/Faucet/src/outbound/packets.jl

```
using AES

# Encrypt a payload, using our PSK and IV
enc(plaintext::Vector{UInt8})::Vector{UInt8} = encrypt(plaintext,
    AESCipher(;key_length=128, mode=AES.CBC, key=target.AES_PSK);
    iv=target.AES_IV).data

# IP checksum
function checksum(message::Vector{UInt8})::UInt16
     checksum = sum([UInt32(message[i]) << 8 + UInt32(message[i+1])
    for i in 1:2:length(message)])
    checksum = ~((checksum & 0xffff) + (checksum >> 16)) & 0xffff
end
```

```julia

# TCP and UDP checksum
function checksum(packet::Vector{UInt8}, tcp_header::Vector{UInt8}, payload::Vector{UInt8})::UInt16
    header_length = length(tcp_header)
    segment_length = header_length + length(payload)
    buffer = zeros(UInt8, 12+segment_length)

    buffer[1:4] = packet[27:30] # Source IP
    buffer[5:8] = packet[31:34] # Destination IP
    buffer[9] = UInt8(0) # Reserved
    buffer[10] = packet[24] # Protocol

    buffer[11:12] = to_bytes(UInt16(segment_length)) # TCP segment length

    for i ∈ 1:length(tcp_header)
        buffer[12+i] = tcp_header[i]
    end

    for i ∈ 1:length(payload)
        buffer[12+header_length+i] = payload[i]
    end
    return checksum(buffer)
end

#=

    Layer 2: Data link

=#

function craft_datalink_header(t::Link_Type, nt::Network_Type, env::Dict{Symbol, Any}, kwargs::Dict{Symbol, Any})::Vector{UInt8}
    if t == Ethernet::Link_Type
        return craft_ethernet_header(nt, env; kwargs...)
    else
        error("Unsupported link type: $t")
    end
end

function craft_ethernet_header(t::Network_Type, env::Dict{Symbol, Any}; source_mac::Union{NTuple{6, UInt8}, Nothing} = nothing, dest_mac::Union{NTuple{6, UInt8}, Nothing} = nothing)::Vector{UInt8}
    header = Vector{UInt8}()
    dest_mac = isnothing(dest_mac) ? env[:dest_first_hop_mac] : dest_mac
    append!(header, dest_mac)
    source_mac = isnothing(source_mac) ? env[:src_mac] : source_mac
```

```julia
    append!(header, source_mac)
    append!(header, to_net(UInt16(t)))
    return header
end

#=

    Layer 3: Network

=#

function craft_network_header(t::Network_Type,
   transport::Transport_Type, env::Dict{Symbol, Any},
   kwargs::Dict{Symbol, Any})::Vector{UInt8}
    if t == IPv4::Network_Type
        return craft_ip_header(transport, env; kwargs...)
    elseif t == ARP::Network_Type
        return craft_arp_header(env; kwargs...)
    else
        error("Unsupported network type: $t")
    end
end

function craft_arp_header(
    env::Dict{Symbol, Any};
    hardware_type::Union{Nothing, UInt16} = 0x0001,
    protocol_type::Union{Nothing, UInt16} = 0x0800,
    hardware_size::Union{Nothing, UInt8} = 0x06,
    protocol_size::Union{Nothing, UInt8} = 0x04,
    operation::Union{Nothing, UInt16} = 0x0001,
    SHA::Union{Nothing, NTuple{6, UInt8}} = nothing,
    SPS::Union{Nothing, NTuple{4, UInt8}} = nothing,
    THA::Union{Nothing, NTuple{6, UInt8}} = (0, 0, 0, 0, 0, 0),
    TPS::Union{Nothing, NTuple{4, UInt8}} = nothing
    )::Vector{UInt8}
    header = Vector{UInt8}()
    append!(header, to_net(hardware_type))
    append!(header, to_net(protocol_type))
    append!(header, to_net(hardware_size))
    append!(header, to_net(protocol_size))
    append!(header, to_net(operation))
    SHA = isnothing(SHA) ? env[:src_mac] : SHA
    append!(header, SHA)
    SPS = isnothing(SPS) ? env[:src_ip] : SPS
    append!(header, SPS)
    append!(header, THA)
    TPS = isnothing(TPS) ? env[:dest_ip] : TPS
    append!(header, TPS)
    return header
end
```

```julia

function ip_checksum(header::Vector{UInt8})::UInt16
    checksum = sum([UInt32(header[i]) << 8 + UInt32(header[i+1]) for i in 1:2:lastindex(header)])
    return ~UInt16((checksum >> 16) + (checksum & 0xFFFF))
end

function craft_ip_header(
            protocol::Transport_Type,
            env::Dict{Symbol, Any};
            version::Union{Nothing, UInt8} = nothing,
            ihl::Union{Nothing, UInt8} = nothing,
            dscp::Union{Nothing, UInt8} = nothing,
            ecn::Union{Nothing, UInt8} = nothing,
            total_length::Union{Nothing, UInt16} = nothing,
            identification::Union{Nothing, UInt16} = nothing,
            flags::Union{Nothing, UInt8} = nothing,
            fragment_offset::Union{Nothing, UInt16} = nothing,
            ttl::Union{Nothing, UInt8} = nothing,
            header_checksum::Union{Nothing, UInt16} = nothing,
            source_ip::Union{Nothing, UInt32} = nothing,
            dest_ip::Union{Nothing, UInt32} = nothing
        )::Vector{UInt8}
    ip_header = Vector{UInt8}()
    version = isnothing(version) ? 0x4 : version
    ihl = isnothing(ihl) ? 0x5 : ihl
    append!(ip_header, to_net(version << 4 | ihl & 0xf))
    dscp = isnothing(dscp) ? 0x0 : dscp
    ecn = isnothing(ecn) ? 0x0 : ecn
    append!(ip_header, to_net(dscp << 2 | ecn & 0x3))
    total_length = isnothing(total_length) ? 0x0000 : total_length
    append!(ip_header, to_net(total_length))
    identification = isnothing(identification) ? rand(UInt16) : identification
    append!(ip_header, to_net(identification))
    flags = isnothing(flags) ? 0b000 : flags & 0b111
    fragment_offset = isnothing(fragment_offset) ? 0x0000 : fragment_offset & 0x1fff
    append!(ip_header, to_net(flags << 13 | fragment_offset))
    ttl = isnothing(ttl) ? 0xf3 : ttl
    append!(ip_header, to_net(ttl))
    append!(ip_header, to_net(UInt8(protocol)))
    _checksum = isnothing(header_checksum) ? 0x0000 : header_checksum
    append!(ip_header, to_net(_checksum))
    source_ip = isnothing(source_ip) ? env[:src_ip].host : source_ip
    append!(ip_header, to_net(to_bytes(source_ip)))
    dest_ip = isnothing(dest_ip) ? env[:dest_ip].host : dest_ip
    append!(ip_header, to_net(to_bytes(dest_ip)))
    return ip_header
```

```
end

#=

    Layer 4: Transport

=#

function craft_transport_header(t::Transport_Type, env::Dict{Symbol, Any}, packet::Vector{UInt8}, payload::Vector{UInt8}, kwargs::Dict{Symbol, Any})::Vector{UInt8}
    if t == UDP::Transport_Type
        return craft_udp_header(payload, env; kwargs...)
    elseif t == TCP::Transport_Type
        return craft_tcp_header(packet, payload, env; kwargs...)
    else
        error("Unsupported transport type: $t")
    end
end

function craft_tcp_header(
            packet::Vector{UInt8},
            payload::Vector{UInt8},
            ::Dict{Symbol, Any};
            sport::Union{Nothing, UInt16} = nothing,
            dport::Union{Nothing, UInt16} = nothing,
            seq::Union{Nothing, UInt32} = nothing,
            ack::Union{Nothing, UInt32} = nothing,
            data_offset::Union{Nothing, UInt8} = nothing,
            reserved::Union{Nothing, UInt8} = nothing,
            flags::Union{Nothing, UInt16} = nothing,
            window::Union{Nothing, UInt16} = nothing,
            _checksum::Union{Nothing, UInt16} = nothing,
            urgent_pointer::Union{Nothing, UInt16} = nothing,
            options::Union{Nothing, Vector{UInt8}} = nothing
        )::Vector{UInt8}
    tcp_header = Vector{UInt8}()
    sport = isnothing(sport) ? rand(UInt16) : sport
    append!(tcp_header, to_net(sport))
    dport = isnothing(dport) ? rand(UInt16) : dport
    append!(tcp_header, to_net(dport))
    seq = isnothing(seq) ? rand(UInt32) : seq
    append!(tcp_header, to_net(seq))
    # Observe flags and alter other fields accordingly (ack, urgent_pointer, etc.)
    if isnothing(flags)
        flags = 0x000
    else
        if flags & 0x10 != 0x0
            ack = isnothing(ack) ? rand(UInt32) : ack
```

```
        end
        if flags & 0x20 != 0x0
            urgent_pointer = isnothing(urgent_pointer) ? rand(UInt16) : urgent_pointer
        end
    end
    ack = isnothing(ack) ? 0x00000000 : ack
    append!(tcp_header, to_net(ack))
    data_offset = isnothing(data_offset) ? 0x05 : data_offset
    reserved = isnothing(reserved) ? 0x000 : reserved
    do_flags = UInt16(data_offset) << 12 | UInt16(reserved) << 9 | UInt16(flags) & 0x01ff
    append!(tcp_header, to_net(do_flags))
    window = isnothing(window) ? 0xffff : window
    append!(tcp_header, to_net(window))
    check = isnothing(_checksum) ? 0x0000 : checksum
    append!(tcp_header, to_net(check))
    urgent_pointer = isnothing(urgent_pointer) ? 0x0000 : urgent_pointer
    append!(tcp_header, to_net(urgent_pointer))
    if !isnothing(options)
        error("No options handling implemented")
    end
    if isnothing(_checksum)
        tcp_header[17:18] = to_net(checksum(packet, tcp_header, payload))
    end
    return tcp_header
end

#=

    Crafting the packets

=#

function craft_packet(;
            payload::Vector{UInt8}, env::Dict{Symbol, Any},
            network_type::Network_Type = IPv4::Network_Type,
            transport_type::Transport_Type = TCP::Transport_Type,
            EtherKWargs::Dict{Symbol, Any} = Dict{Symbol, Any}(),
            NetworkKwargs::Dict{Symbol, Any} = Dict{Symbol, Any}(),
            TransportKwargs::Dict{Symbol, Any} = Dict{Symbol, Any}()
        )::Vector{UInt8}

    packet = Vector{UInt8}()
    #@debug "Crafting packet" p=payload ek=EtherKWargs nk=NetworkKwargs tk=TransportKwargs

    # Craft Datalink header
```

```julia
    dl_header = craft_datalink_header(Ethernet::Link_Type, network_type, env, EtherKWargs)
    append!(packet, dl_header)
    dl_length = length(packet)

    # Get network header
    network_header = craft_network_header(network_type, transport_type, env, NetworkKwargs)
    append!(packet, network_header)


    transport_header = craft_transport_header(transport_type, env, packet, payload, TransportKwargs)
    if network_type == IPv4::Network_Type && packet[17:18] == [0x00, 0x00]
        len = to_net(UInt16(length(network_header) + length(transport_header) + length(payload)))
        packet[dl_length+3:dl_length+4] = len
        # Perform checksum after setting length
    end
    if network_type == IPv4::Network_Type && packet[dl_length+11:dl_length+12] == [0x00, 0x00]
        packet[dl_length+11:dl_length+12] = to_net(checksum(packet[dl_length+1:dl_length+length(network_header)]))
    end
    # Craft Transport header
    append!(packet, transport_header)

    # Append payload
    append!(packet, payload)

    return packet
end

"""
    ARP_Beacon(payload::NTuple{6, UInt8})

Send out a beacon with the given payload. The payload is a tuple of 6 bytes.
"""
function ARP_Beacon(payload::UInt8, source_ip::IPv4Addr, send_socket::IOStream=get_socket(AF_PACKET, SOCK_RAW, IPPROTO_RAW))::Nothing
    src_mac = mac_from_ip(source_ip, :local) # This function is used by the receiver, so the src addr is the target addr
    src_ip = _to_bytes(source_ip.host)
    dst_ip = [src_ip[1:3]...; payload]
    iface = get_dev_from_ip(source_ip)
    @debug "Sending ARP beacon" encoded_byte=payload time()

```

```julia
    packet = craft_packet(
        payload = Vector{UInt8}(),
        env = Dict{Symbol, Any}(),
        network_type = ARP::Network_Type,
        EtherKWargs = Dict{Symbol, Any}(
            :source_mac => src_mac, # Get from regex...
            :dest_mac => (0x00, 0x00, 0x00, 0x00, 0x00, 0x00)
        ),
        NetworkKwargs = Dict{Symbol, Any}(
            :SHA => src_mac,
            :THA => (0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
            :SPS => NTuple{4, UInt8}(src_ip), # Target <- Have from target...
            :TPS => NTuple{4, UInt8}(dst_ip) # first 3 bytes of src_ip, + payload
        )
    )
    # Send packet
    x = sendto(send_socket, packet, iface)
    @assert x == length(packet) "Sent $x bytes, expected $(length(packet))"
    return nothing
end
ARP_Beacon(payload::NTuple{6, UInt8}, source_ip::String) = ARP_Beacon(payload, IPv4Addr(source_ip))


function send_packet(packet::Vector{UInt8}, net_env::Dict{Symbol, Any})::Nothing
    bytes = sendto(net_env[:sock]::IOStream, packet, net_env[:interface]::String)
    @assert (bytes == length(packet)) "Sent $bytes bytes, expected $(length(packet))"
    return nothing
end

# The below functions are just wrappers of send_packet (sendto) and their respective functions in covert_channels

function send_packet(m::covert_method, net_env::Dict{Symbol, Any}, payload::String, template::Dict{Symbol, Any})::Nothing
    return send_packet(craft_packet(;encode(m, payload; template)...), net_env)
end

function send_sentinel_packet(m::covert_method, net_env::Dict{Symbol, Any}, template::Dict{Symbol, Any})::Nothing
    send_packet(craft_packet(;encode(m, craft_sentinel_payload(m.payload_size); template)...), net_env)
end
```

```julia

function send_recovery_packet(m::covert_method, last_verification::Tuple{Int64, UInt8}, known_host::UInt8, net_env::Dict{Symbol, Any}, template::Dict{Symbol, Any})::Nothing
    send_packet(craft_packet(;encode(m, craft_recovery_payload(m.payload_size, last_verification, known_host); template)...), net_env)
end

function send_method_change_packet(m::covert_method, method_index::Int, offset::UInt8, net_env::Dict{Symbol, Any}, template::Dict{Symbol, Any})::Nothing
    payload = craft_change_method_payload(method_index, offset, m.payload_size)
    send_packet(craft_packet(;encode(m, payload; template)...), net_env)
end

function send_discard_chunk_packet(m::covert_method, bits::String, pointer::Int64, net_env::Dict{Symbol, Any}, template::Dict{Symbol, Any})::Int64
    pointer_offset, payload = craft_discard_chunk_payload(m.payload_size, bits, pointer)
    send_packet(craft_packet(;encode(m, payload; template)...), net_env)
    return pointer_offset
end

"""
    pad_transmission(payload::bitstring, method::Symbol=:short)::bitstring

    Take the payload and pad it to the nearest byte boundary.

    Methods are:
        :short => Uses minimal padding, but is less covert
        :covert => Uses more padding, but is more covert
"""
function pad_transmission(raw::String, method::Symbol=PADDING_METHOD)::String
    if method == :short
        return raw * "1"
    elseif method == :covert
        # We know our encrypted payload is a multiple of 128 bits, so divide by 128 to get the number of "segments"
        return raw * lstrip(bitstring(Int64(length(raw) / 128)), '0')
    else
        error("Unknown padding method $method")
    end
```

```
end

"""
    pad_packet_payload(packet_payload::bitstring, capacity::Int, transmission::bitstring, method::Symbol=:short)::bitstring

    Pad the packet payload to the size of the capacity, depending on the method.

    Transmission is the orginal, unpadded, bits for transmission (post-encryption), used for verification.

    Handles same methods as ['pad_transmission'](@ref)
"""
function pad_packet_payload(packet_payload::String, capacity::Int, transmission::String, method::Symbol=PADDING_METHOD)::String
    if method == :short
        padding = "0" ^ (capacity - length(packet_payload))
        @assert remove_padding(pad_transmission(transmission) * padding, method) == transmission "Padding is not valid (:short)"
        return packet_payload * padding
    elseif method == :covert
        padding = join([rand(['0', '1']) for _ in 1:capacity - length(packet_payload)])
        # Random padding CANNOT be a valid length of the payload
        #  it's very likely this will happen, but just in case...
        while remove_padding(pad_transmission(transmission, method) * padding, method) != transmission
            padding = join([rand(['0', '1']) for _ in 1:capacity - length(packet_payload)])
        end
        return packet_payload * padding
    else
        error("Unknown padding method $method")
    end
end

"""
Send a packet using an adaptive covert communication channel
"""
function send_covert_payload(raw_payload::Vector{UInt8}, methods::Vector{covert_method}, net_env::Dict{Symbol, Any})
    # Blacklist our current host, as the target may arp us unrelated to the covert channel
    host_blacklist = [UInt8(hton(net_env[:dest_ip].host) & 0x000000ff)]

    # Protocol blacklisting variables:
    # Current packet + `penalty_size` : When it is no longer "blacklisted" (-1 for permenant blacklist)
```

```
    penalty_size = 30
    # Initialise protocol failure counter
    protocol_failures = 0
    # Number of failures until we switch to next method
    max_failures = 2
    # (Protocol, Release_packet)
    protocol_blacklist = Vector{Tuple{Int64, Int64}}()

    # Initialise sender variables:
    # By default, we start with the first method
    current_method_index = 1
    # Get the method from the vector
    method = methods[current_method_index]
    # And initialise it
    method_kwargs = init(method, net_env)
    # Pointer to the current bit we are sending (start with the first)
    pointer = 1
    # Pointer to the start of the current chunk
    chunk_pointer = pointer
    # Max data packets between integrity checks
    integrity_interval = 6
    # Number of packets sent
    packet_count = 0
    # Time to wait for a (challenge) response from the target
    check_timeout = 10
    # The padding we send to the target with our final payload will be part of the integrity check, so store it
    final_padding = ""
    # When we have sent the final payload we go back to perform an integrity check, but we don't want to send packets after so we set this flag
    finished = false
    # Should verify at next oppurtunity?
    should_verify = false

    # Initialise the payload:
    payload = enc(raw_payload)
    # Convert the payload to a bitstring
    _bits = *(bitstring.(payload)...)
    # Pad the transmission, because we may have to append noise to it later (to fit a methods capacity)
    bits = pad_transmission(_bits)

    # Give the environment queue some time to populate, so sleep a bit
    #  If we don't do this then our chosen method will be volatile...
    time_interval = 10
    @warn "Inital time interval " time_interval
```

```
    sleep(time_interval)

    # Recovery mode variables:
    # Whether we are in recovery mode
    recovery_mode = false
    # At 0 packets we know what the payload is (0x00)
    last_verification_packet_count = packet_count
    last_verification_time = time()
    last_integrity = (0, 0x00)
    next_integrity = (0, 0x00)
    # (length (in s), integrity value)
    # Time interval, packet interval
    #  to start, we don't want to go straight into recovery mode, but we do want the possibility - so just add a delay
    #  50 seconds, 6 packets
    recovery_timeouts = (50, integrity_interval)

    @info "Sending sentinel packet" method=method.name
    # Start communication with a sentinel packet
    send_sentinel_packet(method, net_env, method_kwargs)
    while pointer <= lastindex(bits)
        # Remove blacklisted protocols that have served their penalty
        for (idx, (proto, penal)) ∈ enumerate(protocol_blacklist)
            # If the penalty has expired, remove it
            if penal <= packet_count
                @debug "Removing protocol from blacklist" protocol=methods[proto].name
                deleteat!(protocol_blacklist, idx)
            end
        end

        if recovery_mode
            @info "Entering recovery mode" timeouts=recovery_timeouts
            # Wait until the target is in recovery mode 1.5x what we think it should wait, to be sure
            while !(time() - last_verification_time > recovery_timeouts[1] * 1.5 || packet_count - last_verification_packet_count > recovery_timeouts[2] * 1.5)
                sleep(1)
            end
            # Get all the scores of the method, so we can restart communication with the best one
            S, R = method_calculations(methods, net_env, [x[1] for x ∈ protocol_blacklist])
            # Get permutation of scores that puts the indices in order of best to worst
            idxs = sortperm(S)
```

```
        # Iterate through methods until we find one that works, it will respond and verify...
        while !isempty(idxs)
            i = pop!(idxs)
            # Pop the next highest method, and sleep its inteval time
            # We don't want to ignore covertness because we are in recovery...
            sleep(R[i])
            # get a known host for the integrity check
            known_host = get_local_net_host(net_env[:queue], net_env[:src_ip], host_blacklist)

            # Update the method, and initialise it
            method = methods[i]
            method_kwargs = init(method, net_env)

            # Send recovery packet
            send_recovery_packet(method, last_integrity, known_host, net_env, method_kwargs)
            @debug "Attempting to recover with method" method=methods[i].name

            # Allow a larger timeout here, as the recovery process is a bit longer.
            if await_arp_beacon(net_env[:dest_ip], known_host, check_timeout*4)
                # Success, we have recovered, so update the variables
                current_method_index = i
                pointer = last_integrity[1] + 1
                chunk_pointer = pointer
                recovery_timeouts = (time() - last_verification_time, packet_count - last_verification_packet_count)
                last_verification_time = time()
                last_verification_packet_count = packet_count
                should_verify = true
                sleep(R[current_method_index])
                break # Don't need to try any more methods
            end
        end
        if last_verification_packet_count != packet_count && isempty(idxs) # If we exhausted all methods, end communication
            error("Failed to recover with any method")
        end
        @info "Recovered with method" method=methods[current_method_index].name

```

```
        # Iterate through methods until we find one that works, it will respond and verify...
        recovery_mode = false
    end

    # Determine method to use, strip the penalties from the protocol blacklist, it doesn't need them
    # Give it the current method to discourage it from switching (using a 10% bonus to the score)
    if protocol_failures == 0
        (method_index, time_interval) = determine_method(methods, net_env, [x[1] for x ∈ protocol_blacklist], current_method_index)
    else
        method_index = current_method_index
    end

    # If we have changed method, or are due an integrity check
    if method_index != current_method_index || packet_count % integrity_interval == 0 || should_verify

        # This label is jumped to later (for our final integrity check)
        @label verify

        # Don't send back to back packets
        sleep(time_interval)

        # Get the current integrity, if we have used final padding, append it
        integrity = integrity_check(bits[chunk_pointer:pointer-1] * final_padding)
        # Get a known host for the integrity check
        known_host = get_local_net_host(net_env[:queue], net_env[:src_ip], host_blacklist)

        # Send the challenge
        send_method_change_packet(method, method_index, integrity ⊻ known_host, net_env, method_kwargs)

        # Switch methods
        method = methods[method_index]

        if method_index != current_method_index
            @info "Switched method, performing integrity check" method=method.name interval=time_interval
            protocol_failures = 0
        end

        method_kwargs = init(method, net_env)
```

```
        current_method_index = method_index
        if packet_count % integrity_interval == 0
            @debug "Performing regular interval integrity check" interval=integrity_interval
        elseif !should_verify
            @debug "Final integrity check"
        end

        should_verify = false

        # Await the response
        if await_arp_beacon(net_env[:dest_ip], known_host, check_timeout) # Success
            # reset failures
            protocol_failures = 0
            @debug "Integrity check passed" method=method.name integrity known_host integrity ⊻ known_host
            # reset chunk pointer
            chunk_pointer = pointer
            # update last verified integrity (+ length)
            last_integrity = next_integrity
            next_integrity = (chunk_pointer-1, integrity)
        else # Failure, resend chunk
            @warn "Failed integrity check" method=method.name integrity known_host integrity ⊻ known_host
            # Increment failures
            protocol_failures += 1
            # If we have failed too many times, blacklist the protocol
            if protocol_failures == max_failures
                push!(protocol_blacklist, (method_index, packet_count + penalty_size))
                @info "Blacklisting protocol" protocol=method.name
                protocol_failures = 0
                # Enter recovery mode
                recovery_mode = true
            end
            # Reset pointer to beginning of chunk
            pointer = chunk_pointer
            # Tell the target to discard the chunk, and send some data with it too
            pointer += send_discard_chunk_packet(method, bits, pointer, net_env, method_kwargs)
            # We aren't finished anymore (if we were before)
            finished = false
            final_padding = ""
        end
        # If we have finished, we can just exit here
        if finished
```

```
                break
            end
        end
        # Send payload packet
        if !finished && pointer+method.payload_size-1 > lastindex(bits)
            # For last part of payload, we will need to pad the remaining capacity
            payload = pad_packet_payload("0" * bits[pointer:lastindex(bits)], method.payload_size, _bits)
            @debug "Packet covert payload (Without MP)(FINAL)" payload=bits[pointer:lastindex(bits)] chunk_length=pointer-chunk_pointer total_sent=pointer
            payload_length = length(pointer:lastindex(bits))
            pointer += payload_length - 1
            # Save the final padding, so we can append it to the integrity check
            final_padding = payload[payload_length+1:end]
            # Set finished flag so we jump to verification stage
            finished = true
        elseif !finished
            # For all other packets, we can just send the payload
            payload = "0" * bits[pointer:pointer+method.payload_size-2]
            @debug "Packet covert payload (Without MP)" payload=bits[pointer:pointer+method.payload_size-2] chunk_length=pointer-chunk_pointer total_sent=pointer
            pointer += method.payload_size-1
        end
        # Send packet
        send_packet(method, net_env, payload, method_kwargs)
        # If that was the last packet, jump to verification stage
        if finished
            @debug "Payload sent, verifiying integrity"
            # This takes us to the `@label verify` line
            @goto verify
        end
        # Increment packet count
        packet_count += 1
        # Sleep for the time interval
        sleep(time_interval)
    end
    # Send sentinel packet to end communication
    send_sentinel_packet(method, net_env, method_kwargs)
    @debug "Endded communication via SENTINEL" via=method.name
end
```

C.16 FaucetEnv/Faucet/src/sender.jl

```

include("main.jl")

using ..CovertChannels: covert_methods
using .Environment: init_queue
using .Outbound: init_environment, send_covert_payload

@debug "Creating queue"
net_env = init_environment(target, init_queue())

using Random
alphabet = collect('A':'Z')
rng = MersenneTwister(1234)
covert_payload = UInt8.([alphabet[rand(rng, 1:26)] for i =
    1:parse(Int64, ARGS[3])])

@debug "Sending covert payload" payload=covert_payload

send_covert_payload(covert_payload, covert_methods, net_env)

@info "Finished sending covert payload"

sleep(10)
exit(0)
```

C.17 FaucetEnv/Faucet/src/utils.jl

```
using StaticArrays
import Base: string
using CRC

# Convert Unsigned integer to vector of UInt8
to_bytes(x::UInt8)::SVector{1, UInt8} = [x]
to_bytes(x::UInt16)::SVector{2, UInt8} =
    reverse(unsafe_load(Ptr{SVector{2,
    UInt8}}(Base.unsafe_convert(Ptr{UInt16}, Ref(x)))))
to_bytes(x::UInt32)::SVector{4, UInt8} =
    reverse(unsafe_load(Ptr{SVector{4,
    UInt8}}(Base.unsafe_convert(Ptr{UInt32}, Ref(x)))))
to_bytes(x::UInt64)::SVector{8, UInt8} =
    reverse(unsafe_load(Ptr{SVector{8,
    UInt8}}(Base.unsafe_convert(Ptr{UInt64}, Ref(x)))))

struct IPv4Addr
```

```
    host::UInt32
    function IPv4Addr(host::UInt32)
        return new(host)
    end
end
# For pretty printing
string(ip::IPv4Addr)::String =
   join(Int64.(reverse(to_bytes(ip.host))), ".")

IPv4Addr(host::Vector{UInt8})::IPv4Addr =
   IPv4Addr(unsafe_load(Ptr{UInt32}(Base.unsafe_convert(Ptr{Vector{UInt8}},
   reverse(host)))))
IPv4Addr(host::SVector{4, UInt8})::IPv4Addr =
   IPv4Addr(Vector{UInt8}(host))
IPv4Addr(host::AbstractString)::IPv4Addr = IPv4Addr(SVector{4,
   UInt8}(reverse(parse.(UInt8, split(host, "."), base=10))))

# Convert but keep byte order
_to_bytes(x::UInt8)::SVector{1, UInt8} = [x]
_to_bytes(x::UInt16)::SVector{2, UInt8} =
   unsafe_load(Ptr{SVector{2,
   UInt8}}(Base.unsafe_convert(Ptr{UInt16}, Ref(x))))
_to_bytes(x::UInt32)::SVector{4, UInt8} =
   unsafe_load(Ptr{SVector{4,
   UInt8}}(Base.unsafe_convert(Ptr{UInt32}, Ref(x))))
_to_bytes(x::UInt64)::SVector{8, UInt8} =
   unsafe_load(Ptr{SVector{8,
   UInt8}}(Base.unsafe_convert(Ptr{UInt64}, Ref(x))))

# Convert to network byte order
to_net(in::Unsigned)::Vector{UInt8} = _to_bytes(hton(in))
to_net(in::IPv4Addr)::Vector{UInt8} = _to_bytes(hton(in.host))
to_net(in::Vector{UInt8})::Vector{UInt8} = reverse(in)
to_net(in::SVector{4, UInt8})::Vector{UInt8} = reverse(in)

# Parse a mac string into a UInt8 tuple
mac(s::AbstractString)::NTuple{6, UInt8} =
   tuple(map(x->parse(UInt8, x, base=16), split(String(s), ':'))...)


@enum Transport_Type begin
    TCP = 0x6
    UDP = 0x11
end

@enum Network_Type begin
    IPv4 = 0x0800
    ARP  = 0x0806
end

```

```
@enum Link_Type begin
    Ethernet
end

@enum Layer_type begin
    physical = 1    # Ethernet (on the wire only)
    link = 2        # Ethernet
    network = 3     # IPv4
    transport = 4   # TCP
    application = 5 # HTTP
end

custom_crc8_poly(poly::UInt8) = CRC.spec(8, poly, poly, false, false, 0x00, 0xf4) # Mimic the CRC_8 spec, with a custom polynomial

"""
    integrity_check(chunk::bitstring)::UInt8

CRC8 of the chunk, padded to 8 bits

```markdown
    Integrity: Known by both hosts
    Known_host: Known by challenger

    offset = integrity ⊻ known_host

    Challenger sends offset, responder returns `offset ⊻ integrity`
```
This implementation is CRC8 based, but just has to be deterministic.
"""
function integrity_check(chunk::String)::UInt8
    padding = 8 - (length(chunk) % 8)
    # Payload may not be byte aligned, so pad it
    chunk *= padding == 8 ? "" : "0"^padding
    return crc(CRC_8)([parse(UInt8, chunk[i:i+7], base=2) for i in 1:8:length(chunk)])
end

"""
    remove_padding(payload::bitstring, method::Symbol=PADDING_METHOD)::bitstring

    Removes the padding applied by [`pad_transmission`](@ref) from the payload.
"""
function remove_padding(payload::String, method::Symbol=PADDING_METHOD)::String
    if method == :short
        return rstrip(payload, '0')[1:end-1]
```

```
    elseif method == :covert
        for i = length(payload):-1:1
            if i % 128 == 0
                bitlen = lstrip(bitstring(Int64(i/128)), '0')
                if length(payload) - i >= length(bitlen)
                    if payload[i+1:i+length(bitlen)] == bitlen
                        payload = payload[1:i]
                        return payload
                    end
                end
            end
        end
        error("Incorrect padding")
    else
        error("Padding method $method not supported")
    end
end

# Regular expressions to parse these command's outputs
const ip_address_regex = r"^(?<index>\d+):
    (?<iface>[\w\d]+)(?:@[\w\d]+)?: <(?<ifType>[A-Z,_]+)> mtu
    (?<mtu>\d+) [\w ]+ state (?<state>[A-Z]+) group default qlen
    (?<qlen>\d+)[\s ]+link\/ether
    (?<mac>(?:[a-f\d]{2}:){5}[a-f\d]{2}) brd (?<brd>[a-f\d:]{17})
    [\w\- ]+[\s ]+inet
    (?<ip>(?:\d{1,3}.){3}\d{1,3})\/(?<subnet>\d+)"m
const ip_route_regex = r"^(?<dest_ip>(?:\d{1,3}.){3}\d{1,3}) (?:via
    (?<gw>(?:\d{1,3}.){3}\d{1,3}) )?dev (?<iface>[\w\d]+) src
    (?<src_ip>(?:\d{1,3}.){3}\d{1,3})"m
const ip_neigh_regex = r"^(?<ip>(?:\d{1,3}.){3}\d{1,3}) dev
    (?<iface>[\w\d]+) lladdr (?<mac>(?:[0-9a-f]{2}:){5}[0-9a-f]{2})"m
const ip_from_dev_regex = r"inet
    (?<ip>(?:\d{1,3}\.){3}\d{1,3})\/(?<cidr>\d{1,2})"
const mac_from_dev_regex = r"link\/(?<type>[a-z]+)
    (?<mac>[a-f\d:]{17})"

# Get the interface for the given ip
function get_ip_from_dev(dev::String)::String
    output = readchomp(`ip a show dev $dev`)
    return match(ip_from_dev_regex, output)[:ip]
end
```

C.18 FaucetEnv/Faucet/src/environment/headers.jl

```
import Base: -, ntoh

#=
```

```

    Headers.jl

=#

# C-Related constants
const PCAP_ERRBUF_SIZE  = 256
const ETHERTYPE_IP      = 0x0800
const IPPROTO_TCP       = 0x06
const IPPROTO_UDP       = 0x11
const ETHERTYPE_ARP     = 0x0806

const AF_PACKET         = Int32(17)     # Capture entire packet
const ETH_P_ALL         = Int32(0x0300) # Capture all ethernet frames (Ingoing & Outgoing)
const SOCK_RAW          = Int32(3)      # Raw socket (For listening)
const IPPROTO_RAW       = Int32(255)    # Raw IP packet (For sending)
const ETH_ALEN          = Cuchar(6) # Length of MAC address
const ETH_P_IP          = 0x0800  # IP protocol
const ARPHRD_ETHER      = Cushort(1) # Ethernet hardware format

#=

Generic types

=#

abstract type Header end

# TODO: Don't have this as mutable, it is slower than a generic struct
#        but not just making a new struct each time thats also slow...
mutable struct Layer{T<:Header}
    layer::Layer_type
    header::T
    payload::Union{Layer, Vector{UInt8}, Missing}
end

mutable struct Pcap end

struct Timeval
    seconds::Clong
    ms::Clong
end

function -(t1::Timeval, t2::Timeval)::Float64
    return (t1.seconds - t2.seconds) + ((t1.ms - t2.ms) / 1_000_000)
end
```

```

struct Capture_header
    timestamp::Timeval
    capture_length::Int32
    length::Int32
end

#=

    Protocol header definitions

    _protocol are headers as defined in C,
     they are processed into a more verbose version for convinience

=#

struct Ethernet_header <: Header
    destination::NTuple{6, UInt8}
    source::NTuple{6, UInt8}
    protocol::UInt16
    function Ethernet_header(p::Ptr{UInt8})::Ethernet_header
        return unsafe_load(Ptr{Ethernet_header}(p))
    end
end

getoffset(::Ethernet_header)::Int64         = sizeof(Ethernet_header)
getprotocol(hdr::Ethernet_header)::UInt16   = ntoh(hdr.protocol)

struct Packet
    cap_header::Capture_header
    payload::Layer{Ethernet_header}
end

struct ARP_header <: Header
    hardware_type::UInt16
    protocol_type::UInt16
    hardware_length::UInt8
    protocol_length::UInt8
    opcode::UInt16
    sender_mac::NTuple{6, UInt8}
    sender_ip::NTuple{4, UInt8}
    target_mac::NTuple{6, UInt8}
    target_ip::NTuple{4, UInt8}
    function ARP_header(p::Ptr{UInt8})::ARP_header
        return unsafe_load(Ptr{ARP_header}(p))
    end
end

getoffset(::ARP_header)::Int64 = sizeof(ARP_header)
```

```
getprotocol(::ARP_header)::UInt8 = 0x0

# First read base header, then we can deal with options etc.
struct _IPv4_header
    version_ihl::UInt8 # 4 -> version, 4 -> IHL
    tos::UInt8 # Type of service + priority
    tot_len::UInt16 # Total IPv4 length
    id::UInt16 # identification
    frag_off::UInt16 # Fragmented offset
    ttl::UInt8
    protocol::UInt8
    check::UInt16 # Header checksum
    saddr::UInt32
    daddr::UInt32
    function _IPv4_header(p::Ptr{UInt8})::_IPv4_header
        return unsafe_load(Ptr{_IPv4_header}(p))
    end
end

Base.ntoh(x::UInt8)::UInt8 = x
byte_to_nibbles(x::UInt8)::NTuple{2, UInt8} = (x & 0x0f, (x & 0xf0) >> 4)

struct IPv4_header <: Header
    version::UInt8
    ihl::UInt8
    tos::UInt8
    tot_len::UInt16
    id::UInt16
    frag_off::UInt16
    ttl::UInt8
    protocol::UInt8
    check::UInt16
    saddr::UInt32
    daddr::UInt32
    options::UInt32
    function IPv4_header(p::Ptr{UInt8})::IPv4_header
        _ip = _IPv4_header(p)
        version, ihl = byte_to_nibbles(ntoh(_ip.version_ihl)) # Version + IHL
        options_offset = p + sizeof(_IPv4_header)
        options_size = ihl*4 - sizeof(_IPv4_header)
        options = options_size > 0 ? Base.pointerref(options_offset, 1, options_size) : 0
        return new(
            ihl, version, # These two may be flipped depending on byte order
            ntoh(_ip.tos), ntoh(_ip.tot_len), ntoh(_ip.id),
            ntoh(_ip.frag_off), ntoh(_ip.ttl), ntoh(_ip.protocol),
            ntoh(_ip.check), ntoh(_ip.saddr), ntoh(_ip.daddr),
```

```
                ntoh(options)
            )
    end
end

getoffset(hdr::IPv4_header)::Int64      = hdr.ihl * 4
getprotocol(hdr::IPv4_header)::UInt8    = hdr.protocol

struct _TCP_header
    sport::UInt16
    dport::UInt16
    seq::UInt32
    ack_num::UInt32
    flags::UInt16
    win_size::UInt16
    check::UInt16
    urg_ptr::UInt16
    function _TCP_header(p::Ptr{UInt8})::_TCP_header
        return unsafe_load(Ptr{_TCP_header}(p))
    end
end

struct TCP_header <: Header
    sport::UInt16
    dport::UInt16
    seq::UInt32
    ack_num::UInt32
    hdr_len::UInt8
    reserved::UInt8
    urg::Bool
    ack::Bool
    psh::Bool
    rst::Bool
    syn::Bool
    fin::Bool
    win_size::UInt16
    check::UInt16
    urg_ptr::UInt16
    #options::NTuple{10, UInt32}
    function TCP_header(p::Ptr{UInt8})::TCP_header
        _tcp = _TCP_header(p)
        flags = ntoh(_tcp.flags)
        header_length   =  (flags & 0b1111000000000000) >> 12
        reserved    = UInt8((flags & 0b0000111111000000) >> 6)
        urg             = ((flags & 0b0000000000100000) >> 5) == 0x1
        ack             = ((flags & 0b0000000000010000) >> 4) == 0x1
        psh             = ((flags & 0b0000000000001000) >> 3) == 0x1
        rst             = ((flags & 0b0000000000000100) >> 2) == 0x1
        syn             = ((flags & 0b0000000000000010) >> 1) == 0x1
        fin             = ((flags & 0b0000000000000001)     ) == 0x1
```

```julia
        #options_offset = p + sizeof(_TCP_header)
        #options_size = header_length*4 - sizeof(_TCP_header)
        #options = options_size > 0 ? Base.pointerref(options_offset, 1, options_size) : zeros(UInt32, 10)
        # Go from network byte order to host byte order (excluding flags as we already changed them)
        return new(
            ntoh(_tcp.sport), ntoh(_tcp.dport), ntoh(_tcp.seq), ntoh(_tcp.ack_num),
            header_length, reserved, urg, ack, psh, rst, syn, fin,
            ntoh(_tcp.win_size), ntoh(_tcp.check), ntoh(_tcp.urg_ptr)#, options
        )
    end
end

getoffset(hdr::TCP_header)::Int64           = hdr.hdr_len * 4
getprotocol(lyr::Layer{TCP_header})::UInt64 = lyr.payload[1:min(end, 4)] # No protocol field...

struct UDP_header <: Header
    sport::UInt16
    dport::UInt16
    len::UInt16
    check::UInt16
    function UDP_header(p::Ptr{UInt8})::UDP_header
        return unsafe_load(Ptr{UDP_header}(p))
    end
end

getoffset(::UDP_header)::Int64                    = sizeof(UDP_header)
getprotocol(lyr::Layer{UDP_header})::Unsigned     = lyr.payload[1:min(length(lyr.payload), 4)]

# Default layer -> header call
getoffset(lyr::Layer{<:Header})::Int64            = getoffset(lyr.header)
getprotocol(lyr::Layer{<:Header})::Unsigned       = getprotocol(lyr.header)

#=

    Header tree

=#

struct Node
    type::Type{<:Header}
    id::Unsigned
```

```
    children::Vector{Node}
end

# Transport - 4

HEADER_UDP = Node(
    UDP_header,
    IPPROTO_UDP,
    []
)

HEADER_TCP = Node(
    TCP_header,
    IPPROTO_TCP,
    []
)

# Network - 3

HEADER_IPv4 = Node(
    IPv4_header,
    ETHERTYPE_IP,
    [HEADER_TCP, HEADER_UDP]
)

HEADER_ARP = Node(
    ARP_header,
    ETHERTYPE_ARP,
    []
)

# Link - 2

HEADER_Ethernet = Node(
    Ethernet_header,
    0x00,
    [HEADER_IPv4, HEADER_ARP]
)
```

C.19 FaucetEnv/Faucet/src/environment/queue.jl

```
#=

    queue.jl

=#

```

```
"""
    errbuff_to_error(errbuf::Vector{UInt8})

Raise error with null terminated string, often returned by libpcap
"""
function errbuff_to_error(errbuf::Vector{UInt8})
    # Raise error with null terminated string
    error(String(errbuf[1:findfirst(==(0), errbuf) - 1]))
end

"""
    pcap_lookupdev()::String

Get the default device
!!! Is a deprecated function, but is the simplest way to get the
   device
"""
function pcap_lookupdev()::String
    # Error is returned into errbuff
    errbuff = Vector{UInt8}(undef, PCAP_ERRBUF_SIZE)
    # Get device
    device = ccall(
        (:pcap_lookupdev, "libpcap"),
        Cstring,
        (Ptr{UInt8},),
        errbuff
    )
    if device == C_NULL
        errbuff_to_error(errbuff)
    end
    dev = unsafe_string(device)
    @debug "pcap_lookupdev() returned '$dev'"
    return dev
end

function get_dev()::String
    if length(ARGS) ≥ 2
        @debug "Using device from command line" dev=ARGS[2]
        return ARGS[2]
    else
        return pcap_lookupdev()
    end
end

"""
    pcap_open_live(device::String, snapshot_len::Int64,
   promisc::Bool)::Ptr{Pcap}

Open a live pcap session, returning a handle to the session
"""
```

```
function pcap_open_live(device::String, snapshot_len::Int64,
   promisc::Bool)::Ptr{Pcap}
    # Error is returned into errbuff
    errbuff = Vector{UInt8}(undef, PCAP_ERRBUF_SIZE)
    # 1 is promisc, 0 isn't - so convert from Bool
    promisc = Int64(promisc)
    to_ms = 1000
    handle = ccall(
        (:pcap_open_live, "libpcap"),
        Ptr{Pcap},
        (Cstring, Int32, Int32, Int32, Ptr{UInt8}),
        device, snapshot_len, promisc, to_ms, errbuff
    )
    if handle == C_NULL
        errbuff_to_error(errbuff)
    end
    return handle
end

"""
    pcap_loop(p::Ptr{Pcap}, cnt::Int64, callback::Function,
   user::Union{Ptr{Nothing}, Ptr{UInt8}})

Loop through packets, calling callback on each packet
"""
function pcap_loop(p::Ptr{Pcap}, cnt::Int64, callback::Function,
   user::Union{Ptr{Nothing}, Ptr{UInt8}})::Nothing
    !hasmethod(callback, Tuple{Ptr{UInt8}, Ptr{Capture_header},
   Ptr{UInt8}}) ? error("Invalid callback parameters") :
    cfunc = @cfunction($callback, Cvoid, (Ptr{UInt8},
   Ptr{Capture_header}, Ptr{UInt8}))
    ccall(
        (:pcap_loop, "libpcap"),
        Int32,
        (Ptr{Pcap}, Int32, Ptr{Cvoid}, Ptr{Cuchar}),
        p, cnt, cfunc, user
    )
end

"""
    pcap_breakloop(pcap::Ptr{Pcap})::Nothing

Break out of pcap_loop
"""
function pcap_breakloop(pcap::Ptr{Pcap})::Nothing
    ccall((:pcap_breakloop, "libpcap"), Cvoid, (Ptr{Pcap},), pcap)
end

"""
```

```
    packet_from_pointer(p::Ptr{UInt8}, packet_size::Int32)::Layer{Ethernet_header}

Convert a pointer to a packet into a Layer{Ethernet_header} object, with all headers and payload
"""
function packet_from_pointer(p::Ptr{UInt8}, packet_size::Int32)::Layer{Ethernet_header}
    layer_index = 2
    offset = 0
    # Push layers here as we make them, then we can walk backwards and craft a full payload
    layers::Vector{Layer{<:Header}} = [Layer(Layer_type(layer_index), Ethernet_header(p), missing)]
    # Start at the lowest layer (Ethernet)
    node = HEADER_Ethernet
    # Don't exceed layer 3 (we +1 to make it 4, Transport, and we don't have any application layer protocols defined)
    # Would be an improvement to work this out by finding the depth of our HEADER_Ethernet tree...
    while layer_index ≤ 3
        prev_layer = layers[end]::Layer{<:Header}
        offset += getoffset(prev_layer)
        proto = getprotocol(prev_layer)
        # println("Packet context:", prev_layer, "\noffset:", offset, "\nproto:", proto)
        for child ∈ node.children
            if child.id == proto
                # println("Added child:", child)
                layer_index += 1
                node = child
                push!(layers, Layer(Layer_type(layer_index), child.type(p+offset), missing))
                # There is only 1 header per layer, so skip the search
                break
            end
        end
        if prev_layer.layer == Layer_type(layer_index)
            # End of tree
            break
        end
    end
    l = layers[end]::Layer{<:Header}
    offset += getoffset(l)

    # If there is a payload, read it into a vector
    payload_size = packet_size - offset
    if payload_size > 0
        payload = zeros(UInt8, payload_size)
```

```
        for i=1:payload_size
            payload[i] = Base.pointerref(p+offset+i, 1, 1)
        end
    else
        payload = Vector{UInt8}()
    end

    # Set the payload to the payload of the lowest layer
    l.payload = payload
    # Decrement layer index to walk backwards
    layer_index -= 1
    # Walk backwards through the layers, setting the payload of
    each layer to the layer above it, stopping at the link-layer
    (ETHERNET)
    while layer_index ≥ 2
        layer_index -= 1
        packet = layers[layer_index]::Layer{<:Header}
        packet.payload = l
        l = packet
    end
    # Return the lowest layer (ETHERNET)
    return l
end

"""
    get_callback(queue::CircularChannel{Packet})::Function

Get a callback function for pcap_loop, which will push packets to
    the queue
"""
function get_callback(queue)::Function
    function callback(::Ptr{UInt8}, header::Ptr{Capture_header},
    packet::Ptr{UInt8})::Cvoid
        cap_hdr = unsafe_load(header)
        pkt = Packet(cap_hdr, packet_from_pointer(packet,
    cap_hdr.capture_length))
        put!(queue, pkt)
        return nothing
    end
    return callback
end

"""
Get the local IP address of a device, if one is not given, assume
    default device
"""
function get_local_ip(device::String)::String
    match = get_ip_from_dev(device)
    if isnothing(match)
        error("Could not find IP address for device: ", device)
```

```
    end
    return match
end
get_local_ip() = get_local_ip(get_dev())

"""
    init_queue(device::String,
   bpf_filter_string::String="")::CircularChannel{Packet}

Given the device to open the queue on, return a
   CircularChannel{Packet} which will be filled with packets
"""
function init_queue(device::String=get_dev();
   bpf_filter_string::String="")::CircularChannel{Packet}
    queue = CircularChannel{Packet}(ENVIRONMENT_QUEUE_SIZE)
    handle = pcap_open_live(device, -1, true)
    # Set the filter if one is supplied
    if bpf_filter_string != ""
        program = Ref{bpf_prog}()
        pcap_compile(handle, program, bpf_filter_string, Int32(1),
   UInt32(0))
        # Add filter to pcap handle
        pcap_setfilter(handle, program)
        pcap_freecode(program)
    end
    close_pcap() = pcap_breakloop(handle)
    # Add a hook to close the pcap on exit, so the program exits
   cleanly
    atexit(close_pcap)
    # Create a fuction with the queue bound to it, so we don't have
   to deal with passing it every time
    callback = get_callback(queue)
    @debug "Creating pcap sniffer" device=device
    # Run the listener in a seperate thread
    #  we use errormonitor here so errors on this thread are sent
   to main thread
    errormonitor(Base.Threads.@spawn pcap_loop(handle, -1,
   callback, C_NULL))
    return queue
end
```

C.20 FaucetEnv/Faucet/src/environment/query.jl

```
#=

    Query.jl

```

```
    Functions related to extracting information from the
  environment queue:
        - Custom querying
        - Extracting headers
        - Environment statistics
        - get_tcp_server

=#

# Common tcp services (80, 443, etc.)
const TCP_SERVICES = [UInt16(80), UInt16(443), UInt16(20),
  UInt16(21), UInt16(22), UInt16(25), UInt16(143)]

"""
    get_headers(p::Packet)::Vector{Header}

Seperate the headers from a packet into a vector and return it
"""
function get_headers(p::Packet)::Vector{Header}
    headers = Vector{Header}()
    layer = p.payload
    while true
        push!(headers, layer.header)
        layer = layer.payload
        if !isa(layer, Layer)
            return headers
        end
    end
end

"""
    get_header(p::Packet, l::Layer_type)::Union{Header, Missing}

Get the header of a specific layer from a packet
"""
function get_header(p::Packet, l::Layer_type)::Union{Header,
  Missing}
    headers = get_headers(p)
    for i=1:lastindex(headers)
        if Layer_type(i+1) == l
            return headers[i]
        end
    end
    return missing
end

"""
    get_queue_data(q::CircularChannel{Packet})::Vector{Packet}

Converts the queue to a static vector of packets (Implicitly)
```

```
"""
get_queue_data(q::CircularChannel{Packet})::Vector{Packet} = q

"""
    get_layer_stats(packets, layer)::Dict{String, Int64}

Get the statistics of a layer in the queue

```
Returns a dictionary with the following format:
    - Key: The name of the header
    - Value: The number of packets with that header
```

"""
function get_layer_stats(v::Vector{Packet},
   l::Layer_type)::Dict{String, Int64}
    packet_headers = [get_header(p, l) for p ∈ v]
    info = Dict{String, Int64}()
    info["Missing"] = 0
    for h ∈ packet_headers
        if ismissing(h)
            info["Missing"] += 1
        else
            prot = split(string(typeof(h)), ".")[end]
            if prot ∈ keys(info)
                info[prot] += 1
            else
                info[prot] = 1
            end
        end
    end
    if info["Missing"] == 0
        delete!(info, "Missing")
    end
    return info
end

"""
    get_layer_stats(q::CircularChannel{Packet},
   l::Layer_type)::Dict{String, Int64}

Converts the queue to a static vector of packets pre-processing
"""
get_layer_stats(q::CircularChannel{Packet},
   l::Layer_type)::Dict{String, Int64} =
   get_layer_stats(get_queue_data(q), l)

"""
    get_layer2index(tree_root::Node)::Dict{String, Int64}
```

```

Create a dictionary that maps the layer name to its index in the tree, using the packet tree defined in `headers.jl`
"""
function get_layer2index(tree_root::Node)::Dict{String, Int64}
    layer2index = Dict{String, Int64}()
    current_layer_nodes = Vector{Node}([tree_root])
    next_layer_nodes = Vector{Node}()
    current_layer = 1
    while length(current_layer_nodes) > 0
        for node ∈ current_layer_nodes
            layer2index[split(string(node.type), ".")[end]] = current_layer
            for child ∈ node.children
                push!(next_layer_nodes, child)
            end
        end
        current_layer_nodes = next_layer_nodes
        next_layer_nodes = Vector{Node}()
        current_layer += 1
    end
    return layer2index
end

layer2index = get_layer2index(HEADER_Ethernet)

"""
Returns true if the arguments match the header

# Example
```markdown
In a packet with the following headers:
    - TCP Packet with source port 80

The following query will return true:
    - `query_header(header, Dict(:sport => 80))`
```

"""
function query_header(header::Header, arguments::Dict{Symbol, Any})::Bool
    # Returns true if the header matches the arguments
    if isempty(arguments)
        return true
    end
    o = Vector{Bool}()
    for (k, v) ∈ arguments
        data = getfield(header, k)
        if typeof(v) == Vector{typeof(data)}
            push!(o, data ∈ v)
```

```
        else
            push!(o, data == v)
        end
    end
    return all(o)
end
query_header(::Header, ::Nothing)::Bool = true
query_header(::Nothing, ::Dict{Symbol, Any})::Bool = false


"""
    query_headers(headers, arguments)::Bool

Returns true if arguments match the their respective headers

# Example
```markdown
Packet with the following headers:
    - Ethernet Packet with source MAC 00:00:00:00:00:00
    - TCP Packet with source port 80

The following query will return true:
    - `query_headers(headers, Dict(
        "Ethernet_header" => Dict(:smac => 0x000000000000),
        "TCP_header" => Dict(:sport => 80)
        ))`
```
"""
function query_headers(headers::Vector{Header}, arguments::Dict{String, Dict{Symbol, Any}})::Bool
    # Returns true if the headers match the arguments
    o = Vector{Bool}()
    for (k, v) ∈ arguments
        header = get(headers, layer2index[k], nothing)
        if !isnothing(header) && split(string(typeof(header)), ".")[end] == k
            push!(o, query_header(header, v))
        else
            push!(o, false)
        end
    end
    return all(o)
end

"""
    query_queue(queue, arguments)::Vector{Vector{Header}}

Returns a vector of all packet headers if a packet matches an argument

```

```
Arguments is a vector of dictionaries, each dictionary is a match case
A match case is a dictionary of header types and required values in that header

Example
```julia
arguments = [
    {
        TCP_header => {
            :sport => 0x2f11 # 12049
            :dport => 0x0050 # 80
        }
    },
    {
        IPv4_header => {
            :ihl => [0x05, 0x06]
        }
    }
]
```
!!! note
    will return all headers for packets that either have:
    (tcp header with a source port of `12049` *AND* destination port of `80`) *OR*
    (ipv4 header with an ihl of `5` *OR* `6`)

"""
function query_queue(q::Vector{Packet}, arguments::Vector{Dict{String, Dict{Symbol, Any}}})::Vector{Vector{Header}}
    query = Vector{Vector{Header}}()
    for p ∈ q
        headers = get_headers(p)
        if any([query_headers(headers, case) for case ∈ arguments])
            push!(query, headers)
        end
    end
    return query
end
query_queue(q::CircularChannel{Packet}, args::Vector{Dict{String, Dict{Symbol, Any}}})::Vector{Vector{Header}} = query_queue(get_queue_data(q), args)

"""
    get_tcp_server(queue)::(MAC, IP, Port)

Returns the MAC address, IP address, and port of the most recently active TCP server, using common TCP service ports and SYN packets
Returns nothing if no server can be found
```

```
"""
function get_tcp_server(q::Vector{Packet})::Union{Tuple{NTuple{6, UInt8}, UInt32, UInt16}, NTuple{3, Nothing}}
    common_query = Vector{Dict{String, Dict{Symbol, Any}}}([
        Dict{String, Dict{Symbol, Any}}(
            "TCP_header" => Dict{Symbol, Any}(
                :dport => TCP_SERVICES
            )
        )
    ])
    # Clients connected to common TCP Services, TCP traffic to them is favourable (In terms of covertness)
    services = query_queue(q, common_query)

    syn_query = Vector{Dict{String, Dict{Symbol, Any}}}([
        Dict{String, Dict{Symbol, Any}}(
            "TCP_header" => Dict{Symbol, Any}(
                :syn => true
            )
        )
    ])

    service_mac, service_ip, service_port = nothing, nothing, nothing
    if !isempty(services)
        # Take the most recently active one
        service = pop!(services)
        # This is a packet going toward tcp service
        service_mac = getfield(service[layer2index["Ethernet_header"]], :source)
        # Ethernet_header -> tcp server mac (of next hop from local perspective)
        service_ip = getfield(service[layer2index["IPv4_header"]], :daddr)
        # IP_header.daddr -> tcp server ip
        service_port = getfield(service[layer2index["TCP_header"]], :dport)
        # TCP_header.dport -> tcp server port
    else
        syn_traffic = query_queue(q, syn_query)
        if !isempty(syn_traffic)
            # Take most recently active one
            service = pop!(syn_traffic)
            # Again, a packet going toward a tcp server
            service_mac = getfield(service[layer2index["Ethernet_header"]], :source)
            # Ethernet_header -> tcp server mac (of next hop from local perspective)
            service_ip = getfield(service[layer2index["IPv4_header"]], :daddr)
```

```
            # IP_header.daddr -> tcp server ip
            service_port = getfield(service[layer2index["TCP_header"]], :dport)
            # TCP_header.dport -> tcp server port
        end
    end
    # Currently, the test environment does not support getting a valid tcp server like this (to be fixed...)
    @debug "Found TCP Server, but using hardcoded (due to test environment)" service_ip service_mac service_port
    ip = 0xc91e140a # 10.20.30.201
    mac_raw = match(r"^Unicast reply from (?:\d{1,3}\.){3}\d{1,3} \[(?<mac>(?:[A-F\d]{2}:){5}[A-F\d]{2})\]"m, readchomp(`arping -c 1 10.20.30.201`))[:mac]
    mac = tuple(map(x->parse(UInt8, x, base=16), split(String(mac_raw), ':'))...)
    #mac = (0xfa, 0x4c, 0x92, 0x7f, 0x95, 0x3b)
    port = 0x0050
    return (mac, ip, port)
end
get_tcp_server(q::CircularChannel{Packet})::Union{Tuple{NTuple{6, UInt8}, UInt32, UInt16}, NTuple{3, Nothing}} = get_tcp_server(get_queue_data(q))

get_local_host_count(q::CircularChannel{Packet}, local_address::IPv4Addr, subnet_mask::Int=24)::Int64 = get_local_host_count(get_queue_data(q), local_address, subnet_mask)
function get_local_host_count(q::Vector{Packet}, local_address::IPv4Addr, subnet_mask::Int=24)::Int64
    # CIDR notation is number of bits that denote the network, the rest are host bits
    local_address_mask = typemax(UInt32) << (32 - subnet_mask)
    # We want hosts on the same subnet so ignore the host bits
    local_address = local_address_mask & hton(local_address.host)
    # Get all IPv4 headers
    ipv4_headers = [h[2] for h ∈ query_queue(q, [
        Dict{String, Dict{Symbol, Any}}(
            "IPv4_header" => Dict{Symbol, Any}()
        )
    ])]
    hosts = Set{UInt32}()
    for h ∈ ipv4_headers
        for addr ∈ (h.saddr, h.daddr)
            if (addr & local_address_mask) == local_address
                push!(hosts, addr)
            end
        end
    end
    return length(hosts)
```

```julia
end

"""
    get_local_host(queue, local_address, subnet_mask)::Vector{UInt8}

    Returns the host byte of the local ip address
"""
function get_local_net_host(q::Vector{Packet}, local_address::IPv4Addr, blacklist::Vector{UInt8}=[], subnet_mask::Int=24)::UInt8 # return the host byte of the local ip
    # Do not use .0 or .255 (network and broadcast addresses)
    # Sometimes they will not be, but this is a simpler and "good enough" solution
    push!(blacklist, 0x00, 0xff)
    # Get all IPv4 headers
    ipv4_headers = [h[2] for h ∈ query_queue(q, [
        Dict{String, Dict{Symbol, Any}}(
            "IPv4_header" => Dict{Symbol, Any}()
        )
    ])]
    # CIDR notation is number of bits that denote the network, the rest are host bits
    local_address_mask = typemax(UInt32) << (32 - subnet_mask)
    # We want hosts on the same subnet so ignore the host bits
    local_address = local_address_mask & hton(local_address.host)
    hosts = Dict{UInt8, Int}()
    for ipv4 ∈ ipv4_headers
        for header ∈ (ipv4.saddr, ipv4.daddr)
            if header & local_address_mask == local_address
                # Invert the subnet mask to make it a host mask
                host_byte = UInt8(header & ~local_address_mask)
                if haskey(hosts, host_byte)
                    # Increment the number of times we have seen this host byte
                    hosts[host_byte] += 1
                else
                    hosts[host_byte] = 1
                end
            end
        end
    end
    # Remove blacklisted host bytes from our list
    for b ∈ blacklist
        delete!(hosts, b)
    end
    # Return the most active local host
    return collect(keys(hosts))[findmax(collect(values(hosts)))[2]]
end
```

```
get_local_net_host(q::CircularChannel{Packet},
    local_address::IPv4Addr, blacklist::Vector{UInt8}=[],
    subnet_mask::Int=24)::UInt8 =
    get_local_net_host(get_queue_data(q), local_address, blacklist,
    subnet_mask)
```

C.21 FaucetEnv/Faucet/src/environment/env_utils.jl

```
"""
    get_socket(domain, type, protocol)::IOStream
Return a raw socket, wrapped into an `IOStream`
"""
function get_socket(domain::Cint, type::Cint,
    protocol::Cint)::IOStream
    fd = ccall(:socket, Cint, (Cint, Cint, Cint), domain, type,
    protocol)
    if fd == -1
        @error "Failed to open socket" errno=Base.Libc.errno()
    end
    return fdio(fd)
end

"""
Link-layer socket address structure
    required for sending packets at the link-layer
    indicates the interface to send the packet to.
"""
struct Sockaddr_ll
    sll_family::Cushort
    sll_protocol::Cushort
    sll_ifindex::Cint
    sll_hatype::Cushort
    sll_pkttype::Cuchar
    sll_halen::Cuchar
    sll_addr::NTuple{6, Cuchar}
    function Sockaddr_ll(;
        sll_family::Cushort=hton(UInt16(AF_PACKET)),
        sll_protocol::Cushort=hton(ETH_P_IP),
        sll_ifindex::Cint,
        sll_hatype::Cushort=hton(ARPHRD_ETHER),
        sll_pkttype::Cuchar=Cuchar(0),
        sll_halen::Cuchar=ETH_ALEN,
        sll_addr::NTuple{6, Cuchar}
    )
        new(sll_family, sll_protocol, sll_ifindex, sll_hatype,
    sll_pkttype, sll_halen, sll_addr)
    end
```

```
end

"""
    sendto(sockfd, packet, interface_id)::Cint

Send a packet to the interface with the given id.
"""
sendto(sock::IOStream, packet::Vector{UInt8},
   interface_name::Union{String, Cint})::Cint = sendto(fd(sock),
   packet, interface_name)
sendto(sockfd::Integer, packet::Vector{UInt8},
   interface_name::String)::Cint = sendto(sockfd, packet,
   if_nametoindex(interface_name))
function sendto(sockfd::Integer, packet::Vector{UInt8},
   interface_id::Integer)::Cint
    sockfd = sockfd == Cint(sockfd) ? Cint(sockfd) : error("sockfd
   size is unsupported")
    interface_id = interface_id == Cint(interface_id) ?
   Cint(interface_id) : error("interface_id size is unsupported")
    # Dest addr in packet at know offset
    destination_addr = NTuple{6, Cuchar}(packet[7:12])
    # Create sockaddr_ll
    sockaddr_ll = Sockaddr_ll(sll_ifindex=interface_id,
   sll_addr=destination_addr)
    # Send packet
    bytes = ccall(:sendto, Cint, (Cint, Ptr{UInt8}, Csize_t, Cint,
   Ptr{Sockaddr_ll}, Cint), sockfd, packet, length(packet), 0,
   Ref(sockaddr_ll), sizeof(sockaddr_ll))
    if bytes == -1
        @error "Failed to send packet" errno=Base.Libc.errno()
    end
    return bytes
end

"""
Get the interface index for the given interface name.
see also: [`if_indextoname`](@ref)
"""
function if_nametoindex(name::String)::Cuint
    ifface_idx = ccall(:if_nametoindex, Cuint, (Cstring,), name)
    if ifface_idx == 0
        @error "Failed to get interface index"
   errno=Base.Libc.errno()
    end
    @assert if_indextoname(ifface_idx) != name "Mismatch in
   interface id and name, got '$(if_indextoname(ifface_idx))'
   instead of '$name' ($ifface_idx)"
    return ifface_idx
end

```

```
"""
Get the interface name for the given interface index.
see also: [`if_nametoindex`](@ref)
"""
function if_indextoname(index::Cuint)::String
    name = Vector{UInt8}(undef, 16)
    ret = ccall(:if_indextoname, Ptr{UInt8}, (Cuint, Ptr{UInt8}), index, name)
    if ret == C_NULL
        @error "Failed to get interface name" errno=Base.Libc.errno()
    end
    return String(name)
end

# Sequence of bytes for an ARP packet, starting from ethertype ending at at the opcode (0x0001 for request)
const ARP_SEQUENCE        = [0x08, 0x06, 0x00, 0x01, 0x08, 0x00, 0x06, 0x04, 0x00, 0x01]
# The slice the above sequence is found in
const ARP_SEQUENCE_SLICE  = 13:22
# The slice the source address is found in
const ARP_SRC_SLICE       = 29:32
# The slice the destination address is found in
const ARP_DEST_SLICE      = 39:42

"""
Await an arp beacon from the source address, return nothing if timeout is reached, otherwise return the data
"""
function await_arp_beacon(ip::IPv4Addr, target::UInt8, timeout::Int64=5)
    # Get a fresh socket to listen on
    socket = get_socket(AF_PACKET, SOCK_RAW, ETH_P_ALL)
    heard = Vector{UInt8}()
    @debug "Started listening @" time() timeout
    start = time_ns()
    while (time_ns() - start) < timeout * 1e9
        # Read a packet
        raw = read(socket)
        # Confirm it is more than the mininum size of an ARP packet
        if length(raw) >= 42
            # Check it matches our boilerplate ARP
            if raw[ARP_SEQUENCE_SLICE] == ARP_SEQUENCE
                # Check it is from the source we are looking for
                if raw[ARP_SRC_SLICE] == _to_bytes(ip.host)
                    # Check it is to the target we are looking for
                    if raw[ARP_DEST_SLICE][4] == target
                        return true
                    end
```

```
                    push!(heard, raw[ARP_DEST_SLICE][4])
                end
            end
        end
    end
    @warn "Timed out waiting for ARP beacon" heard_ips="10.20.30."
   .* string.(Int.(heard))
    return false
end
```

C.22 FaucetEnv/Faucet/src/environment/bpf.jl

```
# Berkley packet filter allows a userspace program to apply a
   filter to packets

#
   https://github.com/torvalds/linux/blob/master/samples/bpf/bpf_insn.h

struct bpf_insn
    code::Cushort
    jt::Cuchar
    jf::Cuchar
    k::Clong
end

struct bpf_prog
    len::Cuint
    insns::Ptr{bpf_insn}
end

function local_bound_traffic(local_ip::String)::String
    return "dst host $local_ip"
end
local_bound_traffic() = local_bound_traffic(get_local_ip())

function pcap_setfilter(p::Ptr{Pcap}, fp::Ref{bpf_prog})
    ccall((:pcap_setfilter, "libpcap"), Cint, (Ptr{Cvoid},
   Ref{bpf_prog}), p, fp)
end

function pcap_compile(p::Ptr{Pcap}, fp::Ref{bpf_prog}, str::String,
   optimize::Cint, netmask::Cuint)
    ccall((:pcap_compile, "libpcap"), Cint, (Ptr{Cvoid},
   Ref{bpf_prog}, Ptr{Cchar}, Cint, Cuint), p, fp, str, optimize,
   netmask)
end

```

```
function pcap_freecode(fp::Ref{bpf_prog})
    ccall((:pcap_freecode, "libpcap"), Cvoid, (Ref{bpf_prog},), fp)
end

# Move these functions to environment, and allow for a filter to be
   passed to init_queue

# Only allow packets from $local_ip

# p = pcap_open_live(...)
# program = Ref{bpf_prog}()
# pcap_compile(p, program, "src host $local_ip", 1, 0)
# pcap_setfilter(p, program)
# pcap_freecode(program)
# pcap_loop(...)
```

C.23 FaucetEnv/Faucet/src/target.jl

```
using StaticArrays

# This struct is the basis of our "Agreement"
struct Target
    # The IP of the target (receiver)
    ip::IPv4Addr
    # The methods to use
    covert_methods::Vector{String}
    # The AES PSK and IV
    AES_PSK::Vector{UInt8}
    AES_IV::Vector{UInt8}
    function Target(ip::IPv4Addr, covert_methods::Vector{String},
   AES_PSK::SVector{16, UInt8}, AES_IV::SVector{16, UInt8})
        return new(ip, covert_methods, Vector{UInt8}(AES_PSK),
   Vector{UInt8}(AES_IV))
    end
end
function Target(ip::AbstractString, covert_methods::Vector{String},
   AES_PSK::SVector{16, UInt8}, AES_IV::SVector{16, UInt8})
    return Target(IPv4Addr(ip), covert_methods, AES_PSK, AES_IV)
end

chunk(s::AbstractString, n::Int)::Vector{AbstractString} =
   [s[i:min(i + n - 1, end)] for i=1:n:length(s)]

"""
Parse target from file structure to Target struct
```
Target file structure:
```
```

````
ip: 'aaa.bbb.ccc.ddd'
covert_methods: ['method1', 'method2', ...]
AES_PSK: '000102030405060708090A0B0C0D0E0F'
AES_IV: '000102030405060708090A0B0C0D0E0F'
```
"""
function parse_target_file(file::AbstractString)::Target
    target = Dict{String, String}([attr => value for (attr, value) ∈ split.(split(readchomp(file), "\n"), ": ")])
    # Check all required keys are present, any others will be ignored
    if all(["ip", "covert_methods", "AES_PSK", "AES_IV"] .∉ Ref(keys(target)))
        @error "Invalid arguments" keys(target)
        throw(ArgumentError("Target file must contain ip, covert_methods, AES_PSK, and AES_IV"))
    end

    # extract the values from the text form, using the ' delimiter
    ip = split(target["ip"], "'")[2]
    covert_methods = String.(split(split(split(target["covert_methods"], "['")[2], "']")[1], "', '"))
    AES_PSK = SVector{16, UInt8}(parse.(UInt8, chunk(split(target["AES_PSK"], "'")[2], 2); base=16))
    AES_IV = SVector{16, UInt8}(parse.(UInt8, chunk(split(target["AES_IV"], "'")[2], 2); base=16))

    # Return the target
    return Target(ip, covert_methods, AES_PSK, AES_IV)
end

# Get the target file from the arguments
if length(ARGS) > 0
    target = parse_target_file(ARGS[1])
else
    error("No target file provided")
end

# Example target file
"""
ip: '10.20.30.2'
covert_methods: ['IPv4_identification', 'TCP_ACK_Bounce']
AES_PSK: '0F0E0D0C0B0A09080706050403020100'
AES_IV: '000102030405060708090A0B0C0D0E0F'
"""
````
````

C.24 FaucetEnv/Faucet/src/main.jl

```
#=

    Main

    Setup modules, including the files from inside the module folder

=#

# Convert to argument
PADDING_METHOD = :covert

include("CircularChannel.jl")
include("constants.jl")
include("utils.jl")
include("target.jl")

module Environment

    using .Main: Layer_type, get_ip_from_dev, IPv4Addr, _to_bytes,
   CircularChannel, ENVIRONMENT_QUEUE_SIZE

    export init_queue

    include("environment/headers.jl")
    include("environment/query.jl")
    include("environment/bpf.jl")
    include("environment/queue.jl")
    include("environment/env_utils.jl")

end

module CovertChannels

    using .Main: Layer_type, IPv4, Network_Type, TCP,
   Transport_Type, CircularChannel, MINIMUM_CHANNEL_SIZE
    using ..Environment: Packet, get_tcp_server, get_queue_data,
   get_layer_stats, get_header, get_local_host_count

    export covert_methods

    include("covert_channels/covert_channels.jl")
    include("covert_channels/microprotocols.jl")

end

module Outbound

```

```
    using .Main: Target, target, IPv4Addr, Network_Type,
   Transport_Type, Link_Type, Ethernet, IPv4, TCP, UDP, ARP,
   to_bytes, ip_address_regex, ip_route_regex, ip_neigh_regex, mac,
   to_net, _to_bytes, integrity_check, PADDING_METHOD,
   remove_padding, CircularChannel
    using ..CovertChannels: craft_change_method_payload,
   craft_discard_chunk_payload, craft_sentinel_payload,
   craft_recovery_payload, method_calculations, determine_method,
   covert_method, init, encode
    using ..Environment: Packet, get_socket, sendto,
   await_arp_beacon, get_local_net_host, AF_PACKET, SOCK_RAW,
   ETH_P_ALL, IPPROTO_RAW

    include("outbound/environment.jl")
    include("outbound/packets.jl")

end

module Inbound

    using .Main: MINIMUM_CHANNEL_SIZE, target, integrity_check,
   IPv4Addr, PADDING_METHOD, remove_padding, CircularChannel
    using ..Environment: init_queue, local_bound_traffic, Packet,
   get_local_ip
    using ..CovertChannels: SENTINEL, DISCARD_CHUNK,
   couldContainMethod, decode, covert_method, extract_method
    using ..Outbound: ARP_Beacon

    include("inbound/listen.jl")

end
```