Exploring the use of Adaptive Covert Communication Channels

u2053390

Warwick Manafacturing Group
University of Warwick

Supervised by Peter Norris

Abstract

...
Keywords: Adaptive Covert Communication Channels, Digital steganography, TCP/IP stack

Acknowledgements

Fanks

Contents

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 of Defense (1985), such as altering the Inter-packet delay (IPD) between packets in the TCP/IP stack. However, Active Wardens Wendzel and Keller (2012) 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. 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. The evaluation compares the proposed framework's performance to that of its non-adaptive counterparts in a simulated setting.

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 prisoners problem, outlined by Simmons (1984) Simmons (1984), where two prisoners, Alice and Bob, are being held in separate cells. The warden, Walter, of the prison 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 existance of a hidden message. This is where steganography comes in, Alice and Bob could use a technique such as the first letter of each line are combined to make a message (stegotext), it is important that the message read by Walter (covertext) is not suspicious.

There are two types of Warden that can 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 Walter could reword the 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). This shows the clear benefits to 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 covertext, the benefit to this is the nature of digital media. This opens up many pos-

sibilities in terms of the covertext, for example, a digital image can be used as the covertext, 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, and highlights the effectiveness of using digital media as the covertext. 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 covertext 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 asymmetrical encryption to create secure steganographic systems, Hughes (2000) 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".

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 legitmate communications, which makes the detection and prevention of these channels difficult.

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

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" ISI (1981), since this value is unique, it is essentially random, 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 an high bandwidth channel, while still maintaining a high quality of covertness.

The short-comings of this channel are outlined in Touch (2013), where it states "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 of course could be avoided by using fragmenting the packets, however this is less common and 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 via the three-way handshake of the TCP protocol, It works because a TCP server will respond to an intial connection request (SYN) that has defined an Initial Sequence Number (ISN), with an ackowledgement (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 orignal sender) with a SYN-ACK packet that has the data encoded as the acknowledgement number. Rowland (1996)

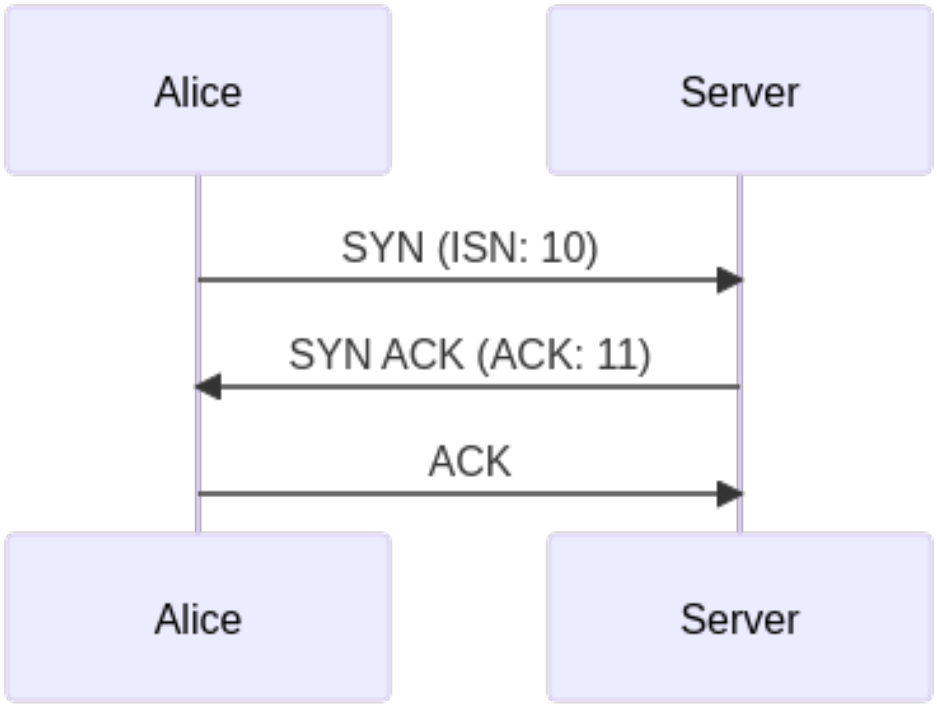


Figure 2.1: Normal TCP three-way handshake

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 actually originated from the sender. This makes it harder to determine if a covert channel is being used, and which systems are involved.

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

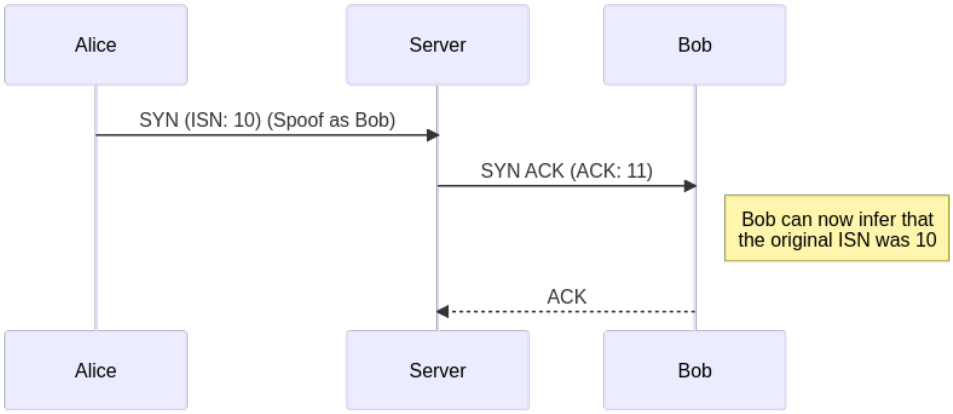


Figure 2.2: TCP three-way handshake with covert channel

The ISN generator is bound to "a (possibly fictious) 32-bit clock that increments roughly every 4 microseconds" Trf (1981), since the clock is a client-side clock, it is essentially random to an observer, for the first communication. However, a warden could possibly 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 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 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 nature of academic work, however these channels are not work well in real-world scenarios because they are 'reliant on a particular method' Yarochkin et al. (2008).

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 are applicable to the current environment, this phase is continuous to allow the system to adapt to live changes in the environment. The other 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 approaches, providing protections communication failures that can arise from interference or from preventative measures such as firewalls, active wardens, and protocol normalisation. Not only is this type of system harder to prevent, it is 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 signatiure based systems Yarochkin et al. (2008).

These additional benefits come at a cost, addtional 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, that makes more difficult to distinguish from noise Zhu et al. (2012).

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

3.2 Dynamic Routing in Covert Channel Overlays Based on Control Protocols

Inorder 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) & Trf (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 "contextful" 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 microprotocols 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 microprotocols Backs et al. (2012), due to their unchanging context.

This paper also proposes a dynamic underlying protocol change Backs et al. (2012), 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 of be implemented regardless of classification Backs et al. (2012), permitting both covert timing channels and covert storage channels.

Chapter 4

Objectives

This paper seeks to propose and evaluate a framework for an adaptive covert communication system. The framework will build upon the works outlined in the Related Work (3) chapter

Chapter 5

Design

5.1 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

5.1.1 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:

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 is incredibly cheap, and thus it is very versatile.

5.1.2 libpcap

libpcap is a cross-platform library for low-level network monitoring Group (2013). There are a number of 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

```
# 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)
```

Listing 1: Raw socket listener implementation

of using a library is that it is appropriately optimised. libpcap creates a memory-mapped ring buffer for asynchronous packet reception [PACKET_RX_RING - https://man7.org/linux/man-pages/man7/packet.7.html], this allows the userspace to the packet data without the need for 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, and thus this will not be included in my evaluation.

5.1.3 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). BPF's are register-based "filter machines" [The BSD Packet Filter] that makes 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.

5.1.4 Note

Eventually I decided that both implementations could be used in the framework, the raw socket was used to capture packets for the sender, without disrupting the main queue of packets. The additional code complexity of searching the queue did not outweigh the overhead of opening a raw socket, there are also some other benefits to using the raw socket, but I will discuss them in context in **??**.

5.2 Packet processing

When packets are captured, we want to process them into objects so we can idomatically 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}`.

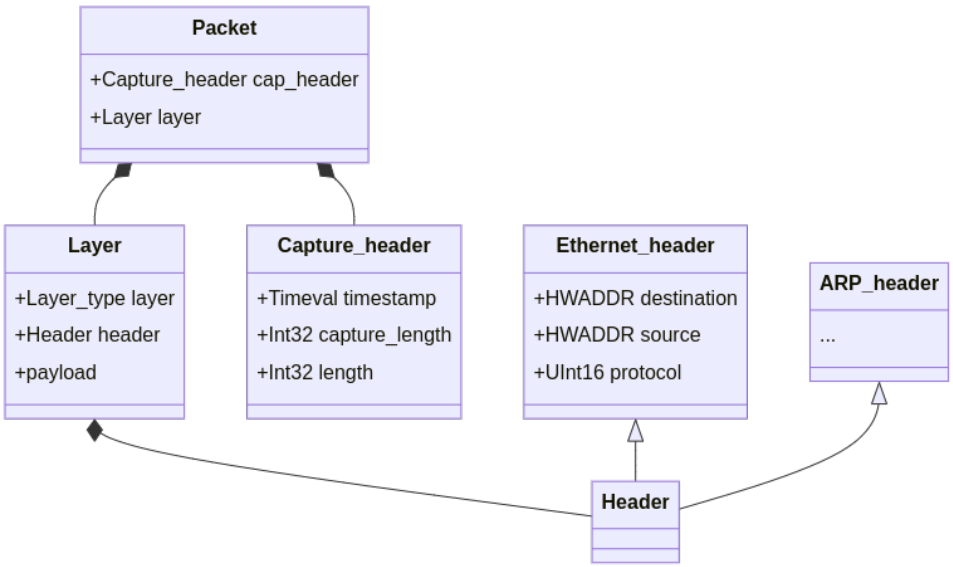


Figure 5.1: Packet structure

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:

```
1 struct Packet
2     cap_header::Capture_header
3     layer::Layer{Ethernet_header}
4 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.

5.3 The Queue

The queue is essential to the operation of the framework, it is how the packets are put by the listening thread, and 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 [JULIADOCS] that worked for the most part, however it introduced a bug into my codebase that 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 isn't a huge problem, since a covert channel is quite exposed on a silent network, it is not ideal. I eventually decuded 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 being blocked until the listening thread pushed something to the queue.

While I knew this action was not truely thread-safe, meaning I would likely be operating with stale data, it wasn't of great importance that my queue was truely 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 intermitently 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 idomatic datastructure, 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.

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

Listing 2: Old "Channel" implementation

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

Listing 3: New "CircularChannel" implementation

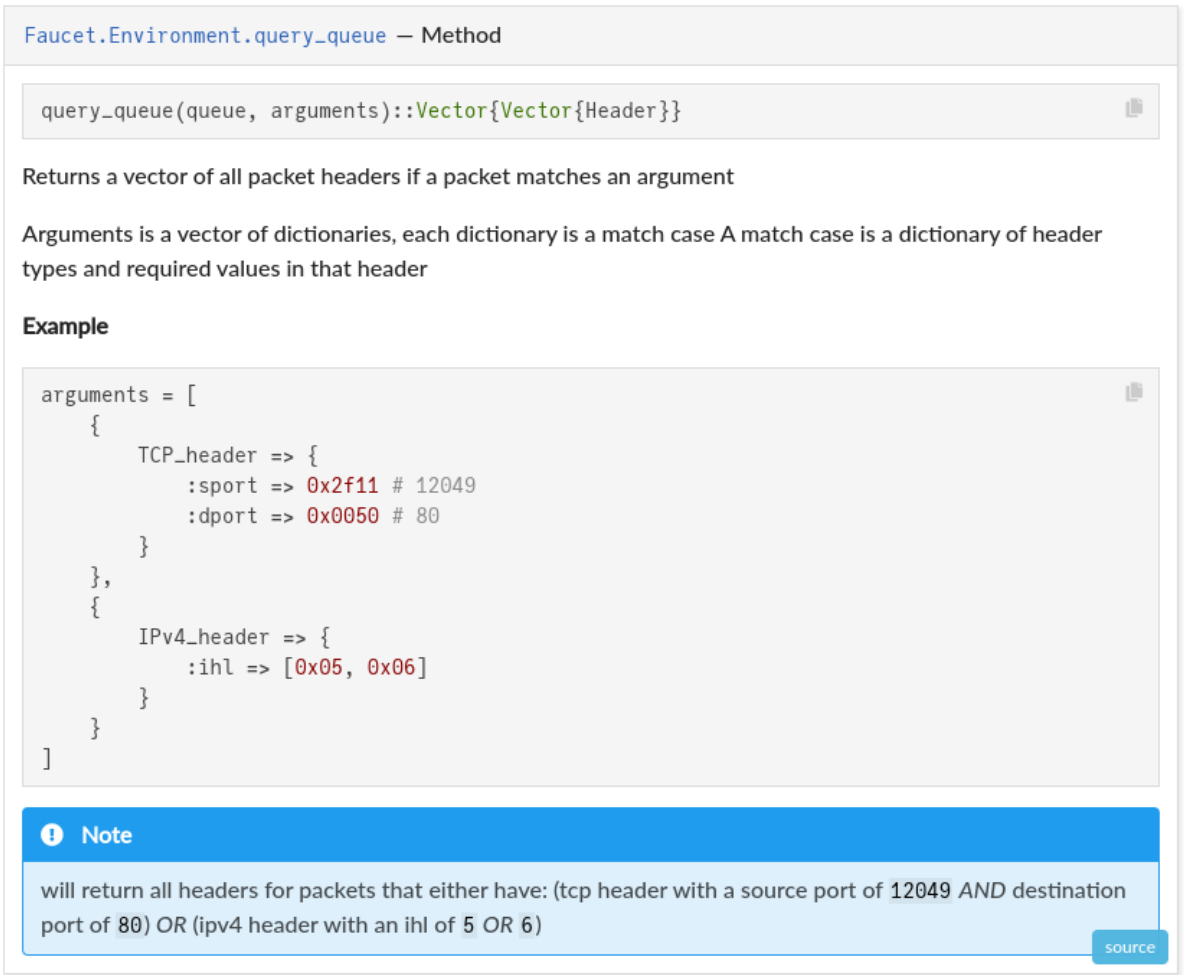
By overloading the `put!` and `take!` functions, I was able to implement their functionality for my new data structure, this meant for most of the code all I had to do was replace the type annotations of `Channel` with `CircularChannel`. This is ofcourse with the exception of the queue, whose changes can be see in **??**, and the `get_queue_data` function that I was using to improperly access the queue, thanks to my use of a central function for accessing this queue data I was able to change the implementation of the function without having to change the code that called it. I did this by creating a type convert funtion for the `CircularChannel{T}` to `Vector{T}`, where T is any data type, since I implemented this in a thread-safe way, I was able to use altered `get_queue_data` function to access the queue data without blocking the main thread.

I efficiently achieved this waiting using `Conditions`, they 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).

5.3.1 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 the 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:



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 5.2: A snippet from the Faucet documentation

5.4 The target (Arrangemnt)

Covert communication is reliant on a preshared arrangement **?**, this arrangement is a set of parameters that both parties will assume the other is using, it contains the following information:

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

The IP address is where the sender will send the packets, and the address 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.

5.5 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 targets 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 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 thes 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 used of covert modules, as outlined in Design of covert modules (5.7).

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 [man Packet(7)] 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 created 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.

5.6 Providing context to the sender

Many of the senders 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 (5.5))
- 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 (5.8), and the interface has already been discussed in the construction of the `sockaddr_ll` struct (see Sending packets (5.5)). The socket is simply a raw socket, in 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 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 is not applicable 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 idomatic.

5.7 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 idomatic, 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.
- An `decode` function.
- A `covert_method` struct instance.

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 (5.5). This function is passed the queue (see The Queue) to allow 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 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 5.6 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 crafting payloads for the channel, removing this burden from the module. The name of the channel is used in The target (Arrangemnt) (5.4) to ensure both parties are using the same cha

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 packets that are valid, so error handling can be omitted from covert modules, unless they are particularly complex.

All of the fields are used to creating an informed decision on which covert channel to use, as outlined in Decision algorithm (5.8).

5.7.1 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 target (Arrangemnt) (5.4)), 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 in.

The first check prevents very small channels being used in the framework, as they are not large enough to carry the Microprotocols (5.9). The second check prevents too many channels being used, as the micro protocols need to be able to uniquely identify each one, if there too many channels then they cannot be used (but since they have been registered they can still be considered when picking an algorithm, causing an failure in the communication).

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.

5.8 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 Queue (5.3). 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 (5.12.1), 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 (6.1).

5.8.1 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 (5.7), 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:

$$C_i = 1 - \frac{E_s - M_c}{10} \tag{5.1}$$

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 because the covertness of the channel is also correlated to its use in the environment, if a covertext is very prominent in the environment, the covertness of the method will be higher and thus a better choice than a slighty more covert, but less common method.

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

5.8.2 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:

$$R_i = \frac{E_h}{P_i * E_r * \frac{C_i}{2}} \tag{5.2}$$

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 being higher than 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 $[0, \infty)$.

5.8.3 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:

$$S_i = P_i * B_i * C_i \tag{5.3}$$

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 out 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.

5.8.4 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 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.

5.9 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 microprotocol, 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 commited 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.

Inorder to implement these protocls effectively, using the contextless design proposed in Dynamic Routing in Covert Channel Overlays Based on Control Protocols (3.2), packets need to desribe 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 (5.11). For the chunk discarding protocol, the space following the protocol is filled with payload data, this is because the sender will have to resend the previous data segments and without doing this there would be repeated fields, which would be suspicious to an observer.

For protocols that do not have auxillary 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.

5.9.1 Protocol compression

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 that otherwise, but since by their nature they are less frequent that data, this is a worthwhile tradeoff.

5.9.2 consequence of this approach

Unfortunately, using a single bit to denote data means that the orignal 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 excaserbated by the unknown nature of the packets pre-communication, as some channels have higher bit capacities, 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, 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 negligble and the code is easier to follow than using bit shifts to read the bits.

5.10 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 soluton that is better suited to covert channels, it does have a higher space cost, however the benefit is more of the data is random. The cost is logarithmically proportional to the size of the data, altough 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.

[HOW PADDING WORKS 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.

Both of these padding methods will be tested and compared in **??** (**??**).

5.11 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. The integrity of the communication is calculated using an 8-bit cyclic redundancy check (CRC-8). This allows errors to be recovered from using the discard chunk protocol (outlined in Microprotocols (5.9)).

This requires the receiver to be able to communicate the integrity of the message back to the sender, so the sender can decide wheter 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 received 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 garuntee 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 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 challenge to be perfomed 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.

To improve the covertness of the ARP Beacon, the sender will (if the channel is large enough) provide an offset to the receiver, 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. 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 wheter the integrity is correct without sending its response, so this is a tradeoff that must be made.

There are some addresses that 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.

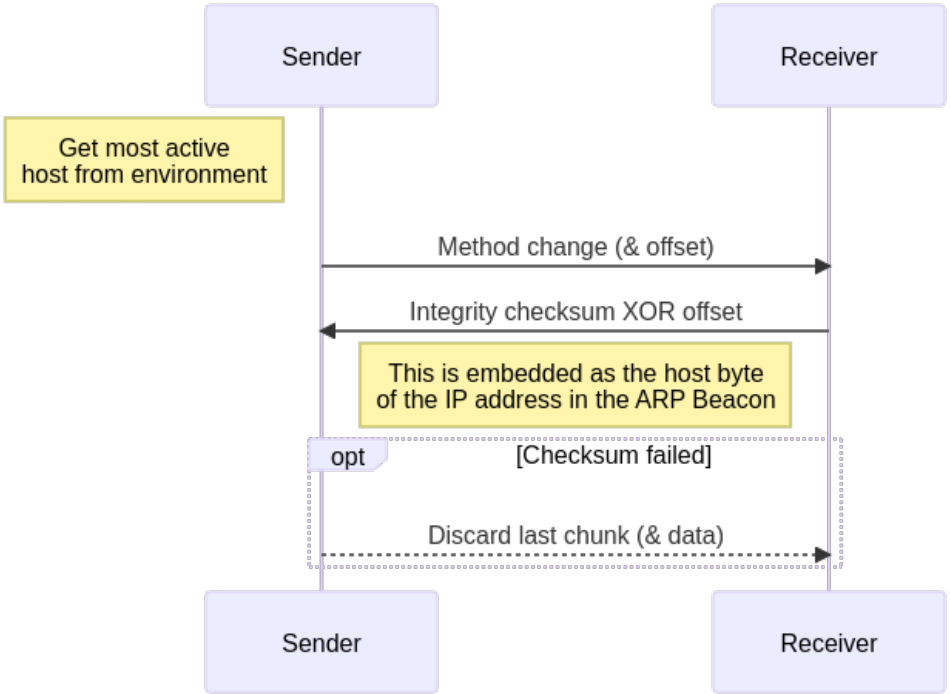


Figure 5.3: Integrity challenge and response (optional parts in brackets)

Since the sender knows what the 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 perfomed irrespective of the underlying covert channel, but it might require limitations on the environment (just as ARP requires a local network).

5.12 Communication 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 secnarios the framework must be able to recover the communication effort, the sender and receiver need a sort of 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 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 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 acts 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 (5.11), however it is used in combination with the length of verified data, this is the length of the last sender verified data. There are some cases where the receiver will have 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 receiver 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 it may not be consistent between the sender and receiver.

5.12.1 Penalising methods

This recovery process is not great 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 being chosen again unless there are no better alternatives. The reason we do not permenantly block the protocol is beacuse it can fail for reasons that are temporary, but we cannot know that until we try again, It is not a good idea to constantly retry failing methods as if they are being caught by wardens then it is much more suspicious for a many packets to have been prevent rather than just a few.

Chapter 6

Implementation

6.1 Algorithm implementation

The input variables, and output of the implementation can be found in Decision algorithm (5.8). 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 variables names, allowing me to use the same mathematical notation in the code as in the design documenation. The following is an extract from the source code of the framework (`/src/covert_channels/covert_channels.jl`)

```
1  function method_calculations(covert_methods::Vector{covert_method},
       env::Dict{Symbol, Any}, Eₚ::Vector{Int64}=[],
       current_method::Int64=0)::NTuple{2, Vector{Float64}}
2      # Get the queue data
3      q = get_queue_data(env[:queue])
4
5      # Covert score, higher is better : Method i score = scores[i]
6      S = zeros(Float64, length(covert_methods))
7      # Rate at which to send covert packets : Method i rate =
       rates[i]
8      R = zeros(Float64, length(covert_methods))
9
10     if isempty(q)
11         @error "No packets in queue, cannot determine method" q
12         return S, R
13     end
14
15     #@warn "Hardcoded response to determine_method"
16     L = [get_layer_stats(q, Layer_type(i)) for i ∈ 2:4]
17
18     # Eₗ : Environment length : Number of packets in queue
19     Eₗ = length(q)
20
21     # Eᵣ : Environment rate : (Packets / second)
```

```
22      Eᵣ = Eₗ / abs(last(q).cap_header.timestamp -
    first(q).cap_header.timestamp)
23
24      # Eₛ : Environment desired secrecy : User supplied (Default: 5)
25      Eₛ = env[:desired_secrecy]
26
27      # Get the count of hosts local to the supplied ip (we don't
    want to consider external ones).
28      Eₕ = get_local_host_count(q, env[:dest_ip])
29
30      for (i, method) ∈ enumerate(covert_methods)
31          Lᵢ_temp = filter(x -> method.type ∈ keys(x), L)
32          if isempty(Lᵢ_temp)
33              @warn "No packets with valid headers" method.type L
34              continue
35          end
36          # Lᵢ : the layer that method i exists on
37          Lᵢ = Lᵢ_temp[1]
38
39          # Lₛ : The sum of packets that have a valid header in Lᵢ
40          Lₛ = +(collect(values(Lᵢ))...)
41
42          # Lₚ : Percentage of total traffic that this layer makes up
43          Lₚ = Lₛ / Eₗ
44
45          # Pᵢ is the percentage of traffic
46          Pᵢ = Lₚ * (Lᵢ[method.type] / Lₛ)
47
48          # Bᵢ is the bit capacity of method i
49          Bᵢ = method.payload_size
50
51          # Cᵢ is the penalty / bonus for the covertness
52          #  has bounds [0, 2] -> 0% to 200% (± 100%)
53          Cᵢ = 1 - ((method.covertness - Eₛ) / 10)
54
55          # Score for method i
56          #  Pᵢ * Bᵢ : Covert bits / Environment bits
57          #  then weight by covertness
58          #@info "S[i]" Pᵢ Bᵢ Cᵢ Pᵢ * Bᵢ * Cᵢ
59          S[i] = Pᵢ * Bᵢ * Cᵢ
60
61          # Rate for method i
62          #  Eᵣ * Pᵢ : Usable header packets / second
63          #  If we used this much it would be +100% of the
    environment rate, so we scale it down
64          #  by dividing by hosts on the network, Eₕ.
65          #  then weight by covertness
66          #  We don't want to go over the environment rate, so
    reshape covertness is between [0, 1] (1 being 100% of env rate)
67          #  (Eᵣ * Pᵢ * (Cᵢ / 2)) / Eₕ : Rate of covert packets /
```

```
    second
68          #  ∴ 1 / Eᵣ * Pᵢ * (Cᵢ / 2) : Interval between covert
    packets
69          #@info "R[i]" Eₕ Eᵣ Pᵢ Cᵢ/2 Eₕ / (Eᵣ * Pᵢ * (Cᵢ / 2))
70          R[i] = Eₕ  / (Eᵣ * Pᵢ * (Cᵢ / 2))
71      end
72
73      # Eₚ (arg) : Environment penalty : Penalty for failing to work
    previously
74      for i ∈ Eₚ
75          S[i] *= 0.1 # 10% of original score
76      end
77
78      # Allow for no current method (as the case is when recovering)
79      current_method != 0 && (S[current_method] *= 1.1) # Encourage
    current method (+10%)
80
81      return S, R
82  end
```

The `method_calculations` function only does the calculations, not the selection of the best method, that 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).

6.2 Integrity check

Chapter 7

Future work

The framework proposed has many issues that prevent its effective use in a real-world scenario:

- No attempt at having a valid covertext:
  - 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 quantity of possible covertexts, 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 bad, it still evaluates the protocol to use very well.

Bibliography

Transmission control protocol. RFC 793, September 1981. URL https://www.rfc-editor.org/info/rfc793.

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.

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.

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.

Group, The Tcpdump. libpcap. https://www.github.com/the-tcpdump-group/libpcap, 2013.

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.

ISI, . Internet protocol. RFC 791, September 1981. URL https://www.rfc-editor.org/info/rfc791.

Kahn, David. *The Codebreakers: The Story of Secret Writing*. 1973. URL https://doc.lagout.org/security/Crypto/The%20CodeBreakers%20-%20Kahn%20David.pdf.

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

of Defense, U.S. Department. 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.

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

Rowland, Craig. Covert channels in the tcp/ip protocol suite, 1996. URL https://firstmonday.org/ojs/index.php/fm/article/view/528/449.

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.

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

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/.

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.