Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

# Secure Port-Knocked Communications

Hugues Fafard

TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

# Secure Port-Knocked Communications

# Sichere Port-Knocking-geschützte Kommunikation

| | |
|---|---|
| *Author* | Hugues Fafard |
| *Supervisor* | Prof. Dr.-Ing. Georg Carle |
| *Advisor* | Sree Harsha Totakura, M. Sc. |
| *Date* | May 15, 2016 |

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, May 15, 2016

Signature

## Abstract

Port-Knocking allows services to be hidden behind a firewall that blocks all access, yet still be accessible by authorized users once a valid port-knocking request has been sent to the server hosting the service.

Many port-knocking solutions are available, each with their own protocol that needs to be supported by the client. Most applications do not support any of those protocols and several applications also do not support encrypting their traffic. This thesis aims to design a solution that allows applications to offload the port-knocking and the encryption. We aim to use the port-knocking library *sKnock*, which uses certificates for port-knocking authentification. Additionally, a Proof-of-Concept is implemented in the scope of this thesis.

## Zusammenfassung

Port-Knocking erlaubt Diensten, sich hinter einer Firewall zu verstecken, welche sämtlichen Verkehr blockt. Dennoch bleiben diese Dienste für authorisierte Benutzer erreichbar, nachdem eine gültige Port-Knocking-Anfrage für den Benutzer beim Server, welcher den Dienst bereitstellt eingegangen ist.

Es sind zahlreiche Port-Knocking-Lösungen verfügbar, welche alle ihr eigenes Protokoll haben, welches vom Client unterstützt werden muss. Die meisten Programme unterstützen jedoch keines dieser Protokolle und einige Programme unterstützen des Weiteren keine verschlüsselte Kommunikation. Diese Arbeit hat zum Ziel, eine Lösung zu entwerfen, welche es Programmen erlaubt, das Port-Knocking und die Verschlüsselung des Datenverkehrs auszulagern. Als port-knocking library soll *sKnock* genutzt werden, welches Zertifikate zur Port-Knocking authentifizierung nutzt. Zusätzlich wird im Rahmen dieser Arbeit eine Proof-of-Concept implementierung entwickelt.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Goals of the Thesis

The major goal of this thesis is to develop a solution to allow existing applications to offload port-knocking and traffic encryption. This will involve some form of tunneling to ensure the traffic encryption. The solution should present as little effort as possible to set up, both for the service's operator and for the user. If possible, the solution should avoid requiring code changes to existing applications in order to be used. Furthermore, the port-knocking authentication should be done via certificates. To achieve this, *sKnock* should be used as a port-knocking library.

## 1.2 Outline

Chapter 1 introduces the concept of port-knocking and explains the rationale behind developing a solution such as ours. This is followed by a brief introduction of *sKnock* as our port-knocking library in Chapter 2. This chapter also explains the three approaches which we considered and explains their respective strengths and weaknesses. Based on this analysis, an approach is selected and its implementation is described in Chapter 3. General concerns about additional features and how to implement them precede the actual description of the implementation. Some limitations of the implementation follow. The test setup used to test the implementation and the results of the test are explained in Chapter 4. Finally, Chapter 5 suggests some improvements for future versions and gives some pointers on how to do this before summarizing the results of this work.

## 1.3 Motivation

### 1.3.1 History of Network Security

When computer networks were first developed in the late 1960s and early 1970s, security was not considered a design goal. The scientists designing the first computer networks wanted to interconnect computers in an easy and reliable way. Adding security requirements to the goals would have added significant complexity to the protocols and thus slowed their development. Furthermore, those networks were only accessible to people deemed trustworthy, such as fellow scientists or the military [1].

It was not until the Internet became more widespread and government computers were attacked in the 1980s that network security became a concern. But only when the Internet became available to the public in the 1990s, network security become a major issue [1]. During this time, public services started to appear. These are by definition accessible by anyone, which is an exploitable weakness. While some threats—such as (Distributed) Denial of Service attacks—often have easily recognizable patterns that can be detected to thwart the attacks, any user can connect to such an open service, whether or not the user has a legitimate interest in the service as such. This is much harder, if not impossible, to detect and allows the user to gain information about the service and potential vulnerabilities. This can partially be partially automatized with port scans, which are, however more easily detectable.

### 1.3.2 Port-Knocking

Port-Knocking is an approach to run services behind a firewall, while still allowing users with a legitimate interest in the service to access it. It is inspired by an ancient security concept from which it also derives its name. A person wanting to enter a restricted area would use a secret knock sequence as proof of his authorization to pass. When transferring this concept to computer networks, the firewall would act as the closed door and the client would attempt to connect to a previously specified sequence of ports. If the sequence is correct, the desired port is opened and the client can connect [2].

This method does, however, have several severe security flaws. First and most importantly, any attacker can eavesdrop on the transmission of such a port-knocking sequence and learn it. The attacker is then able to produce the sequence themselves to gain access to the service. Furthermore, sending several packets before even beginning to actually connect to a service introduces a delay. Additionally, any of those packets can get lost during transmission or the packets may arrive in a different order than they were sent. Both of these phenomena lead to a failure of the port-knocking requiring the client to retry anew.

There are several port-knocking approaches available, some of which successfully solve some or even all of these issues. However, they all require the client to be aware of the port-knocking and support the specific port-knocking protocol. In many cases, a service provider cannot or is not willing to change the source code of the client application to support a port-knocking protocol. This is, after all, a considerable effort. Furthermore, the source code of the client application may not be available to the service provider. A solution that allows these applications to offload the port-knocking to another application would make it easier for service providers to protect their service(s) with port-knocking.

Since there are still many applications which do not support encrypted communication, this can be combined with offloading of the encryption to create a solution that enables any program to use a secure communication channel to a port-knocked service with little to no code changes.

# Chapter 2

# Background

This chapter begins with giving an overview of *sKnock*, the port-knocking library we will use. This is followed by a description and theoretical evaluation of three possible approaches to reach the goal of this thesis. One of these approaches is then selected and this decision explained.

## 2.1 The Port-Knocking Library: sKnock

We have chosen to use the *sKnock* library to provide the port-knocking functionality in this thesis. *sKnock* was recently developed as part of a Bachelor's thesis by Daniel Sel and features an approach to port-knocking, which we find promising. It is, however, debatable whether it can still be called “port-knocking”, as it does not follow the name-giving approach commonly associated with this term, though it does fulfill the same purpose.

Instead of attempting to connect to a sequence of predefined ports to unlock a desired port, it authenticates to the daemon running on the server with a single packet using cryptography and additional information in the payload [3]. Such approaches are sometimes also called Single Packet Authorization (SPA) and can be considered next-generation port-knocking [4].

Unlike other SPA solutions, *sKnock* was designed with scalability and performance in mind. As such, it requires neither the server daemon nor the client to keep track of any state or other per-user information, such as credentials. A Proof-of-Concept implementation bearing the same name was developed during the thesis. It runs in user space and is designed for platform independence, though only Linux support has been implemented yet. While it suffers from performance issues concerning the Elliptic Curve Cryptography (ECC), profiling has shown that this is mostly due to the choice of Python as a programming language and could easily be mitigated [3].

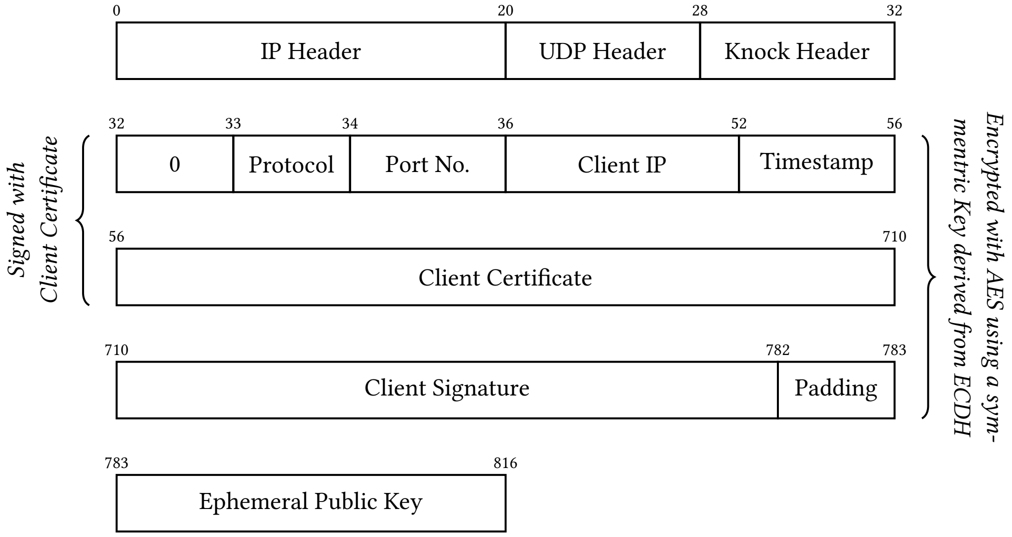

Figure 2.1: An *sKnock* packet. Source: [3]

### 2.1.1 Basic Functioning

TCP requires several packets to be exchanged while performing a handshake thereby increasing the delay caused by performing the port-knocking. Furthermore, this would also make it impossible to authenticate using a single packet. Thus, *sKnock* sends its port-knocking request over UDP. The UDP header is followed by a minimal *sKnock* header, containing a 1-Byte magic number to identify *sKnock* and a 3-Byte field with a protocol version number [3]. Figure 2.1 shows a complete *sKnock* request, including the IP, UDP and *sKnock* headers. It is a schematic overview and the fields are not necessarily to scale.

*sKnock* sends the requested port number and signs the request with its private key. To avoid having to manage user public keys on the server, the corresponding certificate—signed by a CA trusted by the server—is appended to the payload. To prevent an attacker from gaining information about the infrastructure, the payload is encrypted with a symmetric key derived from the server's public key and an ephemeral ECC key-pair based on the same curve. Additional measures have been implemented to prevent Man-in-the-Middle (MitM) [5] and replay attacks [6].

As shown in figure 2.1, the first field of the payload is a zero filled byte. Its purpose is to facilitate efficient checking of the decryption's success to rule out malformed or incorrectly encrypted packets. Next is another 1 Byte long field containing the protocol for which the desired port is to be opened, where '0' represents UDP and '1' represents TCP. This is followed by the port number to be opened.

The IP address for which the port is to be opened is also included to protect against Man-in-the-Middle attacks. To support IPv6, this field is 16 Bytes long. The downside

| **Security Bits** | **Sym. algorithm** | **RSA key-size [bits]** | **ECC key-size [bits]** |
|---|---|---|---|
| 80 | Skipjack | 1024 | 160 |
| 112 | 3DES | 2048 | 224 |
| 128 | AES-128 | 3072 | 256 |
| 192 | AES-192 | 7680 | 384 |
| 256 | AES-256 | 15360 | 512 |

Table 2.1: Security strength comparison of RSA and ECC. Sources: [10, 11]

of this are issues with NAT traversal, the IP address as seen by the Internet differs from the IP address as seen by the client sending the request. For a request through a NAT to be successful, the client requires knowledge of the public IP address. Replay protection is achieved by the inclusion of a 4-Byte timestamp [7].

The next field contains the certificate belonging to the private key used to sign the entire payload up until now, including the certificate itself. This signature is stored in the next field, which is padded to a deterministic length of 72 Bytes, as ECDSA signatures may vary in length with 72 Bytes being the maximum length when using a 256 bit key [8].

The entire payload up until now is encrypted using a symmetric key derived from the server's public key and an ephemeral ECC key-pair based on the same curve. For the server to be able to decrypt the packet, the public key of the ephemeral pair needs to be appended to the message, constituting the only unencrypted field of the payload [3].

### 2.1.2 Elliptic Curve Cryptography

Like RSA, Elliptic Curve Cryptography (ECC) is an approach at public-key cryptography and as such can be used in asymmetric encryption and signature. However, they both rely on different mathematical assumptions. Unlike RSA, which relies on the assumption that the factorization of large numbers is significantly slower and more computationally intensive than multiplying large prime numbers [9], ECC relies on the assumption that reversing a point addition on elliptic curves of finite fields is significantly slower and more computationally intensive than the addition itself. This assumption is known as the elliptic curve discrete logarithm problem (ECDLP) [8].

Advantages of ECC include significantly smaller key sizes than RSA for comparable security assurances. The size difference grows drastically as the security assurance increases [10, 11]. Table 2.1 shows a comparison of key-size at a comparable security level. ECDSA signature sizes are also significantly smaller than RSA signatures [12]. Furthermore, ECC is faster than RSA for key generation and signature generation. On the other hand, RSA is faster in regards to signature verification when using small key sizes [11, 13].

Since *sKnock* relies on sending the client certificate and a signature as part of the port-knocking request, these smaller sizes are an important advantage, as they help keep the packet small. This is especially important because *sKnock* is—for performance reasons—not equipped to reconstruct fragmented requests [3].

## 2.2 Possible Approaches

The primary goal of providing existing applications with an encrypted communication channel and to a port-knocked service can be reached via several approaches. Among them, we have identified a local proxy, an SSH Tunnel and a VPN Tunnel as the three most promising approaches to be further considered.

The following sections contain a more detailed description of the approaches, as well as their potential advantages, drawbacks and other interesting properties.

### 2.2.1 Proxy

The first approach is a local proxy to which the application connects instead of connecting directly to the service. The proxy would handle the port-knocking and encrypt the traffic. However, this means that, unlike a traditional proxy, this approach requires a counterpart on the server to decrypt the traffic before forwarding it to the service. This solution would be the most flexible one, as it would be completely implemented by us. However, this also makes it the most complex solution, as no code from other projects, such as *OpenSSH* or *OpenVPN* can be reused.

#### 2.2.1.1 Functioning

When an application connects to the proxy, it passes the target service's IP address and port on to the proxy, which then performs the port-knocking to open the port on which a counterpart is listening on the server hosting the target service. Once the port is open, the proxy connects to this counterpart and establishes an encrypted tunnel. Subsequent traffic sent to the proxy is encrypted and sent through this tunnel to the proxy's counterpart on the server, which in turn decrypts it and locally forwards the now unencrypted traffic to the service.

*Considerations*—A suitable proxy protocol would have to be selected to transmit information, such as the destination IP address and port. The SOCKS5 protocol would be a good choice for this, as it is well-known and supports both IPv4 and IPv6, as well as TCP and UDP connections [14]. Most browsers already implement support for the SOCKS5 protocol.

However, *sKnock* needs its client private key, its client certificate, the server's certificate and the CA's certificate to perform the port-knocking [3]. This requirement is rather unique and no existing proxy protocol is designed to convey this information. As a result we either need to create a new proxy protocol or design an extension to SOCKS5.

Another way to solve this particular problem is to shift the management of the certificate information from the application to the proxy itself. The proxy would then have to be configured with service profiles. Each of these profiles would assign a tuple of client private key, client certificate, server certificate, and CA certificate to a tuple of server and port. Whenever a connection to a service is established, the proxy can then look up the necessary certificate information without the applications needing to transmit or even know about the certificates. The profiles could be provided by the service operator.

Since the proxy only listens to connections from localhost, it can be judged unnecessary for the application to authenticate itself to the proxy depending on the security requirements.

It is also important that the server accepts requests addressed to the loopback interface, as from the service's perspective, packets sent through the proxy will be originating from the same machine and may be addressed to the loopback interface, depending on the counterpart's implementation.

#### 2.2.1.2 Communication Overhead

Gauging the communication overhead of this solution is difficult at this moment because it depends on too many design decisions yet to be made, such as the choice of ciphers and MACs used. It is however possible to minimize the per-packet overhead far beyond what is possible with the other solutions, as they use more general-purpose tools that are not as specialized as the proxy could be.

#### 2.2.1.3 Advantages & Disadvantages

*Advantages*—As stated above, this approach would be the most flexible one, leaving us free choice of ciphers and MACs. We would also be completely free to design how the proxy and it's counterpart would communicate and minimize the overhead. It would be possible to avoid the additional overhead incurred by encapsulating the entire original packet including its IP and TCP/UDP headers, as would be necessary in the approaches presented in sections 2.2.2 and 2.2.3. This gives this approach the potentially lowest overhead.

Since so many implementation details are open, it is however not possible to pinpoint further advantages that might result from implementation specifics.

*Disadvantages*—Obviously, the application needs to be aware of this solution, as it needs to address the proxy instead of directly connecting to the service. This requires either modification of the source code in the case of a hardcoded service, or additional configuration by the user in the case of a user-specified service.

Furthermore, the communication between the application and the proxy is unencrypted. The applies to the communication between the service and the proxy's counterpart, which is running on the same server as the service. Since these communications all happen on the local machine, the machine must already be compromised for an attacker to exploit this. Thus, this can be considered a minor security risk, but it should not be forgotten nevertheless.

This solution also requires the design of an entire security scheme to ensure that the forwarded traffic stays confidential and cannot be manipulated. This opens up many potential pitfalls, whereas other solutions described in sections 2.2.2 and 2.2.3 allow the use of time-proven security schemes.

The most important disadvantage of this solutions however, is the fact that the application's source code may have to be modified to add support for the proxy protocol. In the event of creating a new protocol or expanding SOCKS5, this will definitely be necessary. Even if the certificate information is moved to proxy configuration and SOCKS5 is used, not all applications will already support the proxy protocol. Services which are accessed through web browsers should not have this problem, as most web browsers already implement support for the SOCKS5 protocol.

### 2.2.2 SSH Tunnel

Another promising approach is to establish an SSH tunnel and connect to the service through that tunnel. An SSH tunnel connects a local port to a single port on a remote machine through the SSH server. The SSH client would be modified to handle the port-knocking and needs to have established an SSH tunnel before the actual application attempts to connect to the service.

If following this approach, we would modify the *OpenSSH* [15] client. *OpenSSH* is an open-source suite of SSH utilities released under a BSD-style license [15, 16]. It is one of the most widespread SSH suites and provides among other useful tools an SSH client and server. The modified *OpenSSH* client would be expanded to use the *sKnock* library to perform the port-knocking. Furthermore it could be stripped down to only the necessary functionality to create an SSH tunnel, though this is not necessary.

#### 2.2.2.1 Functioning

The modified SSH client is started before the application attempts to connect to the service and uses the *sKnock* library to port-knock the SSH port[1] on the server hosting the service. Once the port has been opened in the server's firewall, the SSH client opens an SSH Tunnel to that port, connecting a chosen local port to the service's port on the server. The application can then connect to the selected port on the local machine to have its traffic sent over a secure channel to the service.

While *OpenSSH* also runs on Microsoft Windows through cygwin [15], a modified version of *PuTTY* [19] could be made for Windows. *PuTTY* is one of the most popular SSH clients on Windows and is released under the MIT license [19].

The service's operator also needs to keep in mind that packets sent through the SSH tunnel will be seen by the service as originating from the same host and may be addressed to the loopback interface. The service thus needs to serve requests addressed to the loopback interface and cannot be bound only to the server's public IP address. This also applies to clients which need to bind to an address, such as some peer-to-peer applications.

#### 2.2.2.2 Communication Overhead

The overhead incurred when sending a packet through an SSH tunnel is variable, depending on the selected cipher, MAC algorithm and original packet length [20, 21].

The original package sent by the application to the service including its IP and TCP Header is transmitted as the payload of an SSH tunnel packet. As such, it inevitably incurs the overhead of 20 Bytes for the additional IPv4 header [22] or 40 Bytes for an additional IPv6 header [23]. Another 20 Bytes are added for the additional TCP header [24].

Further overhead is incurred by the SSH protocol itself. An SSH tunnel packet starts off with a 4 Byte field containing the length of the encrypted part of the packet followed by a single Byte containing the length of the padding [20]. Next are another Byte-long field containing a constant marking the packet as tunneled data and a 4 Byte channel ID, followed by the actual payload [21]. This entire structure is padded to a multiple of the cipher block size. The padding must be at least 4 Bytes and at most 255 Bytes long [20].

The last field contains the MAC and as such also has a variable length depending on the MAC algorithm used [20]. Available MAC algorithms are hmac-sha1, hmac-sha1-96, hmac-md5, hmac-md5-96 [20], hmac-sha2-256, and hmac-sha2-512 [25]. *OpenSSH*

[1] SSH has been assigned port 22 by the IANA [17] but it can be configured to use another port [18].

| Algorithm | Status | MAC size [Bytes] |
|---|---|---|
| umac-32 | – | 4 |
| umac-64 | – | 8 |
| umac-96 | – | 12 |
| hmac-sha1-96 | Recommended | 12 |
| hmac-md5-96 | Optional | 12 |
| hmac-md5 | Optional | 16 |
| umac-128 | – | 16 |
| hmac-sha1 | REQUIRED | 20 |
| hmac-sha2-256 | Recommended | 32 |
| hmac-sha2-512 | Optional | 64 |

Table 2.2: MAC algorithms specified by the SSH standard[a]. Sources: [20, 25, 26]

[a] The "`umac-*`" algorithms are not part of the official SSH standard. They are listed here because they are supported by *OpenSSH* nevertheless.

further supports the non-standard umac algorithms umac-32, umac-64, umac-96, and umac-128 [15, 26]. Table 2.2 shows a list of MAC algorithms available to *OpenSSH* and the respective size of their resulting MAC.

The SSH specification allows a large variety of CBC- and CTR-mode ciphers [20, 27]. Table 2.3 shows a list of available ciphers in *OpenSSH* and their respective block size. Many of those ciphers are regarded as broken today and are only contained for compatibility with legacy clients. The cipher is used to encrypt the packet length, padding length, SSH payload[2] and padding fields. The *OpenSSH* package distributed with Debian 8.4 prefers aes128-ctr as cipher and umac-64-etm[3] as MAC by default [18], though this may vary depending on the *OpenSSH* binary used. Note that `arcfour` as a stream cipher does not have a CBC/CTR mode or a block size.

Figure 2.2 is a graphical representation of an SSH tunnel packet using these default settings making for an 8 Byte MAC. A packet sent through an SSH tunnel with these settings thus incurs an overhead between 58 Bytes (With minimum padding of 4 Bytes) and 313 Bytes (With maximum padding of 255 Bytes) depending on the length of the original packet. When using IPv6 those values are 20 Bytes higher, since an IPv6 header is 20 Bytes longer than an IPv4 header.

Further overhead arises from SSH protocol communications. This overhead is independent of the number of packets sent through the tunnel and covers, for example tunnel establishment and key-negotiation. It is also recommended to rekey after every gigabyte sent or every hour depending which happens first [20]. RFC4344 further recommends to rekey after every $2^{32}$ packets sent or ever $2^{L/4}$ blocks encrypted where $L$ is the cipher

[2] In our case, the SSH payload is the tunneled packet.

[3] Encrypt-then-MAC (ETM) means that the MAC is calculated from the ciphertext instead of the plaintext.

| Algorithm | Status (CBC) | Status (CTR) | Block size [bits] |
|---|---|---|---|
| 3des | REQUIRED | Recommended | 64 |
| blowfish | Optional | Optional | 64 |
| twofish256 | Optional | Optional | 128 |
| twofish192 | Optional | Optional | 128 |
| twofish128 | Optional | Optional | 128 |
| aes256 | Optional | Recommended | 128 |
| aes192 | Optional | Recommended | 128 |
| aes128 | Recommended | Recommended | 128 |
| serpent256 | Optional | Optional | 128 |
| serpent192 | Optional | Optional | 128 |
| serpent128 | Optional | Optional | 128 |
| idea | Optional | Optional | 64 |
| cast128 | Optional | Optional | 64 |
| arcfour | — | — | — |

Table 2.3: Ciphers available to *OpenSSH*. Sources: [20, 27]

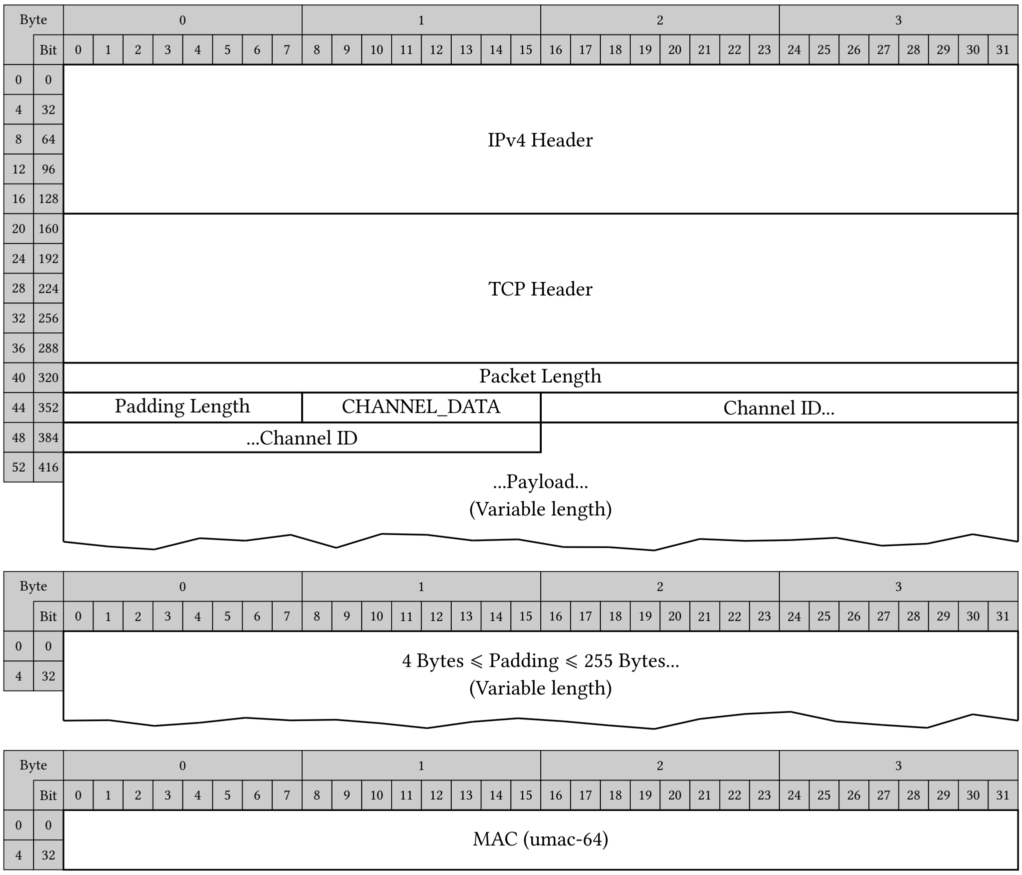

Figure 2.2: An SSH tunneled packet

block length [27]. Due to the low amount of data transmitted and the low frequency at which such control messages are exchanged in pure tunneling use, this overhead becomes negligible fairly quickly if the connection is in use long enough.

#### 2.2.2.3 Advantages & Disadvantages

*Advantages*—One of the biggest advantages of this approach is the widespread use of SSH. Most servers already have an SSH server installed and would therefore most likely only need little to no additional configuration. In the case it is not already present on the server, it is usually easy to obtain.

A user not having access to a modified *OpenSSH* client that can handle the port-knocking could use the reference *sKnock* client and then manually open an SSH tunnel with any SSH client. Once this is done, the application can connect to the service through this tunnel. While this is less convenient, it is well within the grasp of anyone possessing at least intermediate knowledge of such technology.

Another advantage of this solution is the time-proven security of the SSH protocol itself and the *OpenSSH* implementation of the protocol [16]. This may not only further the administrator's and user's trust in this solution, but also avoids the burden of needing to develop a secure means of communication from the ground up.

*Disadvantages*—This solution does however have several rather severe disadvantages. One of the most grave ones is the lack of support for X.509 certificates, despite RFC6187 specifying the use of X.509 certificates for SSH authentication [28]. Instead, it uses it's own minimalistic certificate format. Not implementing support for X.509 certificates was a conscious decision of the development team in order to reduce certificate complexity and theoretical attack surface [29]. It is therefore likely not desirable to add this functionality. This means that the certificates used by *sKnock* cannot be reused by *OpenSSH* to authenticate with the server, thus requiring the variations of each certificate to be passed to the binary even though the cryptographic material—the keys—is the same. On the other hand, this allows to make the authentication for *OpenSSH* independent from the authentication to open a port.

Another big issue is that SSH tunnels only support tunneling TCP connections [18, 30]. While many services one might want to hide behind port-knocking employ TCP connections, this does not hold true for all services. Therefore, restricting applications to TCP connections severely reduces the benefit of this approach.

As mentioned above, SSH tunnels only connect a single local port to a single port on a remote host. In most cases, this should be unproblematic as most services only require one port. However, some services, such as FTP in passive mode, for example, require several ports to function. In this case, several SSH tunnels have to be established manually. This is also necessary when accessing multiple services on the same server. Additionally, the destination port and IP need to be known when establishing the SSH tunnel. This is problematic for protocols that use random ephemeral ports, as those ports are not yet known at that time.

Furthermore, *OpenSSH* works over TCP leading to SSH tunnels running TCP-over-TCP. This can have a negative impact on the throughput [31, 32], especially when dealing with packet loss above a few percent [33]. This is due to TCP being designed to run over an unreliable carrier and being equipped with mechanisms to deal with packet loss. TCP connections have a Retransmission Timeout (RTO) in which the sender expects an ACK from the receiver. If the ACK is not received by the sender within the RTO, the packet is queued for retransmission and the RTO is increased [24]. When tunneling a TCP connection through another TCP connection, both run this coping mechanism in parallel and do not necessarily have the same RTO.

When both RTOs have the same value, lost packets will only be retransmitted twice (Once by the outer and once by the inner connection), which is bearable. But when the outer RTO is longer then the inner RTO, any packet loss can lead to what is often called a “TCP-Meltdown”. The inner RTO being lower, the inner connection will consider the packet lost and queue it for retransmission before the outer connection’s RTO has run out. When the outer RTO runs out, the packet in question is also queued for retransmission by the outer connection. This causes a large amount of unnecessary retransmissions to build up, possibly saturating the connection to the point where it breaks [32, 33].

### 2.2.3 VPN Tunnel

Lastly, it is possible to establish a VPN tunnel and connect to the service through this tunnel. There are several VPN suits with different protocols available, such as *OpenVPN* [34]. *OpenVPN* is an open-source VPN client and server published under a GPL v2 license [35] that uses its own protocol based on SSL/TLS. Unlike IPsec, which runs in kernel space and can have problems traversing NATs[4], *OpenVPN* runs in user space and has no problems traversing NATs [34, 37].

This approach entails creating a modified *OpenVPN* client that can handle the port-knocking and—if necessary—keep the port open by periodically resending a port-knocking request. This may be necessary in cases where the server’s firewall does not support connection tracking for the transport layer protocol used. This issue is discusses in greater detail in section 3.1.1.

[4] When IPsec is run in Authentication Header (AH) mode, a cryptographic hash covering among other things the IP source address is calculated [36]. As the source address is altered during NAT traversal, the hash verification performed by the destination will fail [37].

#### 2.2.3.1 Functioning

A normal, unmodified *OpenVPN* server is running on the server hosting the service. The modified *OpenVPN* client is started before the application attempts to connect to the service. It uses the *sKnock* library to send a port-knocking request for the port on which the *OpenVPN* server is listening[5] and establishes a VPN tunnel. Once the tunnel is established, the application connects to the service using the IP address of the server's VPN interface instead of the public IP address used to connect the *OpenVPN* client. The port number of the service remains unaffected by the use of this solution.

Going one step further, it is also possible to use *iptables* to transparently redirect traffic addressed to the server's public IP to its VPN IP address. This would facilitate the use of this solution for the end-user and make it completely transparent to the application. This feature is described in more detail in sections 3.1.3 and 3.3.4.

Since requests arriving from a user employing this solution will be addressed to the *OpenVPN* tun/tap device's IP address, the service may not be bound exclusively to the server's public IP address(es), but must also accept requests addressed to *OpenVPN*'s tun/tap interface. A tap interface's address can be configured manually or obtained from a DHCP server on the server's network, while a tun interface always has the lowest possible address in the pool allocated to *OpenVPN*'s pseudo-DHCP. Furthermore, the tun/tap interface must not be subject to the port-knocking, for applications to be able to connect to this interface without port-knocking. Since this interface is only addressable after having connected to the *OpenVPN* server, this does not impact the security.

#### 2.2.3.2 Communication Overhead

The per-packet overhead incurred when sending a packet through an *OpenVPN* tunnel depends on several factors, such as:

- Whether *OpenVPN* is running in Routing or Bridging mode
- Whether the tunnel uses IPv4 or IPv6
- Whether the tunnel uses TCP or UDP
- The cipher used
- The MAC algorithm used
- Whether SSL/TLS or a Preshared Key (PSK) is used

*OpenVPN* can operate either in Routing or Bridging mode. Routing mode operates on layer 3, forwarding the entire layer 3 packet through the tunnel [37] thereby incurring

[5] By default, this is port 1194 [17] but can be configured to be any other port.

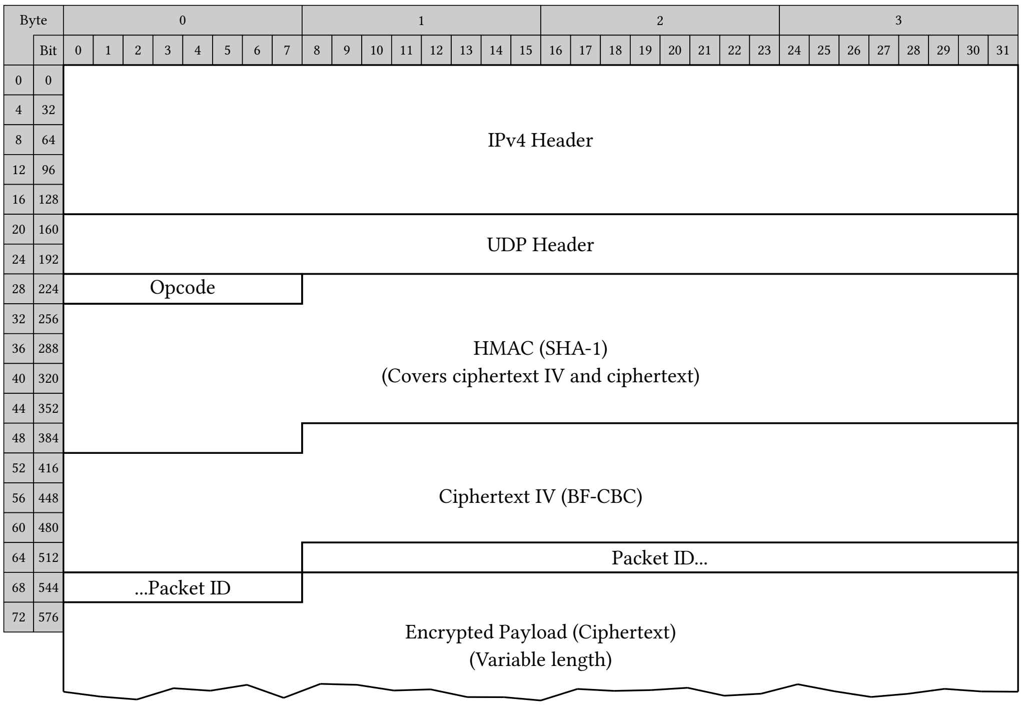

Figure 2.3: A VPN (over UDP) tunnel packet using SSL/TLS

the overhead of additional headers depending on how the Tunnel connection is established. Bridging mode on the other hand operates at layer 2, forwarding the entire layer 2 frame [37] and as such has all of the overhead of Routing mode plus an additional Ethernet header. For our needs, Routing mode is sufficient.

The tunnel can either be run over IPv4 or IPv6. This affects the per-packet overhead because of the different size of the additional header. An IPv4 header is 20 Bytes long [22], while an IPv6 header is 40 Bytes long [23]. *OpenVPN* also supports running the tunnel over either TCP or UDP. UDP is usually preferred because of its lower overhead due to not being connection oriented and having a smaller overhead. Using UDP also avoids potential TCP-over-TCP scenarios. As such it is the default. When using TCP, 20 Bytes for the TCP header [24] are added to the overhead, while UDP adds 8 Bytes for its header [38]. The *OpenVPN* also requires an additional 2 Byte field in its own header when using TCP, which is not necessary with UDP.

Figure 2.3 shows a VPN data packet over IPv4 using SSL/TLS and the default settings[6]. The *OpenVPN* header itself has several fields whose size depend on tunnel settings, such as the cipher and MAC algorithm used. The first field is a 2 Byte field for the packet length, which is only present when tunneling over TCP and thus not present in figure 2.3. The next field is also a conditional field present only when using SSL/TLS cryptography and is a combined 1 Byte field. It is composed of 5 bit message type and a

[6] UDP as the transport protocol, BF-CBC as the cipher and SHA-1 for HMACs.

3 bit key ID.

The next two fields are the HMAC over the entire rest of the packet and an Initialization Vector (IV) for the ciphertext. The length of both of these depends on the selected methods. For SHA-1 and BF-CBC, the HMAC is 20 Bytes long and the IV is 16 Bytes long. Next is the packet ID, which is 4 Bytes long when using SSL/TLS and 8 Bytes long when using a PSK. The last field is the payload, meaning the package that is being sent through the tunnel. Both the packet ID and the payload are encrypted [39, 40].

Since this approach uses SSL/TLS with the same certificates used by *sKnock* for port-knocking, that leaves us with a per-packet overhead of 69 Bytes[7] when tunneling over UDP (preferred) or 83 Bytes[8] when forced to tunnel over TCP, assuming the default cipher and HMAC are used. The values for IPv6 instead of IPv4 are each 20 Bytes higher [39, 40].

Additional overhead arises from the establishment of SSL/TLS sessions and regular key re-exchange, as well as control messages. Since SSL/TLS requires a reliable communication channel, which is not given when tunneling over UDP, *OpenVPN* has an internal ACK mechanism to confirm the receipt of control messages. This mechanism is only used for control messages. Tunneled packets are not subject to this overhead [39, 40]. All of this overhead is independent from traffic sent through the tunnel and it is so low that it can be ignored.

Setting up an *OpenVPN* server is easily done and requires little effort from the service's operator. The only difficulty is configuring the service(s) to serve requests addressed to *OpenVPN*'s tun/tap interface. A client configuration profile can also be created and made available to the end-user by the service's operator to make the use of this solution easier and more comfortable for the user.

#### 2.2.3.3 Advantages & Disadvantages

*Advantages*—This approach has many advantages, one of which is the native support for X.509 certificates. This avoids the need for multiple certificate sets using different standards or using another form of authentication additionally to the certificates. As a result, this solution is both easier to maintain for the service's operator and more comfortable to use for the end-user.

This solution also benefits from a time-proven security scheme. Both *OpenVPN* and SSL/TLS have been around for many years and have been extensively analyzed. As a result, they have incorporated feedback and improved their security over the years.

[7] 69B = 20B (IPv4) + 8B (UDP) + 1B (opcode/key ID) + 20B (SHA-1 HMAC) + 16B (BF-CBC IV) + 4B (packet ID)

[8] 83B = 20B (IPv4) + 20B (UDP) + 2B (packet length) + 1B (opcode/key ID) + 20B (SHA-1 HMAC) + 16B (BF-CBC IV) + 4B (packet ID)

Having been around for this long also has led to administrators and even end-users, in the case of SSL/TLS, to know about these technologies and trust them, thereby facilitating the acceptance of this solution.

*OpenVPN* runs over UDP by default, thus having less overhead than TCP based approaches and avoiding possible TCP-over-TCP situations and the potential performance-related pitfalls thereof, such as the infamous *TCP-Meltdown* described as a disadvantage of the SSH tunnel approach in section 2.2.2.3. Nevertheless, *OpenVPN* allows to tunnel over TCP at the service operator's discretion. The operator can even configure the *OpenVPN* server to accept both TCP and UDP tunnels and leave the choice to the user. This can be useful to allow the use of UDP when possible but leave the ability to fall back to TCP when UDP is unavailable intact.

Another advantage is the ability to redirect traffic destined to the service's public IP address through the tunnel, thus making this approach completely transparent to the application. This increases the comfort for the end-user as they only need to launch the modified *OpenVPN* client to establish the connection and then use the application as if the service was not hidden behind port-knocking.

Since this approach mostly relies on widespread software, a user who does not have access to the modified version of *OpenVPN* due to restrictive company policies, for example, can still use *sKnock*'s reference client to manually perform the port-knocking and then establish a tunnel with an unmodified version of *OpenVPN*. While this is certainly less comfortable than using the modified *OpenVPN* client and does not provide transparent traffic redirection for the service, it can at least allow access to the service even without the modified client.

One last aspect needs to be mentioned, though it can be viewed either as an advantage or disadvantage, depending on the viewpoint. Once the *OpenVPN* tunnel is established, all of the server's ports are accessible through the tunnel without the need for further authentication or port-knocking. While this greatly facilitates the use of applications using multiple ports or random ports, which are not yet known when establishing the tunnel, it can also be viewed as a security risk. The *sKnock* library allows to encode a restrictive list of allowed ports into the certificates [3]. This additional security feature is completely circumvented when using this approach.

*Disadvantages*—This approach only has two notable disadvantages, one of which is the aforementioned opening of all of the server's ports to the user. As has already been stated while enumerating the numerous advantages of this approach, this can however also be viewed as an advantage.

The other disadvantage is the overhead. Of the three approaches presented in this thesis, this one has the highest overhead. This being said, the per-packet overhead of this solution is only 11 Bytes higher than that of the SSH tunnel solution. Furthermore, the impact of this overhead may vary depending on the type of traffic the service generates. Traffic with fewer but larger packets will be less affected than traffic with many smaller packets.

## 2.3 Conclusion

The fact that the SSH tunnel approach only works for services that exclusively use TCP greatly restricts its applicability. The risk of issues resulting from TCP-over-TCP further increase an operator's concerns about using such a solution. Finally, the lacking support for X.509 certificates and the necessity of opening one tunnel per port further reduce the comfort of using it, giving an already unattractive solution the killing blow.

The proxy approach, while seeming a lot more attractive and having a much lower expected overhead, also has one severe disadvantage. The requirement for the client to support the proxy protocol, be it SOCKS5 or a newly designed one results in the possible need to modify the client software to add that support. This is a lot of work and can even be impossible if the client is neither open-source nor developed by the service operator. Most service operators are likely to recoil from a solution requiring such drastic measures.

This leaves only the VPN tunnel. Luckily, the *OpenVPN* solution has few disadvantages, the most notable of which is having the highest overhead of these three approaches. However, this overhead is still not overly high and the VPN approach offers several advantages to compensate. Requiring the least effort to implement from both, the operator and the user, it also happens to be the least invasive approach. Add to that the ability to seamlessly integrate *sKnock* with its EC Cryptography and certificates, this solution becomes the clear winner.

# Chapter 3

# Implementation

This chapter explains the actual implementation of our modified *OpenVPN* client. It describes some general considerations and design decisions that need to be made before giving a simplified description of how *OpenVPN* works. We then explain sample configuration files for the server and the client before we finish with some limitations of our implementation.

## 3.1 General Considerations

Once the *OpenVPN* approach has been selected, there are still a few general issues to consider. One of the advantages of *OpenVPN* is that it leaves its user the choice between TCP and UDP with UDP being the default. This however also means more work for us, as TCP and UDP need to be handled differently in the code. Furthermore, UDP as a stateless protocol may be problematic because of possibly lacking support for connection tracking of stateless protocols in the server's firewall. This eventuality should also be dealt with. Lastly, if we wish to transparently redirect traffic to the service's server through the tunnel, we need to decide on how to achieve that.

### 3.1.1 Connection Tracking and UDP

Like most port-knocking solutions, *sKnock* only opens a port temporarily, closing it again after a short, configurable timeout. To avoid severing the new connections that were established while the port was open, the firewall is configured to allow already established connections to pass even though new connections are blocked.

This requires to track the state of connections. Connections can be identified by at tuple containing their (protocol, source IP, destination IP, source port, destination port) [41]. A firewall that supports connection tracking saves such a tuple and a state associated with the connection whenever a new connection is established [42]. A tuple is removed when the connection it identifies is terminated.

With TCP, this connection tracking is easily done, as TCP is a connection oriented protocol. As such, it performs a handshake with its connection partner to establish a connection and sends a termination message to close the connection.

UDP, however, is a stateless protocol and as such does not have any notion of connection. This makes it harder to detect when a connection is established or closed. Luckily, most modern firewalls are able to track "pseudo connections" in UDP. This works by checking every received packet against a list of tracked connections. If no entry exists yet, a new entry is created with a timeout. If an entry exists, its timeout is refreshed. Entries whose timeout runs out are removed [42,43]. This approach can also be used with other stateless protocols.

While most modern firewalls support this approach, this cannot be assumed to hold true for all firewalls. Thus, for greatest possible interoperability, a mechanism needs to be incorporated into our modified *OpenVPN* version that allows to periodically resend a port-knocking request to keep the port open. The usage of this mechanism should be optional such as to not unnecessarily weaken security by keeping the port open. Furthermore, the interval at which the requests are resent should be configurable such as to adapt to the timeout after which the server's *sKnock* daemon is configured to close the port again.

It would be advisable for service operators whose firewall does not support connection tracking for stateless protocols to not configure a too low value for the duration for which the port stays open, in order to leave some leeway for the client to resend a port-knocking packet using this mechanism before the port closes again.

### 3.1.2 Differences in TCP/UDP Handling

Differences in the way UDP and TCP work affect the decision about when to perform the port-knocking. In the case of TCP, the most logical choice would be to replace the call to the built-in `connect(socket, *address, address_len)` function with a call to a function provided by us. This function would perform the port-knocking and attempt to connect. It would then ensure that the connection has been successfully established. In the event of a failure it would retry the process several times before giving up.

This ensures the shortest possible delay between the opening of the port and the establishment of the connection, thereby allowing for extremely low values for the duration for which the port stays open to be configured for the server's *sKnock* daemon. Additionally, this ensures that a connection attempt does not fail because of a dropped port-knocking packet.

With UDP, however, there is no explicit connection function, since, as a stateless protocol it does not have any notion of connections. Furthermore, there is no way of testing whether a port-knocking request has failed, since it is not possible to attempt to connect and check for the success of the operation.

The best we can do here is to perform the port-knocking right after the socket is created and assume that the port-knocking was successful. In the event of a failed port-knocking request, *OpenVPN* will fail to connect to the server without knowing why and may retry the entire connection sequence. While not as elegant as the TCP solution, this does at least ensure that the port-knocking packet has been sent before the program continues execution and attempt to actually communicate with the server.

The entire process of establishing an *OpenVPN* tunnel usually only takes a few seconds. Even though the delay between port-knocking and actual communication and thus registration in the connection tracking table is potentially higher than with the TCP solution, this should therefore rarely be an issue. Only setting values lower than those few seconds for the duration for which the *sKnock* daemon on the server keeps the port open could cause problems.

### 3.1.3 Transparent Traffic Redirection

In order to make this solution easy to use for the end-user, we want an application accessing a port-knocked service to be useable as if the service was not port-knocked. This means that it should not be necessary to edit the application's settings.

Even once the tunnel is established, the port-knocking still protects every port on the server. It is thus not possible to simply access the service with the server's public IP address, as the service's port is still closed. However, if the service is listening on the server's tun/tap interface, it could be addressed through the server's VPN IP address. While this would ensure that the request to the service is sent through the VPN tunnel to the tun/tap interface, this would mean a different usage of the application than without a tunnel. The tun/tap interface must not subject to the port-knocking for this to work, for a client to be able to connect to this interface without performing the port-knocking itself.

One might be tempted to add an additional modification to the *OpenVPN* that edits the client's routing table in such a way as to route traffic to the server's public IP through the tunnel with the server's VPN IP as the gateway. One would then activate IP forwarding on the server to have the traffic sent from the client to the server's public IP address but arriving through the tun/tap interface forwarded to its public IP address. This approach is, however, flawed, as VPN packets themselves, which contain the encapsulated actual packets would then be routed through the tunnel. This means they would be encapsulated into yet another packet, which would also be routed through the tunnel, thereby creating an endless loop and causing the tunnel to collapse. The VPN packets themselves thus need to be able to reach the server by its public IP address. Moreover, the traffic being forwarded to the server's public IP would again be subject to the port-knocking.

This makes it necessary to exclude traffic from the *OpenVPN* client to the VPN port from this redirection. This is, however, impossible to achieve with routing tables. Thus, we decided to use *iptables*' NAT functionality to rewrite source and destination IP addresses of packets generated on the client instead of tampering with its routing table.

*iptables*' NAT table has three chains: `PREROUTING`, `OUTPUT`, and `POSTROUTING`. The first chain to be processed is the `PREROUTING` chain. It is processed before any routing decisions are made and only applies to packets arriving to the machine, while the `POSTROUTING` chain is the last one to be processed, after all routing decisions have been made and applies to every packet leaving the machine. Source NAT (SNAT) replaces the source address of a packet with another one and is usually done in the `POSTROUTING` chain. The destination address on the other hand is substituted by Destination NAT (DNAT), which is usually done in the `PREROUTING` chain [44]. The last chain, `OUTPUT` has a similar purpose to the `PREROUTING` chain, except it applies to packets which are generated on the machine, instead of being received by the machine [45].

Since the destination IP address can influence the routing decision and, in our case, determines whether the packet is routed through the normal interface or the VPN interface, the destination IP of packets sent by the client to the server's public IP address has to be substituted before these routing decisions are made. This means we have to match packets addressed to the server's public IP address in the client's `OUTPUT` chain and send them to the `DNAT` target.

In order for the server to be able to reply to the application, the source address of packets sent from the client to the server's public IP address also has to be replaced with the client's VPN address. In our case, this can only be done in the client's `POSTROUTING` chain, as the `OUTPUT` chain does not support the `SNAT` target and packets being generated on the local machine do not traverse the `PREROUTING` chain. This is, however, the chain in which SNAT is usually used and does not pose any problems for our setup. Since the destination address of a response is the source address of the request, the response's destination address needs to be changes back to the original source address of the request, meaning the source address it had before being subjected to SNAT. The SNAT target automatically takes care of this replacement too.

Since DNAT is performed in the `OUTPUT` chain and thus before SNAT, the destination address will already have been changed to the server's VPN internal IP address. This means the DNAT rule has to match against the server's public IP address while the SNAT rule has to match against the server's VPN IP address. Putting it all that together, we get the rules shown in listing 3.1.

Listing 3.1: *IPTables* rules

```
iptables -A OUTPUT -d <server public IP> -j DNAT --to <server VPN IP>
iptables -A POSTROUTING -d <server VPN IP> -j SNAT --to <client VPN IP>
```

However, we still need to exclude the VPN packets themselves from these rules. Because the `ACCEPT` target ends the traversal of a chain, we can simply prepend a rule that specifically matches the VPN tunnel and use the `ACCEPT` target on it, thereby preventing it from matching further rules in the table.

Furthermore, because we do not know what rules may already exist in the NAT table and some targets—such as the above mentioned `ACCEPT` target—stop the traversal of a chain, we want to ensure that our NAT rules are processed before any rule already present. To do that, we simply insert our rules at the beginning of the NAT table. This results in the improved rules as shown in listing 3.2.

Listing 3.2: *IPTables* rules

```
iptables -I OUTPUT -d <server public IP> -j DNAT --to <server VPN IP>
iptables -I OUTPUT -d <server public IP> -p <TCP / UDP> --port <VPN port> -j ACCEPT
iptables -I POSTROUTING -d <server VPN IP> -j SNAT --to <client VPN IP>
```

Note that the rules are now inserted (`-I`) instead of appended (`-A`). Therefore, the specific rule for the VPN tunnel needs to be listed after the DNAT rule, as the last inserted rule will be the first one to be processed.

Our modified *OpenVPN* client will be able to automatically determine the necessary IP addresses and ports and add these rules to *iptables* once the tunnel is established. These rules will automatically be removed when the tunnel closes.

Note that *iptables* is unable to cross between IPv4 and IPv6. This means that this feature will only work for services addressed with IPv6 if *OpenVPN* is configured to assign IPv6 addresses inside the tunnel.

## 3.2 Overview of OpenVPN Execution Flow

The first thing *OpenVPN* does on startup is preparing a few things that only need to be done once per process, such as initializing the global context. After that, *OpenVPN* proceeds to execute three nested loops: The outer loop, the inner loop and the main event loop.

The outer loop runs once on startup and after that once every time a `SIGHUP` is sent to *OpenVPN*. Its main task is handling the top level initialization and then executing the inner loop. The inner loop in turn selects and calls the appropriate function containing an implementation of the main event loop depending on *OpenVPN*'s operation mode. It runs once on startup, just like the outer loop and after that once every time a SIGUSR1 is sent to *OpenVPN*.

Because there is only one *OpenVPN* binary for both client and server mode, there exist three functions containing an implementation of the main event loop:

- tunnel_point_to_point() for client mode,
- tunnel_server_tcp() for TCP server mode and lastly,
- tunnel_server_udp()[1] for UDP server mode.

As we do not change the server side code, we will not further discuss it here. The tunnel_point_to_point() function first connects to the *OpenVPN* server, authenticates and establishes the tunnel. Once this is done, it runs the main event loop, which does all the actual work from now on until the tunnel is closed. At each pass, it checks its timers to see if a timer based action, such as sending a keepalive ping needs to be performed. It also checks if TLS needs attention before checking for previously received *OpenVPN* control messages and processing them appropriately.

The main event loop then waits for either one of the following events to happen:

- Data can be read from the physical interface
- Data can be read from the tun/tap interface
- Data can be written to the physical interface
- Data can be written to the tun/tap interface
- A signal (such as SIGUSR1 or SIGHUP) has been received
- The timeout has run out

In the event of the timeout having run out, the main event loop simply proceeds to the next iteration. This is done to ensure that the timers are checked sufficiently often. operating system signals sent to *OpenVPN* are also checked in every iteration of the main event loop. SIGHUP and SIGUSR1, cause *OpenVPN* to proceed to the next iteration of the outer or inner loop respectively.

When data can be read from the physical interface, it is read and processed. For channel control messages, an ACK control message is written to the output buffer of the physical interface and the message itself will be handled during the next main event loop iteration. Data messages are processed to extract the original packet which is encapsulated in it and the "de-encapsulated" message is written to the output buffer of the tun/tap interface,

[1] Actually, tunnel_server_udp() does not contain any logic itself. Instead, it simply calls the function tunnel_server_udp_single_threaded(), which contains the actual implementation of this function. This suggests that a multithreaded UDP server mode may be added in the future.

where it will be handled during the next iteration of the main event loop. This processing includes among other things decryption, defragmentation and decompression.

When data can be read from the tun/tap interface, it is processed. This processing includes compression (if activated), encryption (if activated), and encapsulation among other things. The encapsulated packet is then written to the physical interface's output buffer to be handled during the main event loop's next iteration. When there is data in the output buffer of either the physical interface or the tun/tap interface, it is simply sent through the appropriate socket.

*OpenVPN*'s state is separated into several layers, which are reset at different occasions. There are three state levels with the "level 0 state" being initialized once at program startup and then remaining until the process ends. The "level 1 state" is reset with every iteration of the outer loop, but persist over iterations of the inner loop and the main event loop. Lastly, the "level 2 state" is reset with every iteration of the inner loop. The main event loop does not have an own context level.

## 3.3 Modified OpenVPN Client

The following sections explain the changes and additions done to the *OpenVPN* code in order to achieve our goal in more detail. The code is classified by functionality, starting with the initialization of resources needed by the rest of the code and continuing with the port-knocking itself. Finally, additional features, such as periodic refresh of the port-knocking request and transparent traffic redirection using *iptables*' NAT functionality described in sections 3.1.1 and 3.1.3 are explained.

Most of the new code that was added to enable *OpenVPN* to use *sKnock* to perform the port-knocking, as well as additional features mentioned above, is located in the new files `wilfred.c` and `wilfred.h`[2]. Other files have been slightly modified to include calls to functions in `wilfred.c`. These files are `openvpn.c`, `init.c`, `sockets.c` and `forward.c`. Furthermore, the files `options.c`, `options.h` and `openvpn.h` have been modified to expand *OpenVPN*'s state and add new command line options for port-knocking. In order to pass some state information down the port-knocking functions, several function signatures had to be adapted resulting in minor changes to several other files.

All new functionality added by us is entirely optional and is only used when the appropriate options are specified on the command line, in the configuration file or are pushed by the server. This works without modifying the *OpenVPN* server, as it is able to push arbitrary configuration options, as long as the *OpenVPN* client is able to understand

[2] The name for these files is derived from Wilfred Mott, a character from the Doctor Who series, whom The Doctor saves at the cost of his own life. This event is heralded by Wilfred knocking four times.

them. Our modified *OpenVPN* client can therefore still function as a regular *OpenVPN* client, thereby avoiding to have two versions of basically the same application installed for users who also need a regular *OpenVPN* client.

### 3.3.1 Initialization

The initialization process consists of two steps, starting with the initialization of the C wrapper library for *sKnock*. This can actually be done with only a single call to the `knock_init()` function of the *sKnock* library's C wrapper. Calling `knock_init()`, as well as doing some additional checks to ensure the success of the operation is handled by the function `knock_init_lib()` in `wilfred.c`. It also keeps track of the initialization status of the library in a global variable and is able to print status messages or error messages when something goes wrong. Listing 3.3 shows a simplified version of this function, which has been stripped of screen output. First, it checks whether the library has already been initialized before, in which case it immediately ends successfully. If the library has not been initialized yet, the appropriate function from the *sKnock* library is now called and its success stored. Lastly, the success of the initialization is returned.

Listing 3.3: Initialization of the port-knocking library

```
static bool initialized = false;

bool knock_init_lib () {
  if (initialized) {
    return true;
  }

  initialized = knock_init() ? false : true;

  return initialized;
}
```

The second part of the initialization process is the initialization of the port-knocking handle. This is essentially done with a single call to the *sKnock* library's C wrapper function `knock_new(...)`, which requires a few values pertaining to the verification of the port-knocking's success, the path to the server's certificate and the path to a PKCS#12 file containing the client's private key and certificate, as well as the CA's certificate. Since the PKCS#12 file may be password protected, as it contains the client's private key, a password is also required. For PKCS#12 files that are not password protected, it is simply an empty string. Furthermore, it is necessary for the library to have been initialized before this function can be called.

Our modification to the *OpenVPN* code includes the function `knock_init_handle(...)`, which takes a password source and paths to the server's certificate and the client's PKCS#12 files as parameters. A simplified version of this function, which has been stripped of screen output is shown in listing 3.4. The listing also shows a few C preprocessor definitions, which will be explained later. The function first checks if the library has previously been initialized and aborts with a failure if not.

Next, the password of the PKCS#12 file has to be retrieved. *OpenVPN* already supports PKCS#12 containers and thus already provides methods of retrieving a password, which we can use. To avoid asking for the password twice, we first check if the password has already been queried by *OpenVPN* and is still cached. To access this information, the new function `get_passbuf()` is added to `ssl.c`. It simply returns a pointer to *OpenVPN*'s password cache. If no password is saved in the cache, *OpenVPN*'s `pem_password_setup(...)` function is called. Its task is to retrieve the password from the password source `key_pass_file`, which may either be a path to a file or `stdin`. This source can be passed as a parameter (`--askpass [file]`) when starting *OpenVPN*. `pem_password_setup(...)` stores the password in the password buffer to which we now have a pointer.

Because `knock_new(...)` requires a password string even for certificates without password, a dummy password consisting of an empty string is created and the usual password retrieval skipped for certificates without password.

Should `pem_password_setup(...)` fail, a local password buffer is created and the password is manually queried form the password source and saved into this buffer. The existence of a password is then checked one last time and if still no password could be retrieved, the function fails. If, however, a password could be retrieved by either of these methods, `knock_new(...)` from *sKnock*'s C wrapper is called with the appropriate paths, the password and information on the desired port-knocking success verification. If the allocation of a local password buffer was necessary, it is now freed before the success of the handle initialization is returned.

Listing 3.4: Initialization of the port-knocking handle

```
#define KNOCK_TIMEOUT 5
#define KNOCK_ATTEMPTS 1
#define KNOCK_VERIFY 0

static bool initialized = false;
static struct KNOCK_Handle *knock_handle = NULL;

bool knock_init_handle (const char* key_pass_file,
                        const char* cert_srv_file,
                        const char* pkcs12_client_file) {
  struct user_pass *pass;
  bool free_pass = false;

  if (!initialized) {
      return false;
  }

  if (nopass) {
    /* Use an empty string as password instead of querying one if the
     * certificate does not have a password */
    pass = malloc(sizeof(struct user_pass));
    memset(pass, 0, sizeof(struct user_pass));
    free_pass = true;
  } else {
    /* Retrieve the PKCS#12 password */
    pass = (struct user_pass *) get_passbuf();
    if (!strlen (pass->password)) {
      pem_password_setup (key_pass_file);
    }
    if (!strlen (pass->password)) {
      /* Manually query the password, if the normal way fails */
      pass = malloc(sizeof(struct user_pass));
      free_pass = true;
      get_user_pass (pass,
                     key_pass_file,
                     UP_TYPE_PRIVATE_KEY,
                     GET_USER_PASS_MANAGEMENT|GET_USER_PASS_PASSWORD_ONLY);
    }
    if (!strlen (pass->password)) {
      return false;
    }
  }

  /* Create the port-knocking handle */
  knock_handle = knock_new(KNOCK_TIMEOUT,
                           KNOCK_ATTEMPTS,
                           KNOCK_VERIFY,
                           cert_srv_file,
                           pkcs12_client_file,
                           pass->password);

  /* Free the memory if we had to get user_pass ourselves */
  if (free_pass) {
    free(pass);
  }

  /* Check whether the handle could be successfully initialized */
  if (knock_handle == NULL) {
    return false;
  }

  return true;
}
```

As has already been hinted, the *sKnock* library offers the option to verify the success of the port-knocking attempt and—if necessary—retry the port-knocking. This is done by attempting to open a connection to the port on the server. If the connection is refused or cannot be established before a timeout, the port-knocking is deemed unsuccessful and another attempt is made until a maximum number of attempts is reached. Due to the nature of this verification, it only works for TCP ports, since UDP as a stateless protocol does not have a concept of connections.

This behavior can be configured through the `knock_new(...)` library call's first three parameters, the first parameter being the timeout, the second one the maximum number of attempts and the third one allowing to enable or disable this functionality. We wish to verify the success of the port-knocking operation ourselves, as this allows us to directly use the socket for *OpenVPN* if the port-knocking was successful. Therefore, we disable this functionality.

Calls to these initialization functions are added to the level 1 initialization in the outer loop, which is implemented in `openvpn_main(...)` in `init.c`. An excerpt of the modified version of this function is shown in listing 3.5. Port-Knocking is only initialized if it was enabled by passing the option `--knock` to *OpenVPN* on the command line or in a configuration file.

Listing 3.5: Excerpt of modified `openvpn_main(...)` in `init.c`

```
static int openvpn_main (int argc, char *argv[]) {
  ...

  /* Init the port-knocking library if necessary */
  if (c.options.knock) {
    if (!knock_init_lib())
      break;
    if (!knock_init_handle(c.options.key_pass_file,
                           c.options.knock_cert_srv_file,
                           c.options.pkcs12_file,
                           c.options.knock_nopass))
      break;
  }

  ...
  return 0;
}
```

### 3.3.2 Port-Knocking

The port-knocking itself is handled with the *sKnock* library's `knock_knock(...)` function, which requires a port-knocking handle, the *OpenVPN* server's address, port and transport layer protocol[3]. The port-knocking handle is part of the C wrapper for *sKnock* and mirrors a `ClientInterface` object in from *sKnock*'s Python code.

The Port-knocking in our modified *OpenVPN* client is mainly handled by the function `knock_do_knock(...)`, which takes a file descriptor of the socket for which to port-knock and the *OpenVPN* server's IP address and port. This function first checks whether the library and a port-knocking handle have been initialized before continuing. If either have not been properly initialized, the function fails immediately. If everything is in order, the function proceeds with determining whether *OpenVPN* is running over TCP or UDP by querying the socket. Since this yields us the protocol IDs as specified in the respective RFCs[4] but *sKnock* uses different IDs, we need to "translate" this information. Once this is done, our code calls the *sKnock* library's `knock_knock(...)` function with the address and port of the *OpenVPN* server (supplied as parameters

[3] Only TCP and UDP are supported at this moment.

[4] TCP has protocol ID 6 and UDP has protocol ID 17

to knock_do_knock(...)) and the transport layer protocol determined before. The knock_knock(...) function's return value is only relevant when using *sKnock*'s success verification feature, which we do not, thus it is unnecessary to catch this return value. Listing 3.6 shows a simplified version of this function, which has been stripped of screen output.

Listing 3.6: Simplified version of knock_do_knock(...) in wilfred.c

```
static bool initialized = false;
static struct KNOCK_Handle *knock_handle;

bool knock_do_knock (socket_descriptor_t sd, char *addr_str, int port) {
  int proto;
  socklen_t len;

  /* Abort if not ready to do port-knocking */
  if ((!initialized) || (knock_handle == NULL)) {
    return false;
  }

  /* Determine what protocol to knock on */
  len = sizeof(int);
  getsockopt(sd, SOL_SOCKET, SO_PROTOCOL, &proto, &len);
  if ((proto == IPPROTO_IP) || (IPPROTO_TCP)) {
    proto = KNOCK_TCP;
  } else if (proto == IPPROTO_UDP) {
    proto = KNOCK_UDP;
  }

  knock_knock(knock_handle, addr_str, port, proto);
}
```

This function is accompanied by a few helper functions, which are able to receive information about the *OpenVPN* server's address and port in a different format and pass it on to the knock_do_knock(...) function. knock_do_knock_sockaddr(...) takes a sockaddr structure as its parameter instead of an IP address and port. These are instead extracted from the sockaddr structure. This is done with to two other helper functions, knock_get_params(...), which extracts the information and stores it in a knock_params structure, and knock_do_knock_struct(...), which uses the IP address and port stored in a knock_params structure.

Another important function is the knock_connect(...) function. Its task is to establish a TCP connection and it is intended as a drop-in replacement for the built-in connect() function in C. As such, it requires a socket descriptor and a sockaddr structure, from which to extract the server's address and port, as parameters. It handles the port-knocking and the verification of its success. In the event of a failure, it also handles retrying the process.

The knock_connect(...) function first extracts the *OpenVPN* server's address and port in an IPv6 aware fashion using the knock_get_params(...) helper function and stores those values in a knock_params structure for reuse, in case another port-knocking attempt is necessary. Should this operation fail, the function will immediately abort. The `knock_params` structure is defined in `wilfred.h` and merely contains the IP address and the port. Next, the port-knocking is performed using `knock_do_knock_struct(...)`, which accepts a `knock_params` structure as a parameter. `knock_connect(...)` then attempts to establish a TCP connection. If the return code of this connection attempt indicates a connection refusal or a timeout, it assumes that the port-knocking was unsuccessful and another attempt is made, reusing the address and port extracted earlier. This is done up to four times[5]. A `connect(...)` return code stating that the connection is in progress indicates that the socket is in non-blocking mode. In this case, we are unable to verify the success of the port-knocking and do not repeat the port-knocking. Lastly, the return code of the last `connect(...)` call is returned, as *OpenVPN* performs its own checks on this value. The listing 3.7 shows a simplified version of this function, again stripped of screen output.

Listing 3.7: Simplified version of `knock_connect(...)` in `wilfred.c`

```
static struct KNOCK_Handle *knock_handle = NULL;

int knock_connect (socket_descriptor_t sd, const struct sockaddr *remote) {
  struct knock_params params;
  int status;
  int i;

  if (!knock_get_params(&params, remote)) {
      return false;
  }

  /* Try to knock the port and connect. Try up to four times */
  for (i = 0; i < 4; i++) {
    knock_do_knock_struct(sd, &params);

    status = connect(sd, remote, af_addr_size(remote->sa_family));
    if (status) {
      status = openvpn_errno ();

      /* Only retry if the connection was refused (REJECT) of timed out (DROP) */
      if ((status == ECONNREFUSED) || (status == ETIMEDOUT)) {
        continue;
      }
      /* When the socket is in non-blocking mode, we cannot verify the success */
      if (status == EINPROGRESS) {
        break;
      }
    }

    /* If the connection was successfully established or failed for another
     * reason than a timeout or refused, do not retry */
    break;
  }

  return status;
}
```

[5] Another wink at the Doctor Who series. The 10th Doctor's death is heralded by Wilfred knocking four times. Any other number could be used here, though values above five are probably useless.

As has been explained in section 3.1.2, when running the tunnel over UDP, the port-knocking is done directly after the socket is created. Sockets are created by the function `create_socket(...)` in `socket.c`. The relevant excerpt of this function's modified version is shown in listing 3.8.

Listing 3.8: Excerpt of the modified `create_socket(...)` in `socket.c`

```
static void create_socket (struct link_socket* sock,
                           struct addrinfo* addr,
                           struct context *c) {
  if (addr->ai_protocol == IPPROTO_UDP || addr->ai_socktype == SOCK_DGRAM) {
    sock->sd = create_socket_udp (addr, sock->sockflags);
    sock->sockflags |= SF_GETADDRINFO_DGRAM;

    /* Do port-knocking. */
    if (c->options.knock) {
      knock_do_knock_sockaddr(sock->sd, addr->ai_addr);
    }

    ...
  }
  ...
}
```

When running the tunnel over TCP, the port-knocking is done during connection establishment instead. This happens in `openvpn_connect(...)`, which is in the same file. The calls to the built-in C function `connect(...)` need to be replaced by calls to `knock_connect(...)`. Because of C preprocessor #ifdef commands differentiating between versions compiled with and without non-blocking socket use, there are actually two calls that need to be replaced. Listing 3.9 shows the relevant excerpts of this function's modified version. Since both calls to `connect(...)` are identical and as such replaced by identical calls to `knock_connect(...)`, for simplicity's sake the listing contains only one call.

Listing 3.9: Excerpt of the modified `openvpn_connect(...)` in `socket.c`

```
int openvpn_connect (socket_descriptor_t sd,
                     const struct sockaddr *remote,
                     int connect_timeout,
                     volatile int *signal_received,
                     struct context *c) {
  int status = 0;

  if (c->options.knock) {
    status = knock_connect (sd, remote);
  } else {
    status = connect (sd, remote, af_addr_size(remote->sa_family));
    if (status)
      status = openvpn_errno ();
  }

  ...
}
```

To enable port-knocking, the option `--knock` has to be passed to *OpenVPN* either on the command line, or in a configuration file. When this option is not passed, the modified *OpenVPN* client behaves exactly like a normal client.

### 3.3.3 Port-Knocking Keepalive Refresh

As has been mentioned in section 3.1.1, the server's firewall may not support connection tracking, especially for stateless transport layer protocols, such as UDP. Since the *sKnock* daemon closes the port after a configurable timeout, this would cause the tunnel to be severed shortly after that happens. To prevent this, a mechanism has to be implemented to reiterate the port-knocking request. This would cause the port to be reopened before OpenVPN has a timeout or—in future implementations of *sKnock*—simply keep the port open.

Because this periodic resending of the port-knocking request is not always necessary and constitutes an additional overhead, this feature is entirely optional. It can be activated through the new option `--knock-refresh [n]`, where `n` specifies the interval in seconds at which the port-knocking request is to be resent. It defaults to 10 when not explicitly specified. This option can also be pushed by the server, allowing the service's operator to ensure that the user's modified *OpenVPN* client can maintain a connection despite the firewall not supporting connection tracking for the selected transport layer protocol. If this option is unnecessarily used with a server whose firewall supports connection tracking for the selected transport layer protocol, the option will not have any negative effects beyond the unnecessary overhead of periodically sending an additional packet.

*OpenVPN*'s main loop checks for timer based events on every iteration. To do this, *OpenVPN* stores an `event_timeout` structure for each of these timer based events in its level 2 state[6] Listing 3.10 shows the definition of this structure, which can be found in `interval.h`. `n` stores the interval at which this event is to be triggered in seconds, `last` stores the last time at which the event was triggered and `defined` indicates whether or not this timer is active. Timers which are not defined are ignored. We can thus add such a timer for our port-knocking refresh to the level 2 state.

Listing 3.10: `struct event_timeout` definition in `interval.h`

```c
struct event_timeout
{
  bool defined;
  interval_t n;
  time_t last; /* time of last event */
};
```

Each of these structures needs to be initialized before it can be used. This is done

[6] The level 2 state contains state information that is reset at every iteration of the inner loop. This is described in section 3.2.

with the `event_timeout_init(...)` function. *OpenVPN* initializes all timers in its `do_init_timers(...)` function, which is called during level 2 initialization. This also means that all these timers are reset whenever *OpenVPN*'s inner loop reiterates. We can simply add the initialization of our new timer to this function, as shown in listing 3.11.

Listing 3.11: Excerpt from `do_init_timers(...)` in `init.c`

```
static void do_init_timers (struct context *c, bool deferred) {
  ...

  if (c->options.knock && c->options.knock_refresh_timeout)
    event_timeout_init (&c->c2.knock_refresh_interval,
                        c->options.knock_refresh_timeout,
                        0);
}
```

The main event loop calls a series of functions which eventually lead to a call to `process_coarse_timers(...)`. This function calls a check function for each timer one after another. The check function does nothing more than call `event_timeout_trigger`, which checks whether the timer has run out and updates it if necessary. If the timer has run out, then an appropriate action is taken. Listing 3.12 shows an excerpt of the modified `process_coarse_timers(...)` containing a call to our own check function, which is shown in 3.13.

Listing 3.12: Excerpt from `process_coarse_timers(...)` in `forward.c`

```
static void process_coarse_timers (struct context *c) {
  ...

  /* Does the port-knocking request have to be refreshed? */
  knock_check_refresh (c);
}
```

Listing 3.13: `knock_check_refresh(...)` and `knock_do_refresh(...)` in `wilfred.c`

```
void knock_do_refresh (struct context *c) {
  knock_do_knock_sockaddr(c->c2.link_socket->sd,
                          c->c2.link_socket->info.lsa->current_remote->ai_addr);
}

inline void knock_check_refresh (struct context *c) {
  if (c->options.knock
      && c->options.knock_refresh_timeout
      && event_timeout_trigger (&c->c2.knock_refresh_interval,
                                &c->c2.timeval,
                                ETT_DEFAULT)) {
    knock_do_refresh (c);
  }
}
```

| **IPv6 Tunneling** | Off | | On | |
|---|---|---|---|---|
| | Outer Addr. | Inner Addr. | Outer Addr. | Inner Addr. |
| **IPv4** | ✓ | ✓ | ✓ | ✓ |
| **IPv6** | ✗ | ✗ | ✗ | ✓ |

Table 3.1: Allocation of inner IPv4 and IPv6 addresses when tunneling over IPv4

| **IPv6 Tunneling** | Off | | On | |
|---|---|---|---|---|
| | Outer Addr. | Inner Addr. | Outer Addr. | Inner Addr. |
| **IPv4** | ✗ | ✓ | ✗ | ✓ |
| **IPv6** | ✓ | ✗ | ✓ | ✓ |

Table 3.2: Allocation of inner IPv4 and IPv6 addresses when tunneling over IPv6

### 3.3.4 Transparent Traffic Redirection Using NAT

To make this solution more comfortable and easier to use for the user, we want to offer an option to transparently redirect traffic to the service's public address through the tunnel, as explained in section 3.1.3. For the purposes of this Proof-of-Concept implementation, we decided to use *iptables*' NAT functionalities with the equivalent of the rules shown in listing 3.2. This feature can be enabled through the new option `--knock-nat` and is entirely optional. Its use requires *iptables* to be installed system running the modified *OpenVPN* client.

*OpenVPN* fully supports IPv6. This means that it can establish a tunnel over an IPv6 connection, as well as allow IPv6 traffic to pass through the tunnel. While this means that IPv6 packets can be sent through the tunnel, this feature is disabled by default. However, the server may push configuration options to the client enabling this. In any case, *OpenVPN* will allocate IPv4 addresses inside the tunnel, but IPv6 addresses will only be allocated when IPv6 tunneling is enabled. Thanks to this behavior, it is always possible to redirect from a public IPv4 address to an internal IPv4 address, as can be seen in table 3.1. When tunneling over IPv6, however, it is not possible to redirect to an *OpenVPN* internal IPv4 address. Thus, redirection can only work if IPv6 tunneling is enabled. This is shown in table 3.2.

There are mainly three methods of programmatically interacting with *iptables*: Through the `libiptc` library, by repeatedly calling the `iptables` binary, or by opening a pipe into `stdin` of the `iptables-restore` binary. According to the official *iptables* FAQ, the `libiptc` library was never intended for public use. It is an internal interface used by the various *iptables* binaries. As such it is rather unstable and its use is not recommended. Instead, it is recommended to call one of the aforementioned binaries [46].

While hardcoding a call to a binary is generally not a good practice as file paths or binary versions may not be the same across systems, using an unstable library, which may also differ from system to system is not a good idea either. Since both methods have portability issues, we decided to go with the recommended approach and call one of the binaries. This approach also has the advantage of requiring significantly less code, which should be easier to maintain. We chose to open a pipe into `stdin` of `iptables-restore`, as this allows us to execute several commands while only using one call to a binary, whereas three calls to a binary would be necessary when using `iptables`. It is important to call `iptables-restore` with the `-n` or `--noflush` parameter to prevent it from removing all previously installed rules.

Once the rules have been added to *iptables*, all relevant information needs to be stored in order to be able to remove the rules when the tunnel is closed. This information consists of the server's public and VPN IP addresses, the client's VPN IP address, and the port and protocol over which the VPN tunnel runs. Additionally, it is necessary to store whether the rules are IPv4 or IPv6, as a different binary needs to be called for IPv6. The structure `nat_info` is defined for that purpose. Its definition is shows in listing 3.14. Note that the IP addresses are stored as strings. This allows them to be used directly when calling `iptables-restore`.

Listing 3.14: `struct nat_info` definition in `wilfred.h`

```
struct nat_info {
    bool ipv6;
    char vpn_loc[INET6_ADDRSTRLEN];
    char vpn_rem[INET6_ADDRSTRLEN];
    char pub_rem[INET6_ADDRSTRLEN];
    int proto;
    int port;
};
```

A global pointer to such a structure is declared in `wilfred.c`. Additionally, this pointer is also used to determine whether or not the traffic redirection has been enabled, with a NULL-pointer indicating that traffic redirection has not been enabled. The function `knock_nat_on(...)` adds the *iptables* rules, and while doing so, allocates and fills the global structure. `knock_nat_off()`, on the other hand, removes the *iptables* rules and reads the necessary information to do so from the global `nat_info` structure. Once it is done, it frees the allocated memory, thereby signaling that no redirection is configured anymore.

The simplified code for `knock_nat_on(...)`, which has been stripped of screen output, is shown in listing 3.15. The function first checks if trying to NAT IPv6 to IPv4 and aborts if this is the case. It also checks whether traffic redirection has previously been activated and if so, tries to deactivate the old one before activating the new one. It then determines whether to NAT IPv4 or IPv6 and gather the necessary information. For the IP addresses, this is done by helper functions which extract the IP addresses from various field in *OpenVPN*'s context. These helper functions will be described later. If any of the necessary information could not be determined, the function aborts here. If everything went well until now, a pipe is opened into `stdin` of `iptables-restore` or `ip6tables-restore` and the rules are written to that pipe.

Listing 3.15: Simplified code for `knock_nat_on(...)` in `wilfred.c`

```
static struct nat_info *nat_info = NULL;

bool knock_nat_on (struct tuntap *tuntap, struct sockaddr *sa, int proto) {
  FILE *ipt_restore;
  bool status;

  /* IPtables cannot NAT IPv4 to IPv6 or vice-versa.
   * Abort if only one of public/VPN address is IPv6. */
  if ((sa->sa_family == AF_INET6) && !(tuntap->ipv6)) {
    return false;
  }

  /* If NAT has already been enabled, disable the previous NAT before proceeding. */
  if (nat_info != NULL) {
    knock_nat_off();
  }
  nat_info = malloc(sizeof(struct nat_info));

  /* Dtermine whether to NAT IPv6 or not */
  nat_info->ipv6 = ((sa->sa_family == AF_INET6) && (tuntap->ipv6)) ? true : false;
  /* Fill in the NAT info. */
  status = true;
  status &= knock_nat_ip_vpn_local(nat_info->vpn_loc, tuntap, nat_info->ipv6);
  status &= knock_nat_ip_vpn_remote(nat_info->vpn_rem, tuntap, nat_info->ipv6);
  status &= knock_nat_ip_pub_remote(nat_info->pub_rem, sa, nat_info->ipv6);
  nat_info->proto = proto;
  nat_info->port = ntohs(nat_info->ipv6
                       ? ((struct sockaddr_in6 *)sa)->sin6_port
                       : ((struct sockaddr_in *)sa)->sin_port);

  if (!status) {
    free(nat_info);
    return false;
  }

  /* The actual magic. Open a pipe into stdin of iptables-restore as
   * recommended by the IPtables FAQ. */
  if (nat_info->ipv6) {
    ipt_restore = popen("/sbin/ip6tables-restore␣-n", "w");
  } else {
    ipt_restore = popen("/sbin/iptables-restore␣-n", "w");
  }
  fprintf(ipt_restore, "*nat\n");
  fprintf(ipt_restore, "-I␣OUTPUT␣-d␣%s␣-j␣DNAT␣--to␣%s\n",
          nat_info->pub_rem, nat_info->vpn_rem);
  fprintf(ipt_restore, "-I␣OUTPUT␣-d␣%s␣-p␣%s␣--dport␣%i␣-j␣ACCEPT\n",
          nat_info->pub_rem, proto_ntop_socket(nat_info->proto), nat_info->port);
  fprintf(ipt_restore, "-I␣POSTROUTING␣-d␣%s␣-j␣SNAT␣--to␣%s\n",
          nat_info->vpn_rem, nat_info->vpn_loc);
  fprintf(ipt_restore, "COMMIT\n");
  if (pclose(ipt_restore)) {
    return false;
  }

  return true;
}
```

The helper functions to gather the IP addresses mainly abstract away differences in IPv4 and IPv6 handling. Apart from using slightly different function calls for the conversion into strings, IPv6 addresses—unlike IPv4 addresses—are always stored in network byte order and thus do not require conversion from network to host byte order. Furthermore, IPv4 and IPv6 addresses are stored in different fields of *OpenVPN*'s context.

One of these helper functions which determines the server's internal IP address is however tricky, as this information is not easily accessible and has to be inferred. The way in which this is done depends on the chosen *OpenVPN* topology, of which there are three: p2p, net30 and subnet. The net30 topology allocates a full /30 subnet for every client and thus requires 4 addresses out of the address pool the *OpenVPN* server is configured to use. This topology is deprecated and therefore not supported for transparent redirection. The p2p topology uses Point-to-Point networking and the VPN internal IP address is stored directly in the context. The subnet topology is the recommended topology and uses the *OpenVPN* server's address pool as a subnet. Addressing is done using an IP and netmask out of this pool. The server always takes the lowest address of the pool. In this case, the server's address needs to be derived by adding 1 to the network address, which in turn needs to be derived from the client IP and the netmask.

The function to disable the transparent traffic redirection, `knock_nat_off()`, is simpler than `knock_nat_on(...)`, as it does not need to determine as much information. All necessary information is stored in the global `nat_info` structure and can simply be used as-is. As can be seen in listing 3.16, which shows a simplified version of `knock_nat_off()` that has been stripped of screen outputs, the function first checks whether transparent traffic redirection has previously been enabled and if not, successfully ends at this point, as there is no work to do. If there are *iptables* rules to be removed, the function proceeds to open a pipe into `stdin` of `iptables-restore` or `ip6tables-restore`, just like `knock_nat_on(...)`, but then issues the inverse commands[7] instead, thereby removing the rules. Lastly, it frees the global `nat_info` structure to signal that no transparent traffic redirection is currently active.

Note that changing iptables rules requires root privileges. When configuring *OpenVPN* to drop privileges after the tunnel is established, it will be unable to remove the NAT rules when closing the tunnel.

Listing 3.16: Simplified code for `knock_nat_off(...)` in `wilfred.c`

```
static struct nat_info *nat_info;

bool knock_nat_off () {
  FILE *ipt_restore;
```

[7] Using `-D` for "delete" instead of `-I` for "insert".

```
  /* Only proceed NAT has previously been enabled. */
  if (nat_info == NULL) {
    return true;
  }

  /* The actual magic. Open a pipe into stdin of iptables-restore as
   * recommended by the IPtables FAQ. */
  if (nat_info->ipv6) {
    ipt_restore = popen("/sbin/ip6tables-restore␣-n", "w");
  } else {
    ipt_restore = popen("/sbin/iptables-restore␣-n", "w");
  }
  fprintf(ipt_restore, "*nat\n");
  fprintf(ipt_restore, "-D␣OUTPUT␣-d␣%s␣-j␣DNAT␣--to␣%s\n",
          nat_info->pub_rem, nat_info->vpn_rem);
  fprintf(ipt_restore, "-D␣OUTPUT␣-d␣%s␣-p␣%s␣--dport␣%i␣-j␣ACCEPT\n",
          nat_info->pub_rem, proto_ntop_socket(nat_info->proto), nat_info->port);
  fprintf(ipt_restore, "-D␣POSTROUTING␣-d␣%s␣-j␣SNAT␣--to␣%s\n",
          nat_info->vpn_rem, nat_info->vpn_loc);
  fprintf(ipt_restore, "COMMIT\n");
  if (pclose(ipt_restore)) {
    return false;
  }

  /* Release the nat info */
  free(nat_info);
  return true;
}
```

Calls to those two functions are added to the existing *OpenVPN* code in the functions `do_open_tun(...)` and `do_close_tun(...)`. Those function are responsible for bringing *OpenVPN*'s tun interface up and down respectively. Listing 3.17 shows excerpts of these two functions with the addition of calls to `knock_nat_on(...)` and `knock_nat_off()`. The transparent traffic redirection is activated at the very end of the tun interface's initialization and is deactivated at the very beginning of the tun interface's deinitialization.

Listing 3.17: Excerpts of `do_open_tun(...)` and `do_tun_close(...)` in `init.c`

```
static bool do_open_tun (struct context *c) {
  struct gc_arena gc = gc_new ();
  bool ret = false;
  ...

  /* Enable S/DNAT */
  if (c->options.knock && c->options.knock_nat) {
    knock_nat_on(c->c1.tuntap,
                 c->c1.link_socket_addr.current_remote->ai_addr,
                 c->c1.link_socket_addr.current_remote->ai_protocol);
  }

  gc_free (&gc);
  return ret;
}

static void do_close_tun (struct context *c, bool force) {
  struct gc_arena gc = gc_new ();
  /* Disable S/DNAT */
  if (c->options.knock && c->options.knock_nat) {
    knock_nat_off();
  }

  ...
  gc_free (&gc);
}
```

## 3.4 Configuration Files

This section gives an example for configuration files for the *OpenVPN* server and the modified client. It also explains the options related to port-knocking, as well as some other important options. *OpenVPN* additionally supports a vast number of configuration options that we do not use. Since we use an unmodified server and the modified client retains all the functionality of the original client, these options are all still available, if necessary. Explaining each and every one of these options is, however, not the goal of this section, as this is done by the official *OpenVPN* documentation.

*OpenVPN* configuration files are plaintext files with one option per line. The available options are the same as the ones available as command line options, but without the leading "`--`". The file may contain empty lines and comments, which start with either a "`#`" or a "`;`".

The example configuration files are based on the ones we used during development and testing. They have, however been stripped of comments and some empty lines to save space. Furthermore, they contain a few options, which are commented out to demonstrate their use. Our test server had the IP address 10.0.2.15 and ran on port 2789[8], while the client had the IP address 10.0.2.4.

### 3.4.1 Server Configuration

Listing 3.18 shows an example configuration file for the server. It is based on the configuration we used during development and testing. Lines 1-6 are fairly standard configuration options for *OpenVPN*, setting it to run in routing mode (layer 3), establish the tunnel over UDP on port 2789, and function as a server. The server option's parameters determine the subnet that is used by *OpenVPN* internally. The interpretation of these depends on the topology used. In our case this is `subnet`, which means that the

[8] A wink to the Wheel of Time books. $(100)_{10}$ can be represented as $(AE5)_{16}$, which can be read as "Aes", the first part of "Aes Sedai". Any other port can be used, though using a different port than the standard port may impede detection by a hostile party, which is one of the goals of using port-knocking in the first place.

parameters to the `server` option represent the network address and netmask. It is also possible to bind *OpenVPN* to a single IP address by uncommenting the option in line 6.

Three topologies are possible: "`net30`", "`p2p`", and "`subnet`". `net30`, while deprecated, is still the default value for compatibility reasons. It allocates a full /30 subnet to every client, thereby using up four IP addresses for the pool. `p2p` uses a Point-to-Point topology and only requires one address per client. `subnet` is the recommended topology and configures the client's tun interface with an IP address and subnet mask like a normal interface. It thus also requires only one IP address per client. Whichever value is selected, it is automatically pushed to connecting clients.

Listing 3.18: `server.conf`

```
 1  dev tun
 2  proto udp
 3  port 2789
 4  server 172.16.23.0 255.255.255.0
 5  topology subnet
 6  ;local 10.0.2.15
 7
 8  tls-server
 9  pkcs12 /etc/openvpn/server.p12
10  dh /etc/openvpn/dh2048.pem
11  persist-key
12  persist-tun
13
14  keepalive 10 120
15  ;push "knock-refresh␣30"
16  verb 3
17
18  user nobody
19  group nogroup
20  cipher AES-256-CBC
21  auth SHA256
22  ;tls-auth ta.key 0
```

Line 8 instructs *OpenVPN* to take on the server role during the SSL/TLS handshake. The SSL/TLS role is independent from the *OpenVPN* role, meaning that the *OpenVPN* client could just as well act as the server and the *OpenVPN* server act as the client during the SSL/TLS handshake. The following lines provide the server with its certificate and Diffie-Hellman group. Furthermore, *OpenVPN* is instructed to preserve the tun/tap interface and the private keys read across restarts of its inner loop. This is useful, as later options cause *OpenVPN* to drop root privileges after initialization.

Line 14 defines *OpenVPN*'s ping behavior, by telling it to send an *OpenVPN* ping message to its counterpart every 10 seconds and consider the connection severed when no ping has been received from its counterpart for 120 seconds. These values are automatically pushed to the client. The server is also able to push the `knock-refresh [n]` directive to the client. This allows the operator to ensure that even improperly configured clients are able to maintain a tunnel through a firewall that does not support connection tracking for the selected transport layer protocol.

The verbosity setting in line 16 determines how much information *OpenVPN* outputs. 3 is a reasonable value for general use, while values of 4-6 are useful to debug connection issues. 0 is almost completely silent, except for fatal errors.

The last options are there to improve security. Lines 18 and 19 instruct *OpenVPN* to drop privileges after initialization. *OpenVPN* uses Blowfish as its cipher and SHA1 as its hash algorithm for the HMAC by default. To improve security, we change this to 256bit AES and SHA256. These options have to be mirrored in the client's configuration file too for the communication to be successful. If these options are omitted in the client's configuration, the connection will be established successfully, but data sent through the tunnel will be unreadable by the other side.

Lastly, it is possible to have *OpenVPN* sign every SSL/TLS packet with an HMAC. This allows the server to drop SSL/TLS packets with incorrect authentication code immediately, without wasting further resources to process them. A service operator may want to use this feature to help protecting the server against (Distributed) Denial-of-Service attacks. To use this feature, a secret key which is used for this HMAC has to be distributed to the server and all clients. Both, the server and the client need the `tls-auth` option with the key as its first parameter and either '0' or '1' as the second parameter, with '0' denoting the server and '1' denoting the client. Note that this configuration is only an example and several other ways exist to further improve the security of *OpenVPN*.

### 3.4.2 Client Configuration

The matching client configuration file is shown in listing 3.19. Its contents are similar to the server configuration file with only a few minor differences and additional options pertaining to port-knocking. The first three lines instruct *OpenVPN* to connect to 10.0.2.15 on port 2789 over UDP and run in routing mode (layer 3). The next two options (lines 5 and 6) cause *OpenVPN* to take on the role of the client during the SSL/TLS handshake and provide it with its private key, certificate, and CA certificate packaged in a PKCS#12 file. As stated in 3.4.1, the SSL/TLS roles do not necessarily coincide with *OpenVPN*'s role.

Lines 8-12 are the most interesting ones and contain port-knocking related options. Line 8 enables port-knocking, while line 10 provides *OpenVPN* with the server's certificate. This is necessary to pass this certificate on to *sKnock*, as *OpenVPN* itself does not need it. Line 11 enables transparent traffic redirection. If necessary, *OpenVPN* can also be instructed to periodically resend the port-knocking request. Alternatively, it is possible to omit this completely and have the server push to the client this option. Line 9 can be uncommented to tell OpenVPN to skip querying a password for the certificate and use it without a password.

Listing 3.19: `client.conf`

```
 1  remote 10.0.2.15 2789
 2  proto udp
 3  dev tun
 4
 5  tls-client
 6  pkcs12 /etc/openvpn/client.p12
 7
 8  knock
 9  ;knock-nopass
10  knock-cert-srv /etc/openvpn/server.crt
11  knock-nat
12  ;knock-refresh 30
13
14  pull
15  verb 3
16
17  cipher AES-256-CBC
18  auth SHA256
19  ;tls-auth ta.key 1
20  ;ns-cert-type server
```

The next two lines are miscellaneous options. `pull` is a very important option for the client and causes *OpenVPN* to accept options pushed by the server. `verb` determines how much screen output *OpenVPN* generates.

The last block of options serves the purpose of increasing security. `cipher` and `auth` set the cipher and HMAC hash to match the server's settings. If configuring *OpenVPN* to append SSL/TLS messages with an additional HMAC, as briefly described in section 3.4.1, the option `tls-auth FILE 1` has to be specified with FILE being the secret key used to calculate the HMAC and 1 denoting the client role. Lastly, it is possible to enforce that the client only connects to servers whose certificate have the nsCertType field set to "server". This feature protects against Man-in-the-Middle attacks, where authorized client attempts to connect to the client, and can be activated by uncommenting line 19.

Such a client configuration file can easily be prepared by a service's operator and distributed to the users in order to spare them having to configure *OpenVPN* themselves. The user is, however, free to modify this configuration file or create their own to better suit their needs if they wish to do so.

## 3.5 Limitations

*sKnock*—This solution has several minor limitations due to the use of *sKnock*. Firstly, though the *sKnock* library provides C bindings through a C wrapper, it is actually written in Python and uses several Python libraries. Consequently, its use adds additional dependencies on Python, the Python libraries used by *sKnock* and the C wrapper. Furthermore, *OpenVPN* allows to use inline certificates, which is a technique to embed certificates and private keys into the configuration file. However, the *sKnock* library's API expects to be handed paths to the certificates and any private keys it needs. This means that *OpenVPN*'s inline certificates cannot be used in combination with the port-knocking functionality.

Another limitation tied to *sKnock* is the need for the server's certificate to be supplied. *sKnock* uses the server's certificate in combination with an ephemeral ECC key pair on the same curve to derive a symmetric key with which the port-knocking packet is encrypted. However, an unmodified *OpenVPN* client does not need to be supplied with the server's certificate. We thus had to add a new option, `--knock-cert-srv FILE`, to pass a server certificate to *OpenVPN* so that it can in turn pass it on to *sKnock*.

*Double Certificate Transmission*—At the moment, the client's certificate needs to be sent to the server twice. It is sent once by *sKnock* during the port-knocking and once again by *OpenVPN* during the SSL/TLS Authentication. Though this seems unnecessarily inefficient, the additional overhead of sending a certificate once more than theoretically necessary is minimal. Improving this would require the *sKnock* daemon on the server to pass the client's certificate on to *OpenVPN*. This in turn would require significant changes to both the *sKnock* codebase and the *OpenVPN* codebase in order to allow them to cooperate in this way.

*NATting IPv4 and IPv6*—The last noteworthy limitation is the inability to transparently redirect traffic when *OpenVPN* is running the tunnel over IPv6, but is not configured to forward IPv6 traffic through the tunnel. This is due to our choice of using *iptables* NAT functionality to implement this traffic redirection, as *iptables* does not support NATting from IPv6 to IPv4. This can, however, easily be circumvented by activating *OpenVPN*'s ability to forward IPv6 packets through the tunnel. In this case, the modified *OpenVPN* client would simply NAT the public IPv6 address to the VPN's internal one with no significant difference to the user.

This limitation makes sense when contemplating that NATting from IPv6 to IPv4 would require replacing the entire IPv6 header by an IPv4 header. In the case of transport layer protocols that contain a checksum covering parts of or the entire IP header, such as TCP, this checksum would have to be entirely recalculated. This represents a considerable effort to remove a limitation that can easily be circumvented. Perhaps nftables, the successor to *iptables*, which is currently in development, will support this, though it seems unlikely.

Another aspect to consider in relation to *iptables* is the need for root privileges to alter the rules. *OpenVPN* offers the option to drop privileges after initialization. When used in combination with transparent traffic redirection using *iptables*' NAT functionality, this would lead to the rules being added successfully when the tunnel is established.

However, once the tunnel is established, the root privileges would be dropped and the rules could not be removed when closing the tunnel.

*Operating System Support*—The current implementation has not been developed with support for Windows in mind. As such, it currently only support *nix systems. Additionally, the current implementation of the transparent traffic redirection feature depends on *iptables*. Porting this feature to operating systems on which *iptables* does not run, such as Windows, will be tricky and may require a different way of implementing the feature, depending on the NAT capabilities of the Windows Firewall. Since this feature, while comfortable, is not essential for the functioning of our solution, we deem this to be a minor issue.

# Chapter 4

# Testing

## 4.1 Test Setup

Our test setup involves two virtual machines. One of them takes on the role of the server hosting a service, while the other takes on the role of a client trying to access that service. Both machines run Debian 8.4.

*Server VM*—The server virtual machine has the IP address 10.0.2.15 and runs a simple *apache* server with a test page. This serves as our test service. *Apache* is installed from the Debian repositories. The default configuration displays a "It works!" page, which is sufficient to check the service's reachability.

Furthermore, the server VM also runs the *sKnock* daemon and *iptables*. While *iptables* can be installed from the Debian repositories, sKnock has to be installed manually. *sKnock*'s toplevel directory needs to be part of the python search path. This can be achieved by adding the path to the environment variable `PYTHONPATH`, which works exactly like the environment variable `PATH`. *sKnock* also requires a few python modules to be installed. These are "python-iptables", "m2crypto", "cryptography", "six" and "hkdf". The modules can either be installed through the *pip* utility or through the Debian repositories. However, since some of the modules are not available in Debian's repositories, we chose to install them all with *pip*. The *sKnock* daemon is configured to use the same certificate file as *OpenVPN*, which is located in `/etc/openvpn/server.p12`.

*OpenVPN* can be installed from the Debian repositories as only the client has been modified. It uses the same configuration options that were explained in section 3.4.1. The configuration file, as well as the private key, certificate, and CA certificate—which are bundled into a PKCS#12 file—are is located in `/etc/openvpn`. Most notably, *OpenVPN* is configured to run over UDP on port 2789. It uses the "subnet" topology and allocates addresses from the 172.16.23.0/24 subnet to its client, while assigning the IP address 172.16.23.1 to its own tun interface.

*Client VM*—The client virtual machine has the IP address 10.0.2.4 and has the *sKnock* library and its C wrapper installed. Just like for the server setup, *sKnock*'s toplevel directory needs to be part of the python search path. The environment variable `PYTHONPATH` is thus appended with the correct path. The C wrapper's shared object has to be in one of the folders on the library path. We chose to place it under `/usr/local/lib`. Unlike the *sKnock* daemon running on the server, the client side *sKnock* library does not require any additional configuration.

Furthermore, the modified *OpenVPN* version is installed on the system. The configuration options used are the same ones that are described in section 3.4.2. Most notably, it connects to the server (IP 10.0.2.15) on port 2789 over UDP. Port-Knocking and transparent redirection are also enabled. The configuration file is stored in `/etc/openvpn/`as `client.conf`. The client's private key, certificate, and CA certificate are bundled into a PKCS#12 file, which is stored in the same directory as the configuration file. The server's certificate is also stored in that directory.

## 4.2 Results

We use *Firefox* on the client VM to access the apache server running on the server VM. As expected, the website is unreachable without further steps, as the port is closed. Once we launch the modified *OpenVPN* client and have it establish a tunnel, the website is now available under the server's VPN internal IP address, 172.16.23.1. Thanks to the transparent traffic redirection, the website is also accessible by entering the server VM's public IP address, 10.0.2.15. After closing the tunnel, the website is once again unavailable through either IP address. All features, especially the `--knock-refresh [n]` and `--knock-nat` have been tested and have been found to work properly. Furthermore, variations of the configuration where the server pushes the `--knock-refresh [n]` option to the client have also been tested and worked as expected.

We have however noticed an issue with the compatibility of the certificates between *OpenVPN* and *sKnock*. When using the certificates provided by *sKnock*'s author as an example, *OpenVPN* is unable to find a common cipher suit, even though several cipher suits overlap. These certificates were generated on Windows using appropriate Microsoft tools, which might be the cause for this issue. The parameters used in the creation of the example certificates are unknown to use, which might also be the source of the issue. When using certificates generated by *OpenSSL* through the *OpenVPN* team's *EasyRSA 3* script, *OpenVPN* has no problem finding a common cipher suit and establishing a connection.

Note that *sKnock* only works with the NIST-P 256 curve at the moment [3]. Using any other curve will cause *sKnock* to fail to perform the port-knocking.

Another observation we have made, is that when the VPN tunnel is active, even a simple HTTP request causes several packets to be sent which are recognized by *sKnock* as potential port-knocking requests. They are thus processed further before being found to be malformed. We suspect this to be due to the *OpenVPN* encapsulation increasing the packet size to a similar size to that of a port-knocking request. *sKnock* uses the packet size to coarsely filter packets before processing them further to avoid wasting time on unnecessarily decrypting packets which can not be valid requests. The effects of this phenomenon on *sKnock*'s performance are unknown. Because of time constraints, we did not investigate this further.

Since we did not change the way in which the OpenVPN core operates, our modifications have no influence on its performance. This performance has been the subject of several papers, such as [47] and [48]. Measuring the performance again would not have been of any use. Moreover, no performance test we could have done in the scope of this thesis could have been as thorough as the existing ones. The additional delay to the tunnel establishment incurred by using port-knocking is the delay added by *sKnock*, which has also already been measured [3]. As such, we did not deem it necessary to perform any new measurements.

# Chapter 5

# Conclusion

Over the course of this work, we explained the rationale behind developing a solution to allow existing applications to communicate through an encrypted channel with a port-knocked service, before moving on to present several approaches to achieve this goal. We wanted the solution to be easy for the operator and the user to set up, which also means avoiding to change the application's source code. Since this involves tunneling in some form or another[1] to ensure the encryption of the traffic even if the application itself does not support encrypted communication, this comes at the cost of additional transmission overhead. This overhead was analyzed, as well as other respective strengths and weaknesses of the approaches. Based on this analysis, an approach has been selected and then implemented.

## 5.1 Future Work

*OpenVPN* supports what it calls "inline certificates". Normally, the certificates and private keys are saved as separate files. Inline certificates are instead embedded in the configuration file. However, our implementation does not currently support using such inline certificates in combination with the port-knocking functionalities. This is due to *sKnock* expecting to be handed a path to the required certificates and keys. When using inline certificates, it would be directly handed the certificate or key itself. Future work could add support for these inline certificates, either by creating a temporary file and writing the contents of the inline certificate to it before handing the path to this temporary file to *sKnock*, or by modifying *sKnock* itself and adding support for accepting the certificate directly, instead of a path to a file. Inline certificates could facilitate the distribution of configuration files to end-users by sparing the user from the task of placing the certificates and keys at specific locations on the file system where the configuration file tells *OpenVPN* to look for them.

[1] For example an SSH tunnel, an IPsec tunnel, or an *OpenVPN* tunnel

Furthermore, the current implementation requires the client's certificate to be sent twice: once by *sKnock* for the port-knocking and once by *OpenVPN* for the SSL/TLS handshake. The additional overhead is not huge, but it would be more elegant if the certificate could be sent only once. This could be achieved by modifying *sKnock* and *OpenVPN* to allow them to work together. While these changes are far beyond the scope of this work, they would create a base for further modifications of the *sKnock* API to allow it to pass certificates on to any service that uses certificate based authentication and is aware of the *sKnock* API.

The current implementation is not able to disable the transparent traffic redirection when closing the tunnel if the root privileges were dropped after creating the tunnel. A future implementation could offload this functionality into a separate process, which does not drop the root privileges. This process could communicate with the main process through IPC and should only be able to enable and disable the *iptables* rules for transparent traffic redirection.

Lastly, *sKnock* allows to encode a list of permitted ports into the client's certificate. The client is then restricted to opening only those ports. Using the modified *OpenVPN* client could completely circumvent this security feature. Future work could enable *OpenVPN* to either extract this list of ports from the certificate itself, or implement some form of cooperation with *sKnock* to receive this information from the *sKnock* server daemon. It would then be necessary to implement some kind of port filtering in *OpenVPN* to restore the security benefits of this feature, while maintaining the comfort granted by the current implementation.

## 5.2 Summary

Of the three approaches we presented, we selected *OpenVPN*, mostly because of its flexibility and ease-of-use. The other approaches also turned out to have important drawbacks, from which the *OpenVPN* approach did not suffer. The proxy approach, for example, does not completely eliminate the need to change the code of the original application, as some application would need to have SOCKS5 support added. The *OpenSSH* approach on the other hand only allows to forward TCP connections and does this over another TCP connection, thereby incurring the risk of problems resulting from TCP-over-TCP.

Once the choice was made, we moved on to implement this solution step by step. First we added support for initializing and using the port-knocking library to open the *OpenVPN* server port. Then we started adding additional features, such as refreshing the port-knocking request if necessary. We also added a transparent redirection feature, which—when activated—allows the user to just start the modified *OpenVPN* client and then use the application as if the modified *OpenVPN* client was not there.

Only the client side was modified, while the *OpenVPN* server's code was left untouched. Because of this, the resulting modified *OpenVPN* version remains fully compatible with unmodified *OpenVPN* versions and can thus function as a normal *OpenVPN* client without any of the additional functionality we added.

We then went on to describing the new configuration options added for the new features and explained sample configuration files to showcase the use of our modified *OpenVPN* version. Moreover, we explained that the service's operator could create and distribute client configuration files to make it easier for the user to set up. Lastly, we explained the limitations of our approach and gave pointers on how to remove or alleviate these limitations in future versions.

# Appendix A

# Additional Packet Diagrams

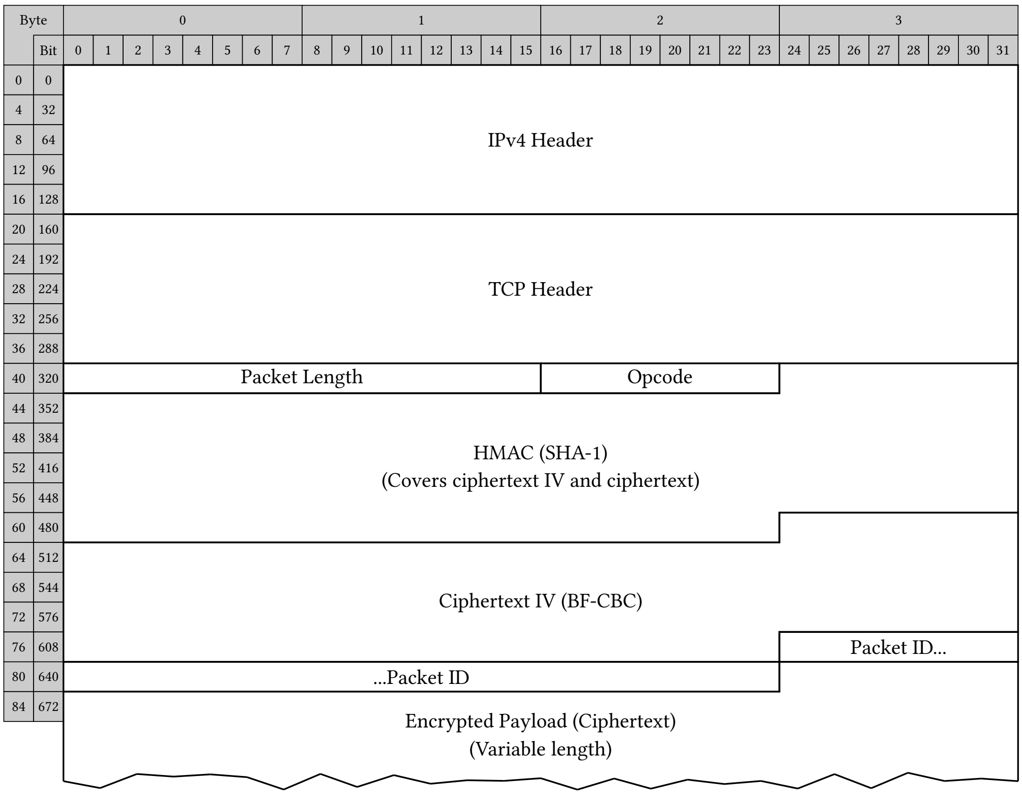

Figure A.1: A VPN packet over IPv4 and TCP

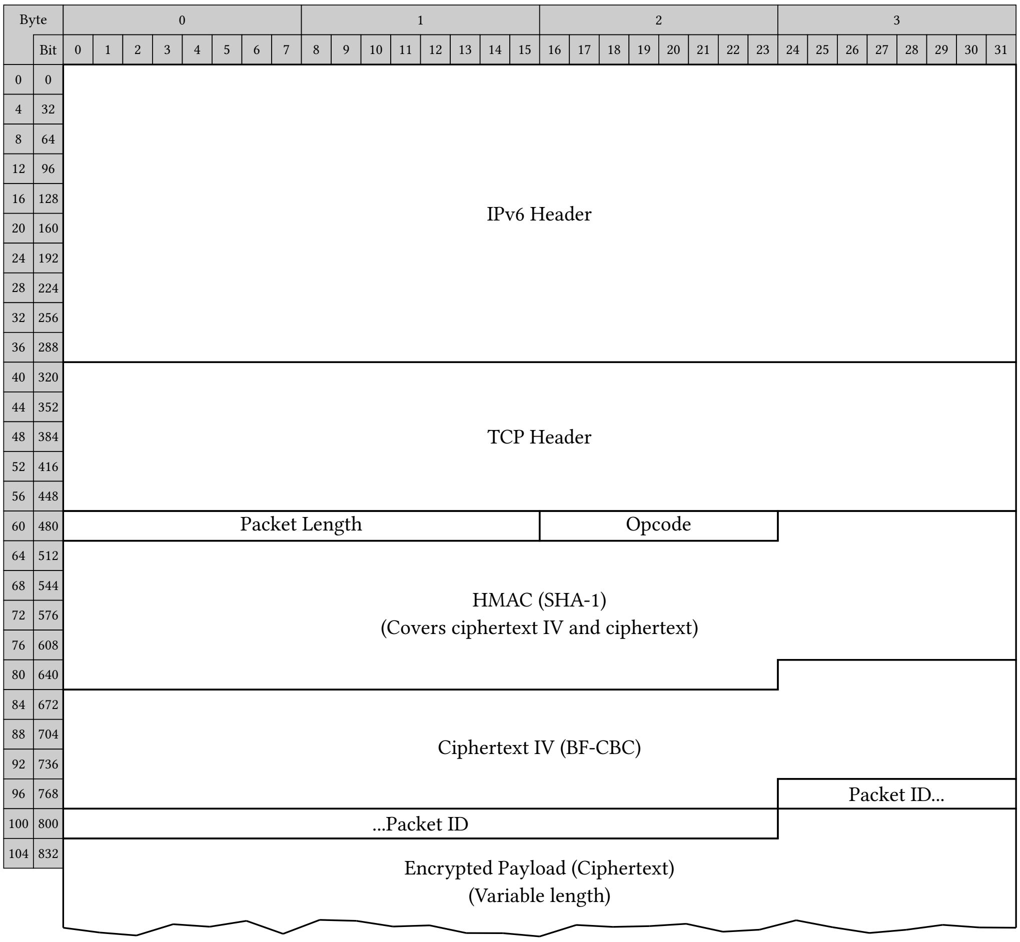

Figure A.2: A VPN packet over IPv6 and TCP

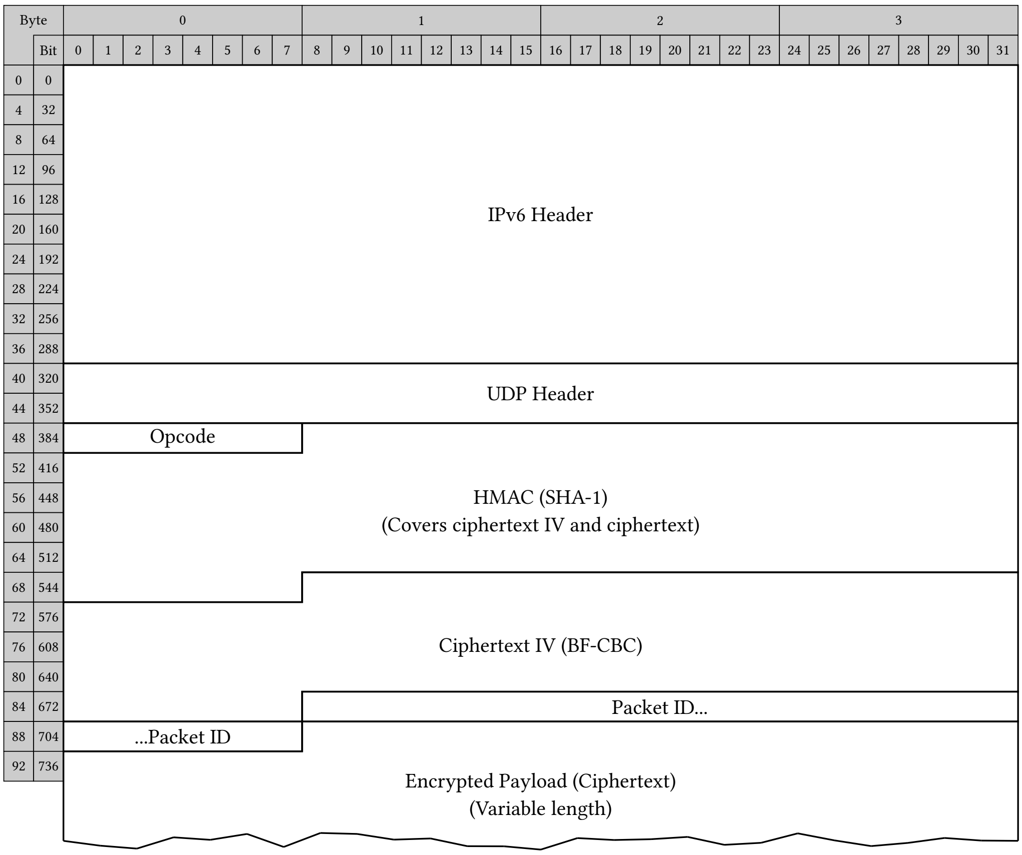

Figure A.3: A VPN packet over IPv6 and UDP

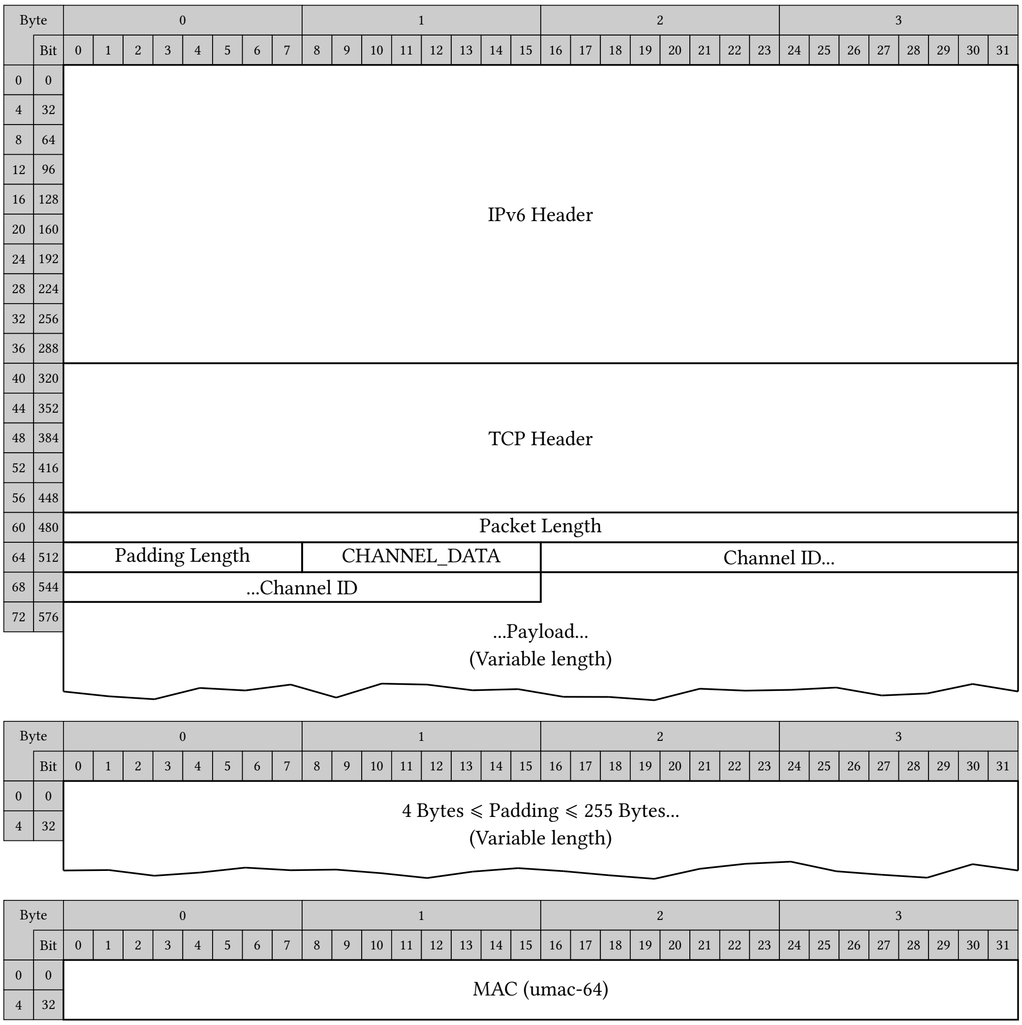

Figure A.4: As SSH packet over IPv6

# Bibliography

[1] B. Daya, “Network security: History, importance, and future,” *University of Florida Department of Electrical and Computer Engineering*, 2013.

[2] M. Krzywinski, “Port knocking from the inside out,” *SysAdmin Magazine*, vol. 12, no. 6, pp. 12–17, 2003.

[3] D. Sel, “Authenticated scalable port-knocking,” Bachelor Thesis, Technical University Munich, 2016.

[4] M. Rash, “Single packet authorization with fwknop,” *login: The USENIX Magazine*, vol. 31, no. 1, pp. 63–69, 2006.

[5] Y. Desmedt, “Man-in-the-middle attack,” in *Encyclopedia of Cryptography and Security.* Springer, 2011, pp. 759–759.

[6] P. Syverson, “A taxonomy of replay attacks [cryptographic protocols],” in *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings.* IEEE, 1994, pp. 187–191.

[7] D. E. Denning and G. M. Sacco, “Timestamps in key distribution protocols,” *Communications of the ACM*, vol. 24, no. 8, pp. 533–536, 1981.

[8] D. Johnson, A. Menezes, and S. Vanstone, “The elliptic curve digital signature algorithm (ecdsa),” *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[9] R. L. Rivest, A. Shamir, and L. M. Adleman, “Cryptographic communications system and method,” Sep. 20 1983, uS Patent 4,405,829.

[10] A. Kumar, A. Jerome, G. Khanna, H. Veladanda, N. Ly, Hoa amd Chai, and R. Andrews, “Elliptic curve cryptography (ecc) certificates performance analysis,” may 2013. [Online]. Available: https://www.symantec.com/content/en/us/enterprise/white_papers/b-wp_ecc.pdf

[11] K. Maletsky, “Rsa vs ecc comparison for embedded systems,” jul 2015. [Online]. Available: http://www.atmel.com/images/atmel-8951-cryptoauth-rsa-ecc-comparison-embedded-systems-whitepaper.pdf

[12] T. Oder, T. Pöppelmann, and T. Güneysu, "Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 110:1–110:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593098

[13] N. Jansma and B. Arrendondo, "Performance comparison of elliptic curve and rsa digital signatures," *nicj. net/files*, 2004.

[14] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "Rfc 1928: Socks protocol version 5," mar 1996.

[15] O. Team. Openssh website. [Online]. Available: http://www.openssh.com/

[16] G. Venkatachalam, "The openssh protocol under the hood," *Linux J*, vol. 156, p. 6, 2007.

[17] IANA. (2016, apr) Service name and transport protocol port number registry. [Online]. Available: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[18] O. Team. Manpage ssh_config(5). [Online]. Available: https://manpages.debian.org/cgi-bin/man.cgi?query=ssh_config&manpath=Debian+8+jessie

[19] S. Tatham. (2016, apr) Putty website. [Online]. Available: http://www.chiark.greenend.org.uk/~sgtatham/putty/

[20] T. Ylonen and C. Lonvick, "Rfc 4253: The secure shell (ssh) transport layer protocol," jan 2006.

[21] T. Ylonen and C. Lonvick, "Rfc 4254: he secure shell (ssh) connection protocol," jan 2006.

[22] J. Postel, "Rfc 791: Internet protocol," sep 1981.

[23] S. Deering and R. Hinden, "Rfc 2460: Internet protocol, version 6 (ipv6)," dec 1998.

[24] J. Postel, "Rfc 793: Transmission control protocol," sep 1981.

[25] M. Bider and M. Baushke, "Rfc 6668: Sha-2 data integrity verification for the secure shell (ssh) transport layer protocol," *NIST Special Publication*, vol. 800, p. 131A, jul 2012.

[26] D. Miller and P. Valchev, "Draft: The use of umac in the ssh transport layer protocol," sep 2007.

[27] C. Namprempre, T. Kohno, and M. Bellare, "Rfc 4344: The secure shell (ssh) transport layer encryption modes," jan 2006.

[28] D. Stebila and K. Igoe, "Rfc 6187: X.509v3 certificates for secure shell authentication," mar 2011.

[29] O. Team. (2012, mar) Openssh certificates. [Online]. Available: http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/PROTOCOL.certkeys?rev=1.9

[30] T. Ylonen and C. Lonvick, "Rfc 4251: The secure shell (ssh) protocol architecture," jan 2006.

[31] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama, "Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency," in *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, M. Atiquzzaman and S. I. Balandin, Eds., vol. 6011, Oct. 2005, pp. 138–146.

[32] O. Titz. (2001, apr) Why tcp over tcp is a bad idea. [Online]. Available: http://sites.inka.de/bigred/devel/tcp-tcp.html

[33] M. Ullholm Karlsson, M. Habib *et al.*, "Ssh over udp," 2010.

[34] O. Team. Openvpn. [Online]. Available: https://openvpn.net/

[35] O. Team. (2010, apr) Openvpn. [Online]. Available: https://raw.githubusercontent.com/OpenVPN/openvpn/master/COPYING

[36] S. Kent, "Rfc 4302: Ip authentication header," dec 2005.

[37] C. Hosner, "Openvpn and the ssl vpn revolution," *SANS Institute, Aug*, 2004.

[38] J. Postel, "Rfc 768: User datagram protocol," aug 1980.

[39] O. Team. Openvpn security overview. [Online]. Available: https://openvpn.net/index.php/open-source/documentation/security-overview.html

[40] O. Team. Openvpn ssl.c. [Online]. Available: https://raw.githubusercontent.com/OpenVPN/openvpn/master/src/openvpn/

[41] R. Russel and H. Welte. (2002, jul) Linux netfilter hacking howto. [Online]. Available: http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-4.html#ss4.4

[42] P. Ayuso, "Netfilter's connection tracking system," *LOGIN: The USENIX magazine*, vol. 31, no. 3, 2006.

[43] O. Andresson. (2008) The state machine. [Online]. Available: http://www.iptables.info/en/connection-state.html

[44] R. Russel. (2002, jan) Linux 2.4 nat howto. [Online]. Available: http://www.netfilter.org/documentation/HOWTO//NAT-HOWTO-5.html

[45] O. Andresson. (2008, jan) Traversing of tables and chains. [Online]. Available: http://www.iptables.info/en/structure-of-iptables.html

[46] Iptables faq. [Online]. Available: http://www.netfilter.org/documentation/FAQ/netfilter-faq-4.html

[47] B. Hoekstra, D. Musulin, and J. J. Keijser, “Comparing tcp performance of tunneled and non-tunneled traffic using openvpn,” *Universiteit Van Amsterdam, System & Network Engineering, Amsterdam*, pp. 2010–2011, 2011.

[48] I. Kotuliak, P. Rybár, and P. Truchly, “Performance comparison of ipsec and tls based vpn technologies,” in *Emerging eLearning Technologies and Applications (ICETA), 2011 9th International Conference on.* IEEE, 2011, pp. 217–221.