Bachelor thesis
Computer Science

Radboud University

Creating a secure virtual private network using minimal code

Author:
Stan Derksen
s4386388

First supervisor/assessor:
Dr. Peter Schwabe
peter@cryptojedi.org

February 9, 2017

Abstract

This thesis introduces a new open-source VPN solution called NaClShuttle. It explains what the importance is of a minimal program and describes how NaClShuttle is designed to succeed in achieving its goals to be a minimal and secure VPN solution. Furthermore it describes how NaClShuttle works and how it compares to currently used popular VPN solutions. NaClShuttle features a cryptographically secure tunnel while maintaining the potential to be audited by third parties.

Contents

Chapter 1

Introduction

The importance of security in an online environment becomes larger each day. As many new technologies develop, they bring many new threats to security with them. An example of this is the introduction of JavaScript in 1995. JavaScript brought great dynamic possibilities to static web browsers, making it possible to dynamically alter elements of a website without having to reload it in its entirety. Even though this was a great and useful addition for web developers and users, it also allowed malicious attackers to create cross-site scripting attacks for example, which nowadays are still one of the most used attacks against websites.

At the moment online security is largely improved, developers are more aware of threats and secure their websites more, development frameworks are introduced which have built-in security, more and more websites enforce HyperText Transfer Protocol Secure (HTTPS), etc. However, you might still end up in a situation where you are connected to an unsecured WiFi network, for example in restaurants or at airports. If you still want to make sure that your data is sent over a secure line you could make use of a Virtual Private Network (VPN).

A VPN routes your traffic through a secure encrypted tunnel to a destination of your choice. An added benefit is that you will also have access to files on the server you are connecting to, making it a great solution for large companies with a protected private network. There exist many different VPN solutions which all offer various packages including different add-ons, platform support, usability, price, etc.

Most VPNs have a very complex and large source code. This means that even though the main functionality of a VPN is to provide security, it is very hard to verify that these services are actually secure because it is way too time consuming and expensive to audit such huge codebases even though just one line of code might lead to large security flaws. A famous example of this problem is Apple's SSL/TLS bug [28], which had the following lines of code:

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```

As you can see, the developer accidentally duplicated the `goto fail;` line, making it so that even if the if-condition is false, it will always hit the second `goto fail;` line. The lines of code were from a signature verification method, which resulted in the fact that the signature verification never failed.

From Apple's SSL/TLS bug we can see how easily flaws in very important pieces of code are overlooked. Now imagine how many of these small mistakes can be made by human error in a project that has around 500.000 lines of code. It is especially worrying if these errors are made in parts of code that have to deal with security of information, which brings us to the Trusted Code Base (TCB).

The first time a TCB was mentioned was by J. Rushby in 1981 [33], who described it as the collection of the kernel and trusted processes that are running. Later, in [27], TCB is described as "a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security."

The problem with VPN solutions that are available today is that they all have a TCB which is way too large to check for security flaws. Some of these will be discussed in the third chapter. Because people are trusting these VPN applications to keep them secure and to send large amounts of private information it is essential that there is an application that is assumed to be secure. To do this we need to create a VPN solution with a TCB that is as small as possible.

The main idea of a new solution is to create a program that is just a basic tunnel between two entities and add cryptography on this tunnel using an as small as possible cryptographic library. This will result in a program that is minimal enough to be audited by a third party without depending on too much other software. Finally, the idea is to make the TCB that runs as root as small as possible.

Compared to other solutions that are currently available the code of the program should be way smaller. In terms of lines of code the aim is to reduce the size to less than 10% of the most popular VPN solutions currently in use. Also the TCB should be so small that it is easy to pinpoint exactly where the TCB is, in contrary to the current VPN solutions, where the TCB is buried somewhere in the gigantic codebase.

Chapter 2

Preliminaries

To understand more about the problem at hand and how current solutions work, it is assumed that the reader has basic knowledge about terms used in computer networks and how computer networks work. Before we go into further details about how VPNs work, we will describe a basic computer network and the principles involved, that are used in this thesis.

2.1 Computer networks

2.1.1 Ethernet

The definition of a computer network is two or more computers that are connected, so that information can be transmitted between them. Examples of computer networks are your WiFi at home, the Internet, two laptops linked with an Ethernet cable, etc. To make computer networks perform well, strict standards must be agreed upon to send and receive data correctly. This is where network protocols come in place, which are sets of rules and conventions of how computers communicate in a certain network. Typical aspects of network protocols are message format (in particular header format), data encoding, allowed messages and expected answers, session initialisation and termination and synchronisation of communication.

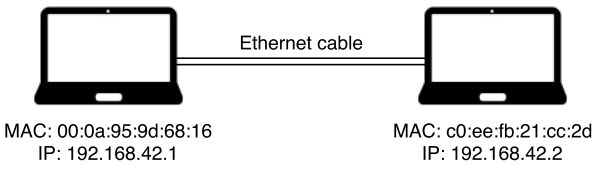

Figure 2.1: A simple Ethernet computer network

The easiest form of computer networking is Ethernet. A very basic Ethernet setup is shown in Figure 2.1. The principle of Ethernet is placing

data on the cable, which can be seen by everyone connected to this cable. Data packets that are sent on an Ethernet cable are sent in frames, with each frame having the same layout, shown in Figure 2.2.

Preamble	Start of delimiter	Dest. MAC address	Source MAC address	802.1Q tag (optional)	Ethertype or length	Payload	Frame check sequence	Interpacket gap
7 Bytes	1 Byte	6 Bytes	6 Bytes	4 Bytes	2 Bytes	46 (or 42) - 1500 Bytes	4 Bytes	12 Bytes

Figure 2.2: An Ethernet frame

The important parts, as marked in blue, are the destination and source MAC address and the payload. A media access control (MAC) address is a 48-bit address, denoted in the form of ff:ff:ff:ff:ff:ff, which is a unique identifier for network interfaces. MAC addresses are usually fixed, which is fine for the network in Figure 2.1, but in a larger network this might lead to issues. For example, when a computer or even just a network card is replaced, all other computers in the network must be updated with its new MAC address.

2.1.2 Internet Protocol and routing

This is where Internet Protocol (IP) comes in. Currently there are two versions of IP, namely Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6). For this thesis we will look at IPv4. IP provides higher-level 32-bit logical addresses, denoted in *dotted decimal*, such as 192.168.42.1. IP addresses have a network part and a host part, addresses with the same network part are directly reachable. In the beginning of IP, the first byte was used for the network part and the last three parts were the host part. Nowadays we use a variable length subnet mask (VLSM) to denote how many bits are used for the network part. This means that a network with a 255.255.255.0 netmask has the first three bytes as the network part. Together with an IP address this is denoted as 192.168.42.1/24. Like Ethernet has MAC frames, IP has IP packets with IP headers. In Figure 2.3 is shown what an IP header looks like.

To make use of these IP addresses we need a protocol that maps IP addresses to MAC addresses. This protocol is called Address Resolution Protocol (ARP). In this protocol entity A with IP address 192.168.42.1 sends an ARP message "Who has 192.168.42.2? Tell 192.168.42.1" to broadcast address ff:ff:ff:ff:ff:ff. All computers on the network then check whether they have IP address 192.168.42.2. Entity B with IP address 192.168.42.2 then replies with "192.168.42.2 is at c0:ee:fb:21:cc:2d", which entity A saves in its ARP cache.

Apart from wired connections like Ethernet, wireless connections such as IEEE 802.11 (WiFi) are widely used nowadays. Wireless connections are designed to behave in the same way as wired connections. For example,

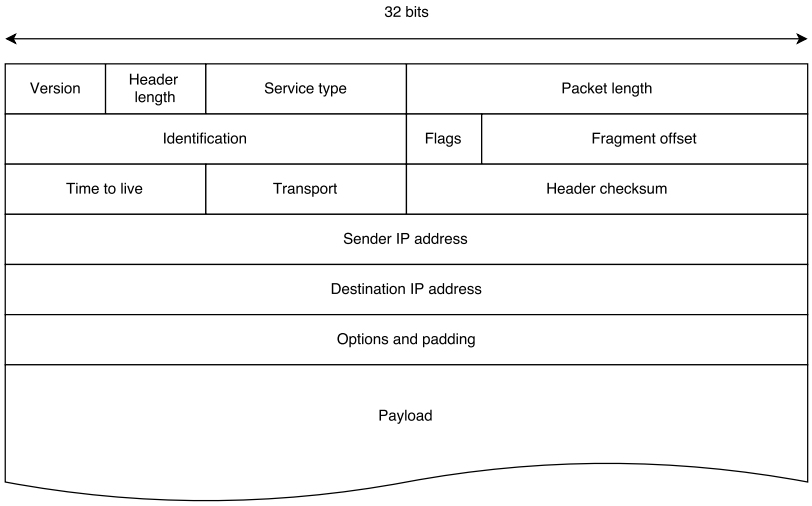

Figure 2.3: An IP packet

association with a network corresponds to plugging in an Ethernet cable. Wireless networks nearby are identified by their service set identification (SSID) and communication is physically split using different channels, which are frequencies. In infrastructure mode, communication goes through access points (AP). APs send beacon frames, by default 10 per second, containing timestamp, beacon interval, SSID and frequency hopping parameters. To connect to this AP, a client sends an authentication request including an identifier, which is its MAC address, the SSID, etc., if the AP decides to accept the request, it sends authentication OK back, then the client sends an association request and the AP sends association OK back.

In order for IP to deliver packets from one host to another it needs to find a path between these two hosts. Finding the path between the source and destination is called routing. Routers are computers that forward packets in a network. The simplest form of routing is static routing, where routes are saved in routing tables. To view the routing table on Linux, running `route -n` shows an routing table like the exampe in Figure 2.4.

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         145.116.144.1   0.0.0.0         UG    600    0        0 wlan0
131.174.117.20  145.116.144.1   255.255.255.255 UGH   600    0        0 wlan0
145.116.144.0   0.0.0.0         255.255.252.0   U     600    0        0 wlan0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 wlan0
```

Figure 2.4: An example of a routing table

Adding a new static route is done by executing the following command:

```
ip route add 192.168.42.0/24 via 192.168.42.254
```

This command routes all traffic with a destination that belongs to 192.168.42.0/24

via the 192.168.42.254 gateway. A special route command is the default gateway route. This is added by running:

```
ip route add default via 192.168.42.254
```

Now all traffic that is not routed to a specific address by any other rule is routed to 192.168.42.254 by default. Again, static routing works okay for small networks, but is not feasible for large networks because it will become way too complex and it is hard to react to changes in the network. To adapt to a larger network dynamic routing should be used. In dynamic routing routers communicate to their neighbours and create a dynamic routing table for efficient routes.

2.1.3 Network address translation

IPv4 uses 32-bit addresses, which means that it has at most 4.294.967.296 different addresses. Some addresses are special addresses and because of network seperation it is hard to use all addresses. This means that there is a problem: We are running out of IP addresses. The long-term solution for this problem is using IPv6, which uses 128-bit addresses, but implementing and standardising this is difficult. That is why the short term solution network address translation (NAT) is used.

NAT makes use of the fact that there are multiple hosts on a single network and only one host, the gateway, has an IP address which is routed to the Internet. Multiple hosts will then use this single gateway as IP address by rewriting the source IP address for outgoing packets and saving this translation as shown in Figure 2.5, in terms of IP address and port number, to rewrite the destination address for incoming packets.

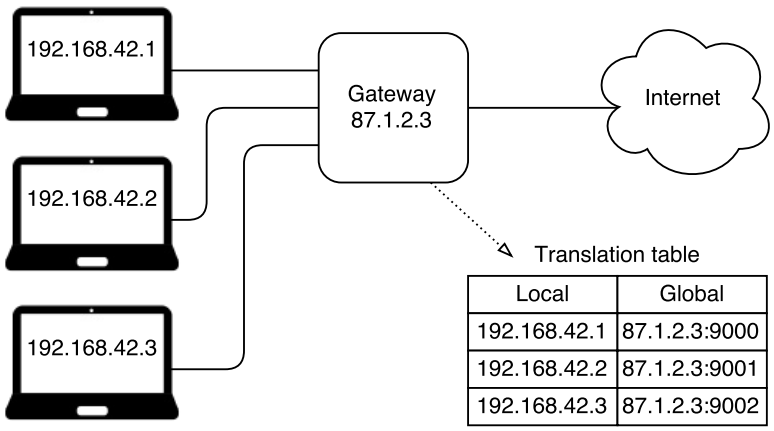

Figure 2.5: Network address translation

The concept above is also known as source NAT, because it rewrites the source address of a packet. But say we run a server at a certain port, how can we make sure that the packet arrives at the right process? Forwarding connections to a server is called port forwarding, also known as destination NAT. Port forwarding can be done with Linux' `iptables`. The general layout of the routing in `iptables` can be found in Figure 2.6.

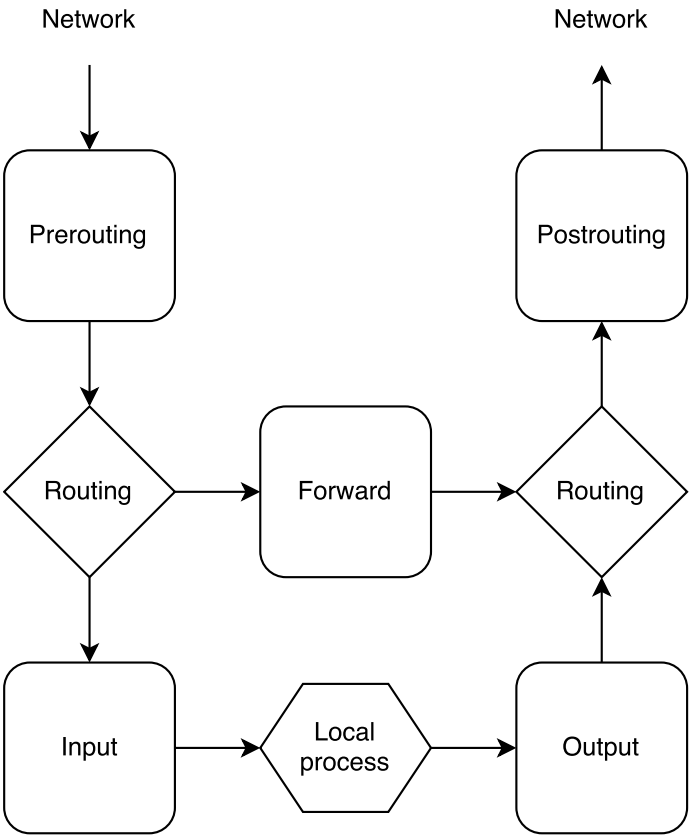

Figure 2.6: General layout of `iptables`

Using `iptables`, packets that are meant for a certain IP address and port combination are forwarded to a different address combination. This way computers or services in the local area network (LAN) that are not directly accessible can now become accessable for specific clients. Enabling NAT, also known as IP masquerading, in `iptables` can be done through the following commands:

```
iptables -t nat -A POSTROUTING -j MASQUERADE
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Port forwarding can then, for example, be done by:

```
iptables -A PREROUTING -t nat -p tcp --dport 1337 -j DNAT --to 192.168.42.1:22
iptables -A FORWARD -p tcp -d 192.168.42.1 --dport 22 -j ACCEPT
```

Packets that are directed destination address 87.1.2.3:1337 are now forwarded to 192.168.42.1:22.

2.1.4 Tunneling

A tunnel is a mechanism used to transport packets of a foreign protocol across a network that normally does not accept those packets. Tunneling is done by encapsulating packets, this means that an entire packet is transported in the payload of a different packet. An example is a tunnel over Secure Shell (SSH).

SSH is a protocol that offers the possibility to securely log into a system from a remote destination, along with other secure network services over an insecure network. Clients can authenticate through either a password or a private and public key pair. SSH typically runs over a reliable TCP/IP connection.

Say we want to connect to an SMTPS mail server, running on port 465, at `mail.destination.com`, but port 465 is blocked and 80 is open. Say you have an SSH server running at `source.com` on port 80. You can then host an SSH tunnel using the following command:

```
ssh -p 80 -L 12345:mail.destination.com:465 source.com
```

It is now possible to go to `localhost:12345` in your web browser and connect to the mail server. What happens is that SSH will forward the connection to mail.destination.com at port 465. To `mail.destination.com` it will look like the connection is coming from `source.com`.

This concept is especially useful if we forward a port from our own computer to a remote computer or server, effectively sending all your data over the SSH connection, making it secure. This is because when a port is forwarded in SSH it listens to a local socket for a connection. When this connection is made, it will forward the entire connection on to the remote host and port, making a secure tunnel on this port.

2.1.5 TUN/TAP devices

TUN/TAP devices are virtual network interfaces available on Linux and most UNIX-based operating systems. The devices allow for user-space programs to do packet transmission and reception at either the Ethernet or IP level, depending on which device is used. Unlike regular network interfaces, TUN/TAP devices do not have a physical component.

TUN/TAP devices act like they are network interfaces, which means that when data is supposed to be put “on the wire”, it is sent to the TUN/TAP

device instead. The data is then sent to an user-space program that is attached to the device. This program can then utilise a special file descriptor which can be written to and read from.

The difference between TUN and TAP devices is that a TAP device outputs and must be given Ethernet frames while a TUN device outputs and must be given raw IP packets. This means that TUN devices operate at level three and TAP devices at level two of the network stack.

2.1.6 Virtual private network

A VPN allows you to create a direct connection to a different network over a public network such as the Internet. Large corporations and educational institutes use VPNs to enable remote access to their private network. VPNs basically route all your traffic to the connected network, which has a lot of benefits.

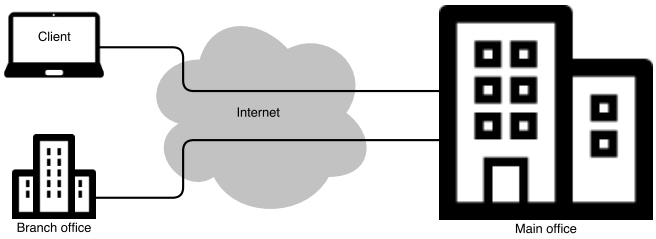

Figure 2.7: Basic overview of remote access using a VPN

One of the benefits is to access files on the remote network which can normally not be accessed, because your computer will act like it is on the same local network as the network that it is connected to. It is also possible to access a business network while travelling or even access your own network.

Another use of a VPN is to hide your browsing activity from your local network or Internet service provider (ISP). This is useful for when you are connected to public WiFi networks that a lot of restaurants offer. If you are using this network and you are browsing websites that do not enforce HTTPS you are visible to everyone that is also connected to the network. When you are connected to a VPN, the only thing they can see is a secured connection, which all your traffic is routed through. While you can use this to hide your browsing activity form your ISP, keep in mind that it is possible that your VPN provider or the server that you are connected to logs your activity.

There are countries that block a large portion of the Internet, examples are The Great Firewall of China [14]. Other sites and software are geographically blocked, for example Netflix or certain games. To circumvent these

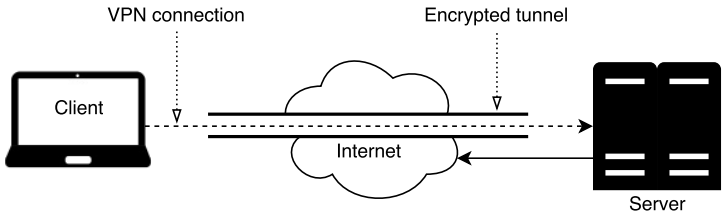

Figure 2.8: Hiding activity using a VPN

types of censorship a VPN is also a great solution. Say a website is only available in the USA, just create a VPN connection to a server in the USA and the website will think you are connecting from within the USA.

2.2 Cryptography

Cryptography is the practise of techniques that allow for secure communication in the presence of an adversary, which is a malicious entity with the main goal of preventing the users of a cryptographic system to achieve their goal. Entities in a cryptographic system are often denoted as Alice (entity A, sender), Bob (entity B, receiver) and Eve (eavesdropper, adversary). Cryptography also includes various terms of information security, which are confidentiality, integrity, availability, and non-repudiation.

2.2.1 Information security

Confidentiality ensures that private information does not fall in the wrong hands, while making sure that the right target can read it. This means that access to the private information must be restricted to those who are allowed to access it. In terms of cryptography this means that when Alice sends a message to Bob, Bob should be able to read it while Eve has no way of accessing it. Data encryption is a regular method of ensuring confidentiality.

Integrity involves that the data that is sent between two entities is consistent, accurate and trustworthy. Data that is sent must remain unaltered by unauthorised parties. In terms of cryptography this means that when Alice sends a message to Bob, Eve can not alter this message without Bob knowing. Ways of ensuring integrity are message authentication codes (MAC) and signatures.

In order for an information system to fulfil its goal, its information must be available when it is required by entities. This covers the concept of availability. Availability ensures that when an entity sends data to another, this data transfer will not be interrupted or stopped. In terms of cryptography this means that when Alice sends a message to Bob, there is no way for Eve to stop this from happening. In order to provide availability one must be

protected against denial-of-service (DoS) attacks by means of firewalls and proxy servers, for example.

Finally there is non-repudiation. Non-repudiation is the fact that when data is transmitted, neither party can deny the fact that it is both sent and received towards anyone. In terms of cryptography this means that when Alice sends a message to Bob, Alice cannot deny that she has sent the message and Bob cannot deny that he has received the message. Non-repudiation can be provided by cryptographically signing messages.

In this thesis we will only look at confidentiality and integrity. To get a better understanding about these terms we will look at encryption of data. Specifically secret-key (also known as symmetric-key or shared-key) cryptography and public-key cryptography.

2.2.2 Secret-key cryptography

Secret-key cryptography is cryptography where two entities use the same pre-shared key for encryption as well as decryption of data. The keys represent the shared secret between two entities used to provide them with a private secured data link. Figure 2.9 shows how encryption and decryption works in the protocol.

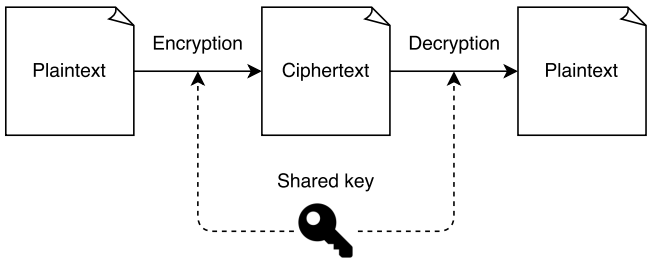

Figure 2.9: Secret-key cryptography

Secret-key encryption and decryption can either be done using stream ciphers or block ciphers. Stream ciphers encrypt data per digit, usually per bit, one at a time. Block ciphers however divide a message in data blocks, where too short blocks are padded by adding extra data, and encrypts the message per data block.

Currently the standard for secret-key cryptography is the Advanced Encryption Standard (AES), which has been approved by the National Institute of Standards and Technology (NIST) in December 2001. An example of a stream cipher is Salsa20 [6], which is recommended by eSTREAM, a project was a intended to promote the design of efficient and compact stream ciphers suitable for widespread adoption.

Using these ciphers, it is possible to create a Message Authentication Code (MAC). A MAC is a cryptographic checksum that is generated using

a session key, which means it requires a key and an arbitrary-length message in order to generate the MAC. The MAC is then sent with the message to the receiver, which can then verify the authenticity of the message using the MAC and the session key.

2.2.3 Public-key cryptography

Public-key cryptography (also known as asymmetric-key cryptography) is based on the use of cryptographically generated keypairs. A keypair includes one public key, which is widely distributed and available publicly, and one private key, which is secret. Like secret-key cryptography, public-key cryptography also provides both confidentiality and integrity. In public-key cryptography, if Alice wants to send a message to Bob securely, she encrypts her message with Bob's public key and sends the ciphertext to Alice. Alice then decrypts the ciphertext using her own private key. This is shown in Figure 2.10.

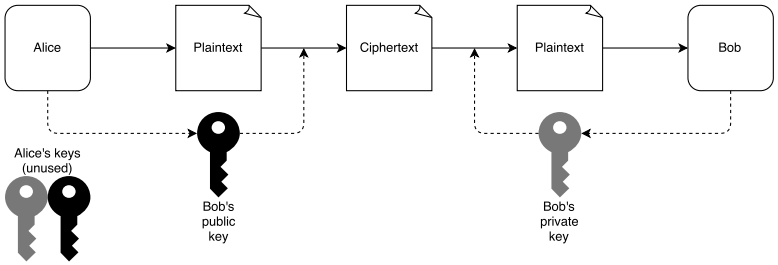

Figure 2.10: Public-key cryptography

When Alice encrypts her message with Bob's public key she indicates that the message is meant for Bob. Bob is then the only one that can decrypt the message because he has the private key that pairs his public key. Public key cryptography also aids symmetric-key cryptography by public-key exchanges. One of the most notable public-key exchange algorithm is the Diffie-Hellman algorithm. This algorithm works like shown in Figure 2.11.

Alice and Bob agree to two public prime numbers g and p. Then each party uses its private key, a for Alice, b for Bob, and calculate $g^a \mod p$ and $g^b \mod p$. They exchange these calculated values. Then Alice uses her private key on Bob's calculated value and Bob uses his private key on Alice's calculated value, so Alice calculates $(g^b \mod p)^a \mod p$ and Bob calculates $(g^a \mod p)^b \mod p$.

Now they both get the same common secret key for future use because $(g^b \mod p)^a \mod p = (g^a \mod p)^b \mod p = g^{ab} \mod p$. The most important part about this is when the calculated value gets exchanged, because it is impossible for Eve to derive the common secret key because it is com-

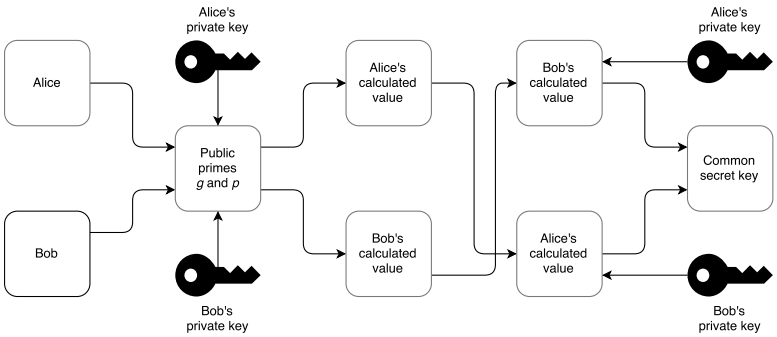

Figure 2.11: The Diffie-Hellman protocol

putationally difficult to calculate this because of the very large values that are used, also known as the discrete logarithm problem.

2.2.4 NaCl

One of the most important aspects of NaClShuttle is its security, which is based on NaCl (pronounced *salt*) [7]. NaCl stands for Networking and Cryptography library, which is an easy-to-use high-speed software library for network communication, encryption, decryption, signatures, etc., developed by D. J. Bernstein, T. Lange, and P. Schwabe.

The library supports C and C++ and is planning to support Python in the near future. It provides functionalities such as public-key cryptography and signatures, secret-key cryptography and authentication and low-level functions like hashing and string comparison.

For secret-key cryptography, NaCl uses Salsa20 [6], which created by Bernstein. Salsa20, also known as Snuffle 2005, is a stream cipher that maps a 256-bit (or 128-bit) key, 64-bit nonce and 64-bit stream position to a 512-bit block of the key stream, giving it the advantage to efficiently seek to any position in the key stream in constant time, making Salsa20 consistently faster than AES. The 8-round Salsa20/8, 12-round Salsa20/12 and 20-round Salsa20/20 are supported in NaCl, where fewer rounds mean faster speed. Currently the best known attack [3] on Salsa20 breaks anything up to the 8-round variant of Salsa20.

For secret-key authentication NaCl uses Poly1305 [4], also created by Bernstein. Poly1305 is a state-of-the-art cryptographic MAC used to verify data integrity and the authenticity of a message. Poly1305 computes a 16-byte authenticator of a variable-length message, using a 16-byte key, an 16-byte additional key, and a 16-byte nonce. Bernstein guarantees that Poly1305 is secure as long as AES is secure. Poly1305 offers consistent high speed on many different CPUs, not just a single one.

As core security features NaCl provides protection against timing at-

tacks. The library has no data flow from secrets to load addresses and branch conditions to protect against timing and replay attacks and no patterns can be deducted responses from sending millions of messages to the library. It is also centralising randomness by reading bytes from the operating system kernel's cryptographic random-number generator. This randomness is only used at places where it is actually necessary, avoiding unnecessary randomness. Instead, NaCl uses deterministic cryptographic operations as much as possible, meaning that their outputs are determined entirely by their inputs.

Another cryptographic disaster is that even though some cryptographic functions are perfectly secure, cryptographic performance issues might cause users to reduce their cryptographic security levels or even turn them off. To prevent disasters like this NaCl provides extremely fast speeds in comparison to other libraries like OpenSSL. NaCl is also capable of automatic CPU-specific tuning that, instead of trying to recognise CPUs and select implementations of certain functions based on that CPU, compiles all implementations of a function and checks which one performs best on a certain CPU.

NaCl also takes into account which cryptographic primitives are potentially dangerous. This is done by paying attention to cryptanalysis. Researching extensive cryptanalytic literature describing the limits of attacks on cryptographic primitives results in higher confidence in NaCl's cryptographic primitives and pushes NaCl to extremely high speeds, which prevents other disasters described above.

Chapter 3

Related Work

Different types of VPN systems are available. It is possible to classify a VPN system by its protocols, tunnel termination point, connectivity (e.g. host-to-network or network-to-network), level of security, the TCP/IP protocol stack layer on which they operate and the number of simultaneous connections.

That said, it is no surprise that there are a lot of different VPN types out there. In this section we will reduce the more general types, describing on what protocols most VPNs are based on and giving examples of the most popular implementations. These types also have many different implementations so some example will be given, but keep in mind that there are more solutions available.

3.1 Internet Protocol Security

Internet Protocol Security (IPsec) [18] is a protocol suite that offers encryption and authentication of data packets, in this case IP packets. IPsec was initially developed for Internet Protocol version 6 (IPv6) [15] and is now a recommendation for both IPv6 and IPv4. IPsec can provide authentication, integrity and confidentiality for a communication session, depending on how it is configured. It specifies three protocols: Encapsulating Security Payloads (ESP), Authentication Headers (AH) and Security Association (SA).

3.1.1 Encapsulating Security Payload

Say an original IP packet consists of the IP header, the transport protocol (in this case Transmission Control Protocol) and the data of the packet. The IP header includes important information about the packet like the source and destination address. The basic structure of an IP packet is shown in Figure 3.1.

ESP protects the contents of a message, providing confidentiality and optionally protects against data tampering, providing authentication and

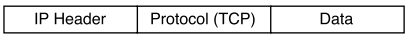

Figure 3.1: A normal IP packet

integrity. When ESP authentication is used, an ESP header is inserted between the IP header and data of a packet, providing authentication. This provides authentication for the original IP packet, however, only the original data of the IP packet is encrypted. Figure 3.2 gives an overview of the new packet that is created.

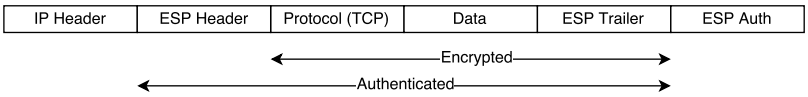

Figure 3.2: A IP packet with IPsec ESP in tunnel mode

3.1.2 Authentication Header

AH protects against data tampering, providing authentication and integrity. It uses the same structure of ESP, however, it also protects against replay-attacks. As confidentiality is not mentioned, AH does not protect the data in a packet against sniffing. If a packet using only AH is captured, the original message can be read. The packet that is created when AH is used is shown in Figure 3.3.

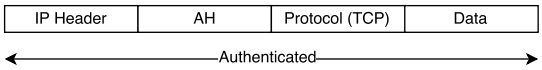

Figure 3.3: A IP packet with IPsec AH in tunnel mode

3.1.3 Security Associations

SAs establish links between two entities using the protocols mentioned above. An IPsec tunnel typically exists of two undirected SAs. IPsec can operate in two modes, called *transport mode* and *tunnel mode*. Transport mode only alters the payload of a packet while the IP header remains unchanged. The packet then becomes the payload of another packet with the same IP header as the original packet. In tunnel mode the entire packet is encrypted and becomes the payload of a new packet with a new IP header containing the addresses of the two IPsec gateways, as shown in Figure 3.4. As you might have noticed, this is perfect for a VPN.

Typically a network structure has a range of private Local Area Network (LAN) IP addresses connected to a gateway, this gateway is then connected

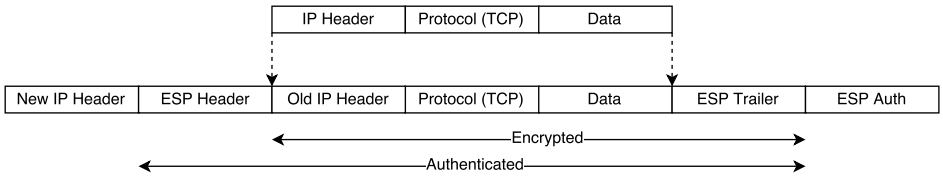

Figure 3.4: A IP packet with IPsec SA in tunnel mode

with the Internet using its public Wide Area Network (WAN) IP address, shown in Figure 3.5.

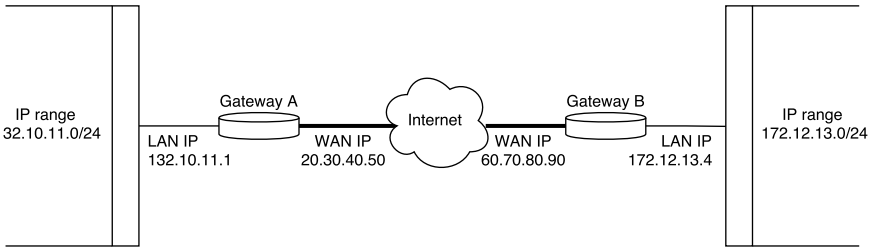

Figure 3.5: Network interface addressing example

In case of an SA, two entities which can be networks, PCs, routers, firewalls or in this case gateways, are configured to have information about each other using IPsec's Internet Key Exchange (IKE). This way the SAs can be used to establish a secure tunnel. In Figure 3.6 is shown which part of the network is replaced by the VPN tunnel.

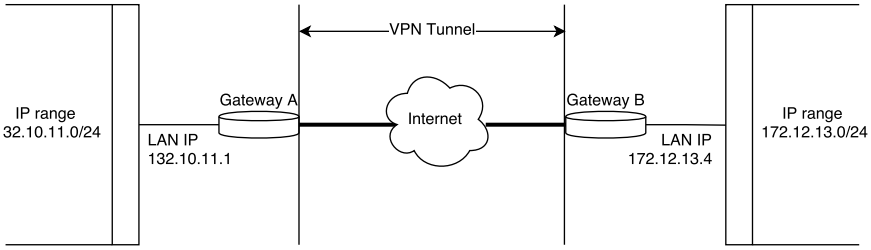

Figure 3.6: Network interface addressing example with VPN tunnel

The VPN tunnel has all information necessary to traffic data securely and encrypted from one entity to another, in the case of Figure 3.6 from Gateway A to Gateway B. Because of this all the entities connected to the gateway do not have to worry about entering all credentials needed to make use of the VPN tunnel. For a gateway to obtain this information an IKE must take place. Alternatively, all the IPsec values can be manually configured on both gateways.

In an IKE session entity A initiates the process in an attempt to connect to entity B. When entity B receives this initiation, the first phase,

authentication, of IKE starts. In this phase the two parties first negotiate which encryption and authentication algorithms to use in the IKE. After the algorithms are determined the parties authenticate each other using the mechanism previously agreed upon. Finally, a master key is generated using the Diffie-Hellman [17] public key algorithm.

In the second phase, negotiation, the parties negotiate which encryption and authentication algorithms to use in the IPsec SAs. The master key that was previously agreed upon is now used to generate the IPsec key for each SA. Once these keys are created and each SA has received its corresponding key, both SAs have all tools needed to protect data between the two VPN gateways.

Once the two phases are completed the secure data transfer can take place. During this time all data that travels between gateway A and gateway B is properly encrypted using the keys agreed upon in the IKE phases. The SAs are terminated when either one of the two gateways gives a termination signal or the session is timed out.

3.1.4 IPsec implementations

IPsec has multiple implementations, aiding in making it easier to use in practice. One of the most straightforward implementations is OpenSWAN [13]. OpenSWAN is an open-source implementation of IPsec, usually found by default on most Linux distributions. OpenSWAN is a code fork of the FreeS/WAN [22] project. Once installed, OpenSWAN provides a file `/etc/ipsec.conf` that can be configured according to the wishes of the user, making it easy to connect two entities using IPsec.

Another implementation that is also widely used is StrongSWAN [34]. StrongSWAN was initially also based on FreeS/WAN, but since a new IKE daemon was written in an object-oriented coding style the project does not share code with its ancestor anymore. The main differences between OpenSWAN and StrongSWAN is that StrongSWAN has a better and more detailed documentation, has support for Extensible Authentication Protocol (EAP) authentication methods and is cross-platform while OpenSWAN supports more hardware cryptography accelerators than StrongSWAN.

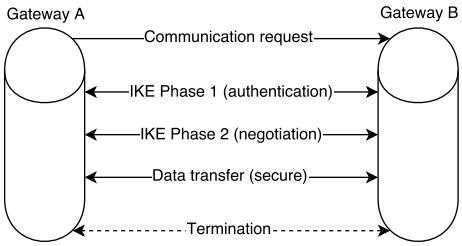

Figure 3.7: IPsec SA Internet Key Exchange

3.2 OpenVPN

OpenVPN [19] is an open-source application by James Yonan, developed by OpenVPN Technologies Inc., that creates secure host-to-host or network-to-network connections using VPN techniques like tunneling using a TUN (layer three) or TAP (layer two) device. It is based on a security protocol that, even though TLS is capable of tunneling the entire traffic of a network through a tunnel, utilises TLS for key exchange only. It allows users to authenticate using symmetric key cryptography, certificates or a username and password combination. For encryption OpenVPN heavily relies on OpenSSL, allowing it to use their ciphers.

OpenVPN multiplexes the SSL/TLS session used for authentication and key exchange with the actual encrypted tunnel data stream [25]. Transport Layer Security (TLS) [16] and its predecessor Secure Sockets Layer (SSL) [11] are also cryptographic protocols that provide security for communications. A TLS communication is private as the connection is encrypted using symmetric cryptography where each connection is secured by a shared secret agreed upon with a TLS handshake [30]. The connection is authenticated because public-key cryptography [17] used in the connection ensures that the connection is only used by owners of the private keys and provides integrity because it checks the message with a Message Authentication Code (MAC) to prevent loss or tampering.

Transportation of the traffic is then done over User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) using a TUN or TAP device [26] for tunneling. OpenVPN can be run with either static encryption or Public Key Infrastructure (PKI). Setting up OpenVPN with PKI requires a Certificate Authority (CA). Both client and server will need a private key, certificate and CA certificate, these are used for encryption.

It is then possible to use TLS authentication, where each packet sent to OpenVPN will have a signature. OpenVPN then calculates a signature and compares it with the one of the packet. If the signatures do not match the packet is dropped. It is also possible to use username and password authentication.

3.3 Secure Shell

Secure Shell (SSH) [37] is another cryptographic network protocol used for executing network pursuits securely over unsecured networks. SSH is mainly used to remotely log into a different computer or server by using a client-server protocol. The connection is secured by public-key cryptography and the connection can be used using the automatically generated public-private key pairs or simply by password authentication.

An example of an application utilising SSH for a VPN connection is

sshuttle [32], also known as "the poor man's VPN". Sshuttle creates a VPN connection, using SSH to connect the client to the server. In contrary to a lot of other free VPN services, which can lead to a lot of risk and hassle by for example advertisements or logging events, sshuttle is an free open-source application which everyone using Linux or MacOS can use.

As long as a remote server has Python 2.3 [20] or higher installed and a Secure Shell daemon running, sshuttle allows you to create a VPN connection to that remote server using SSH. For this to work you need to have root access on your own local machine but no root access is needed on the server, except to start the Secure Shell daemon. It is possible to run sshuttle multiple times to different servers to gain access to multiple servers at once. Sshuttle can even be run on a router, forwarding the traffic of an entire network to the VPN.

Sshuttles synopsis looks as follows:

```
sshuttle [options] [-r [username@]sshserver[:port]] <subnets>
```

Let's say we want to route our entire traffic to a server with domain name `example.org` where we have an user account called `test`. In this case running following command will make that happen:

```
sshuttle -r test@example.org 0.0.0.0/0
```

If you would also like to have your DNS queries to be routed to the DNS server of the server you are connecting to you can use the optional `--dns` parameter like:

```
sshuttle --dns -r test@example.org 0.0.0.0/0
```

When run, sshuttle creates an ssh session to `test@example.org`, which was specified after `-r`. If `-r` is omitted, both the client and server will start locally, which is useful for testing. Sshuttle then uploads and executes its source code to the remote server, meaning that the remote server does not need to have sshuttle installed nor will version issues take place.

Because we specified `0.0.0.0/0`, the traffic of the entire network will be routed through the tunnel, essentially creating a VPN. It is also possible to provide a subnet like `12.34.56.78` (a single IP address) or `12.34.56.78/24` (a subnet with a `225.225.225.0` submask), causing only the traffic of this specific subnet to be routed through the tunnel.

3.3.1 Lines of code

We have now described various implementations of VPN solutions using multiple protocols like IPsec, SSL/TLS and SSH. To give a very global overview of the complexity of these projects we will compare the lines of code used in the project. We use the statistics of Open Hub [12] to determine

the lines of code as of June 2016, provided on GitHub [23]. The results can be found in Figure 3.8.

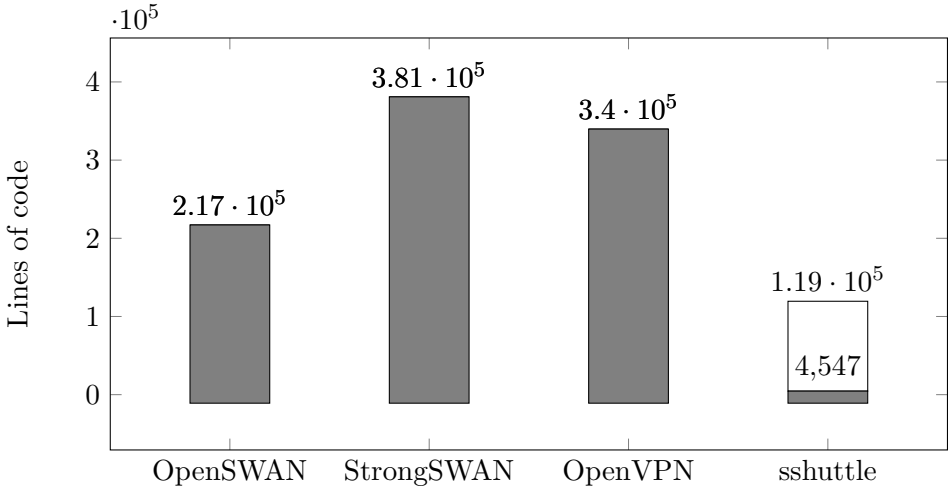

Figure 3.8: A bar graph of the lines of code of various VPN implementations

Open-source solutions of big companies involve many lines of code which makes the project very complex. As we can see, sshuttle, which is developed by several individuals, is massively smaller than the other projects. However, approximately 4500 lines of code is small, maybe even small enough to verify the code for security, but the lines of code is not the only aspect that can cause security issues because it also relies on SSH, which is shown by the white bar.

Chapter 4

Our solution: NaClShuttle

From all the VPN systems and applications that are available, the ones mentioned in the previous chapter are among the most popular ones. They all work exactly as they are supposed to, so one might wonder why we need yet another program that tunnels our traffic to a remote location.

The problem with IPsec is that it is incredibly difficult to set up correctly, leading to security risks as a result of wrongful implementations or minor mistakes. In addition to this problem there is also the fact that IPsec quickly runs into trouble when it has to deal with Network Address Translation (NAT) or firewall rules [1].

Luckily, OpenVPN uses TLS which does not have problems with NAT or firewalls. However, as mentioned in the previous chapter, OpenVPN heavily relies on OpenSSL for its cryptographic procedures, which is bad because OpenSSL currently still has had a lot of security vulnerabilities (for example [36] [2] and [35]), making OpenVPN insecure in its entirety as long as these vulnerabilities are not fixed.

Then there is sshuttle, which does a pretty good job so far. The only problem is that it still relies on SSH for connections to remote servers. This means that the user still relies on a very large complex structure for its connection. While SSH a better security track record than SSL, it is still suspicious after the files Edward Snowden leaked [31], that revealed that organisations like the National Security Agency (NSA) might be able to decrypt SSH traffic.

Because of these reasons it is clear that there is still a need for a secure open-source VPN application that is minimal and independent that it can be audited and verified by third parties without having to compromise security.

4.1 NaClShuttle

The problems mentioned in the introduction of this chapter led to the creation of NaClShuttle (pronounced *salt shuttle*). NaClShuttle is designed

with security and a minimal trusted code base (TCB) as number one priority. The goal of the project is a very lightweight program which is as small as possible that can deliver a fully functional VPN tunnel to route traffic from one entity to another. Its source code can be found in appendix A.

The total package of NaClShuttle consists of just 1,013 lines of code while a proper code standard is respected. This includes all cryptographic libraries that are used by the tunnel to securely route traffic and the setup script. The package is about 183 KB in total, making the program very lightweight.

NaClShuttle is based on OTPTunnel [24], mainly using its tunnel mechanism to traffic data through the tunnel correctly. All one time pad encryption is stripped and replaced by TweetNaCl, which will be discussed later.

4.1.1 Requirements

NaClShuttle is designed to require as little requirements as possible. To use NaClShuttle, one must have a Unix-based system with root access and Python 2.7 or above installed.

4.1.2 Installation

Installing NaClShuttle is fast an easy. To install NaClShuttle, first clone the repository to your system by running the following command:

```
git clone git@sandor.cs.ru.nl:naclshuttle.git
```

After the repository is cloned you will have four files on your system: `naclshuttle`, `tunnel.py`, `setup.sh` and `tweetnacl.so`. The files `naclshuttle` and `tunnel.py` are the actual program while `tweetnacl.so` includes the cryptographic library used to make NaClShuttle secure. To set up the program on your system simply run

```
sudo ./setup.sh <parameter>
```

with either `-C` or `-S` as `parameter` for client or server mode, respectively. NaClShuttle is built in a way only requires root access once for the program, for an easy one-time setup. After this is done the tunnel is ready to be used in user space for as many times as the user desires. In server mode the `setup.sh` file does the following:

- **Enable port forwarding:** This enables packets with a certain address to be forwarded to that address if they are directly connected or available in the routing table.

- **Configure iptables:** By setting POSTROUTING chain of the NAT table to MASQUERADE it allows to send responses of incoming requests back to the correct address. This is also described in Section 2.1.3.
- **Create a TAP device:** This device is later used by the program to route traffic through.
- **Configure the TAP device:** This assigns correct addresses to the TAP device so that it can be used properly.

In client mode the `setup.sh` file does the following:

- **Create a TAP device:** This device is later used by the program to route traffic through.
- **Configure the TAP device:** This assigns correct addresses to the TAP device so that it can be used properly.
- **Adjust the routing table:** This way the TAP device is set as default gateway so that all traffic routes through the TAP device.

4.1.3 Usage

After NaClShuttle is installed and set up on the client and server side, both parties need a keyfile in order to encrypt the data sent over the tunnel. This keyfile is a pre-shared key which is owned by both the server and client side. After this is completed the program can be run. This is done on the server side by running

```
./naclshuttle -S -K <keyfile>
```

where `keyfile` is the path to the keyfile. The client can then connect to the server by running

```
./naclshuttle -A <address> -K <keyfile>
```

where `address` is the (IP) address of the server that is running NaClShuttle in server mode and `keyfile` is the path to the keyfile. If everything runs correctly the server side will show a message stating which IP address is connected to the program. All data is now routed through the TAP device from the client to the server and responses are routed back to the client through the TAP device.

4.1.4 How it works

When running NaClShuttle, the `naclshuttle` file handles all arguments given to the program. Mandatory arguments are `-A` or `-S` for client or server

mode, respectively and `-K` for the keyfile. If no other arguments are given, all other arguments will be given the default value. The other arguments and their default values are as follows:

- Set the remote port (`-P`), default: 12000
- Set TAP device local address (`--tap-addr`), default: 10.8.0.1 (server mode), 10.8.0.2 (client mode)
- Set TAP device netmask (`--tap-netmask`), default: 24
- Set TAP device maximum transmission unit (MTU) (`--tap-mtu`), default: 32768
- Set the address to which the socket will bind (`--local-addr`), default: 0.0.0.0
- Set the port to which the socket will bind (`--local-port`), default: 12000

The program then collects all arguments and checks whether the server argument, `-S`, is set. If the argument is not set it will set the value of the TAP device address to 10.8.0.2. After that the program runs the Python file `tunnel.py` with the given arguments.

The `tunnel.py` file starts with the constructor that initialises variables with the arguments given. The constructor also connects to the TAP device and if the program runs in server mode it will listen to the socket for any new message from the TAP device. If a message is received, the address of the sender is saved and set as return address for response traffic. If the program is in client mode it will create a new socket, different from the one that is bound to 0.0.0.0:12000 by default, to send an initialisation packet to the server side, handled as described above.

When NaClShuttle is properly initialised, it continues to the main loop of the program. Before entering the loop, it creates a variable called `files` containing the TAP device (`self._tap`) and the socket (`self._sock`). This variable is used by Python's `select` module [21], which waits for input and output completion, in the following line of code:

```
r, w, x = select.select(files, files, [])
```

Every iteration of the main loop this line of code is executed. The `select()` method takes as arguments three lists. The method waits until these lists are ready to be read, written to and until an exceptional condition occurs, respectively. The return value of the method is a triple that contains lists of objects that are ready, subsets of the three arguments. Since we don't need the exceptional condition argument we pass an empty list to this argument.

Using the returned lists the case distinction will take place, which forms the most important part of the program. The code case distinction looks like the following:

```
if self._tap in r:
    to_sock = os.read(self._tap, mtu)

if self._sock in r:
    to_tap, addr = self._sock.recvfrom(65535)
    key = 'ThisKeyIsNotSoSecretThisKeyIsNot'
    nonce = 'NonceNonceNonceNonceNonc'
    to_tap_decrypted = nacl.crypto_secretbox_open(to_tap,
        nonce, key)
    to_tap = to_tap_decrypted

if to_tap and self._tap in w:
    os.write(self._tap, to_tap)
    to_tap = None

if to_sock and self._sock in w:
    key = 'ThisKeyIsNotSoSecretThisKeyIsNot'
    nonce = 'NonceNonceNonceNonceNonc'
    to_sock_encrypted = nacl.crypto_secretbox(to_sock, nonce,
        key)
    to_sock = to_sock_encrypted
    self._sock.sendto(
        to_sock, (self._remote_address, self._remote_port))
    to_sock = None
```

There are four different cases in the loop. The first case checks if the TAP device is ready to be read from. If this is true the data from the device is stored for later use in a variable called `to_sock`.

The second case checks if the socket is ready to be read from. If this is true the data and address are saved in the `to_tap` and `addr` variables, respectively. The message on the TAP device is encrypted using TweetNaCl's `crypto_secretbox_open()` method, which will be described later in this chapter, and must therefore be decrypted using the keyfile and nonce which the client and server agreed upon. The decrypted data is then stored for later use in a variable called `to_tap`.

The third case checks if the `to_tap` variable is not empty and if the TAP device is ready to be written to. If this is true then the `to_tap` data is written to the TAP device. The `to_tap` variable is emptied afterwards.

The last case checks if the `to_sock` is set and the socket is ready to be written to. If this is true the `to_sock` data will be encrypted using Tweet-

NaCl's `crypto_secretbox()` method using the keyfile and nonce which the client and server agreed upon. The encrypted packet will then be sent to the socket, which sends it to the remote address. The `to_sock` variable is emptied afterwards.

The main loop also checks for errors using Python's `try` and `except` functionality. If an error occurs of the type *interrupted system call* the program will skip the iteration and continue. Any other error will stop the program. Keyboard interruptions like `Ctrl + C` also end the program.

To summarize this, what NaCl basically does is first setting up routing as a root user using the `setup.sh` script and setting up parameters and addresses so the program knows where to route its traffic to. Then it starts running, which is basically the main loop. The client sends all its traffic to the TAP device. When NaClShuttle detects that there is a packet ready to be read on the TAP device, it takes this packet, reads the destination, encrypts it using NaCl and then sends it to the socket. On the server side the program waits for messages to arrive on the socket and, when a packet is available, receives packets, decrypts them and sends them to the TAP device. When there is a response available it will be directed to the TAP device because of the settings in `iptables`, and then the whole process goes in the other direction. An overview of these steps can be found in Figure 4.1.

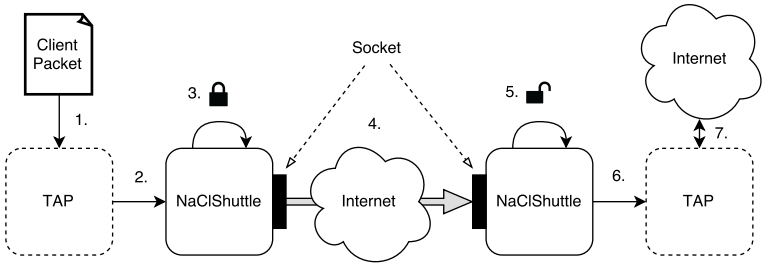

Figure 4.1: An overview of how NaClShuttle works

4.2 TweetNaCl

Even though NaCl is great in its own form, the purpose of NaClShuttle is to have an as small as possible TCB to provide minimal room for error and maximal transparency. While we are importing an entire library, NaCl is not qualified for this case. Luckily there is a variant of NaCl called TweetNaCl (again, pronounced *tweet salt*) [10]. According to their own website [8]: "TweetNaCl is the world's first auditable high-security cryptographic library." Meaning that TweetNaCl is not only easily readable, but also ready to be audited against a mathematical description of the function-

ality in NaCl. This makes it possible to audit the complete cryptographic part of the trusted code base of NaClShuttle.

To illustrate how small TweetNaCl is, the developers have tweeted [9] the entire source code of the program in just 100 tweets, while maintaining support for all 25 C NaCl functions used by applications. The tweeted code can also be found in Appendix A. The tweeted file, `tweetnacl.c`, originally has 809 lines of code. To put this into perspective, OpenSSL, a different well-known open-source cryptographic library, contains around 460.000 lines of code.

Additionally, since NaClShuttle is a Python program, TweetNaCl is used in Python form by means of a Python wrapper around the C implementations of TweetNaCl. The specific wrapper used is Python-TweetNaCl [29], created by Mojžíš. Since NaClShuttle uses secret-key cryptography, the NaCl function used for encryption, as shown in Section 4.1.4, is

```
c = crypto_secretbox(m,n,k)
```

where `c` is the ciphertext, `m` is the message, `n` is the 24-byte nonce and `k` is the 32-byte secret key agreed upon by the two entities using the function. On the receiving side, the function that is used for decryption is

```
m = crypto_secretbox_open(c,n,k)
```

where `m` is the original message, `c` is the received ciphertext, `n` is the nonce and `k` is the secret key. To make things easier, the naming convention of these two functions are consistend in both TweetNaCl and Python-TweetNaCl. The functions are designed to meet current standards for privacy and authenticity. The cryptographic primitive used for this function is a combination of Salsa20 for secret-key encryption and Poly1305 for authentication.

Chapter 5

Conclusions and future work

NaClShuttle is a program that perfectly fills the gap in currently available VPN solutions. A gap created by many complicated VPN solutions. Problems include: Hard to use, too many lines of code, slow performance or reliant on too many libraries. To illustrate why NaClShuttle perfectly fills this gap we will quickly summarise the strong aspects of the tool.

First of all there's the simplicity of the NaClShuttle. Out if the box it is very easy to install and use: If the files are on your device and that of the server all you have to do is run four simple commands to route all your traffic through an encrypted tunnel. Unlike other solutions like IPsec there is no need for any configuration or setup. Everything that is essential is done by NaClShuttle itself, therefore leaving no room for the user to make mistakes.

Then there is the fact of how minimal NaClShuttle is in terms of size. With just 1,013 lines of code and 183 KB in size, including the cryptographic library used to securely encrypt all traffic, NaClShuttle is probably the smallest secure VPN solutions available today. If we add NaClShuttle, including TweetNaCl, in the bar graph of Section 3.3.1, the bar of the complete NaClShuttle package would be so small that it looks like a line on the null coordinate. To give a better illustration, we compared NaClShuttle to sshuttle *without* SSH. This bar graph can be found in Figure 5.1. Being small also makes it easier for third parties to audit the software to confirm its security.

Finally, NaClShuttle is secure. NaClShuttle uses TweetNaCl, which is a minimal version of NaCl. NaCl uses state-of-the-art protocols of cryptographic primitives that are considered secure at the moment. Like NaClShuttle, NaCl also comes with the desired features of simplicity, speed and security.

That said, NaClShuttle does have room for improvement. NaClShuttle is a minimal VPN solution that is well suited for the job, however, in its current form it is still quite unpredictable from time to time. It is for example

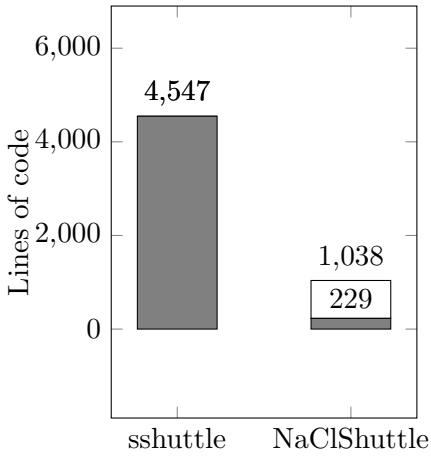

Figure 5.1: NaClShuttle incl. TweetNaCl compared to sshuttle excl. SSH

possible to disconnect from the server while the server keeps running, how the server handles new connection attempts is unsure and must be improved.

Currently root access is required for NaClShuttle to run, which is not an ideal situation. Generally one would want to separate the user and root logic of a program as much as possible. Running software in user-space is safer in general because it denies access to critical and sensitive information of a device. This might be improved in the future by replacing lines of code that require root-access with alternatives and putting all root-required lines in a separate file for an easier overview of the part of NaClShuttle that requires root.

Another improvement is that currently NaClShuttle uses only a few functions of TweetNaCl, while TweetNaCl is imported in its entirety. To improve this it is possible to remove the functions that NaClShuttle does not need from the library so it runs with even less lines of code. As the TweetNaCl library is 80% of the lines of code and 95% of the size of NaClShuttle, this would make a great impact.

The current form of encryption works fine and the only way to decrypt messages is if the pre-shared key is obtained. To improve the encryption further it is possible to implement Perfect Forward Secrecy (PFS). If an attacker managed to get hold of a key he is able to decrypt all previous messages sent with this key. This involves creating a session key each time a new session is started.

Implementing PFS is possible because TweetNaCl supports public-key cryptography using Curve25519 [5] in addition to Salsa20 and Poly1305. Curve25519 is a high-security elliptic-curve-Diffie-Hellmann function, achieving extremely fast speeds. It has benefits like free key compression and validation, and state-of-the-art timing-attack protection.

To summarise, if it comes to a secure VPN solution that has minimal

code and a TCB that is as small as possible, is fast enough for daily use and is usable by the common user, NaClShuttle beats most other solutions in multiple aspects, making it a go-to application for daily use as secure VPN.

Bibliography

[1] Bernard Aboba and William Dixon. IPsec-Network Address Translation (NAT) compatibility requirements. `https://tools.ietf.org/html/rfc3715`, 2004. Accessed 07-02-2017.

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, pages 185–203. Springer, 2007.

[3] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of Latin dances: analysis of Salsa, ChaCha, and Rumba. In *International Workshop on Fast Software Encryption*, pages 470–488. Springer, 2008.

[4] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *Fast Software Encryption*, volume 3557 of LNCS, pages 32–49. Springer, 2005.

[5] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records - pkc 2006. In *Public Key Cryptography*, volume 3958 of LNCS, pages 207–228. Springer, 2006.

[6] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, volume 4986 of LNCS, pages 84–97. Springer, 2008.

[7] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of LNCS, pages 159–176. Springer, 2012.

[8] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl. `http://tweetnacl.cr.yp.to/`. Accessed 07-02-2017.

[9] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl on Twitter. `https://twitter.com/tweetnacl`. Accessed 07-02-2017.

[10] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*, pages 64–83. Springer, 2014.

[11] Mittal S Bhiogade. Secure socket layer. In *Computer Science and Information Technology Education Conference*, volume 2, pages 85–90. InSITE, 2002.

[12] Inc. Black Duck Software. Open hub. `https://www.openhub.net/`. Accessed 07-02-2017.

[13] Xelerance Corp. OpenSWAN. `https://www.openswan.org/`. Accessed 07-02-2017.

[14] Orlando Crowcroft. Behind the Great Firewall, China is winning its war against internet freedom. `http://www.ibtimes.co.uk/behind-great-firewall-china-winning-its-war-against-internet-freedom-1558550`. Accessed 07-02-2017.

[15] Stephen E Deering. Internet protocol, version 6 (IPv6) specification. `https://tools.ietf.org/html/rfc2460`, 1998. Accessed 07-02-2017.

[16] Tim Dierks. The transport layer security (TLS) protocol version 1.2. 2008.

[17] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[18] Naganand Doraswamy and Dan Harkins. *IPsec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

[19] Markus Feilner. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd, 2006.

[20] Python Software Foundation. Python. `https://www.python.org/`. Accessed 07-02-2017.

[21] Python Software Foundation. Python select module. `https://docs.python.org/2/library/select.html`. Accessed 07-02-2017.

[22] John Gilmore. FreeS/WAN. `http://freeswan.org/`. Accessed 07-02-2017.

[23] Inc. GitHub. Github. `https://github.com/`. Accessed 07-02-2017.

[24] Robert Graham. OTPTunnel. `https://github.com/rpgraham84/otptunnel`. Accessed 07-02-2017.

[25] OpenVPN Technologies Inc. OpenVPN: Security overview. https://openvpn.net/index.php/open-source/documentation/security-overview.html. Accessed 07-02-2017.

[26] Maxim Krasnyansky and Maksim Yevmenkin. Universal TUN/-TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt, 2002. Accessed 07-02-2017.

[27] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 165–182. ACM, 1991.

[28] Adam Langley. Apple's SSL/TLS bug. https://www.imperialviolet.org/2014/02/22/applebug.html, 2014. Accessed 07-02-2017.

[29] Jan Mojžíš. Python-TweetNaCl. https://mojzis.com/software/python-tweetnacl/. Accessed 07-02-2017.

[30] Paul Morrissey, Nigel P Smart, and Bogdan Warinschi. A modular security analysis of the TLS handshake protocol. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 55–73. Springer, 2008.

[31] Spiegel Online. Inside the NSA's War on Internet Security. http://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html, 2014. Accessed 07-02-2017.

[32] Avery Pennarun. Sshuttle. https://github.com/apenwarr/sshuttle, 2010. Accessed 07-02-2017.

[33] John M Rushby. *Design and verification of secure systems*, volume 15. ACM, 1981.

[34] Andreas Steffen. StrongSWAN. https://www.strongswan.org/. Accessed 07-02-2017.

[35] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+ RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[36] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 15–27. ACM, 2009.

[37] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) protocol architecture. https://tools.ietf.org/html/rfc4251.html, 2006. Accessed 07-02-2017.

Appendix A

Source code

A.1 NaClShuttle

A.1.1 naclshuttle

```
#!/usr/bin/env python
import argparse
import textwrap
import socket
import sys
from tunnel import Tunnel


def main():
    parser = argparse.ArgumentParser(
        formatter_class=argparse.RawDescriptionHelpFormatter,
        description=textwrap.dedent('''\
        A VPN-like server/client that utilizes a user specified
        one time pad for the XOR'ing of network traffic over a TAP
            interface.
        '''),
        epilog=textwrap.dedent('''\

        Examples:

        To start a server listening on default settings,

        naclshuttle -S -K ~/random.bin

        If that server's IP is 192.168.1.1, and you have the same
            keyfile
        in your home directory, you can connect to it using,

        naclshuttle -K ~/random.bin -A 192.168.1.1 --tap-addr 10.8.0.2

        '''))
    parser.add_argument('-S', '--server', action="store_true", dest='
        server',
                        help="set server mode (default: client mode)")
    parser.add_argument('-A', dest='remote_address',
                        help='set remote server address')
    parser.add_argument('-P', type=int, dest='remote_port', default
        ='12000',
```

```
                        help='set remote server port')
    parser.add_argument('--tap-addr', type=str, dest='taddr', default
        ='10.8.0.1',
                        help='set tunnel local address (default:
                            10.8.0.1 for '
                             'server, 10.8.0.2 for client)')
    parser.add_argument('--tap-netmask', default='24', dest='tmask',
                        help='set tunnel netmask (default: 24)')
    parser.add_argument('--tap-mtu', type=int, default=32768, dest='
        tmtu',
                        help='set tunnel MTU (default: 32768)')
    parser.add_argument('--local-addr', default='0.0.0.0', dest='laddr
        ',
                        help='address to which OTPTunnel will bind (
                            default: 0.0.0.0)')
    parser.add_argument('--local-port', type=int, default=12000, dest
        ='lport',
                        help='set local port (default: 12000)')
    args = parser.parse_args()

    if not args.server:
        if args.taddr == '10.8.0.1':
            args.taddr = '10.8.0.2'
        if not args.remote_address:
            parser.print_help()
            return 1
        try:
            client = Tunnel(
                args.taddr, args.tmask, args.tmtu,
                args.laddr, args.lport, args.remote_address,
                args.remote_port)
        except socket.error as e:
            print >> sys.stderr, str(e)
            return 1
        print 'NaClShuttle: Running in client mode, press Ctrl + C to
            cancel.'
        client.run()
        return 0
    else:
        server = Tunnel(
            args.taddr, args.tmask, args.tmtu,
            args.laddr, args.lport, args.remote_address,
            args.remote_port)
        print 'NaClShuttle: Running in server mode, press Ctrl + C to
            cancel.'
        server.run()
        return 0


if __name__ == '__main__':
    sys.exit(main())
```

A.1.2 setup.sh

```
#!/bin/bash
declare error
if [[ $EUID -ne 0 ]]; then
    echo 'NaClShuttle: [ERROR] This script must be run as root.'
    exit 1
fi
function echo_error
{
```

```
    echo -n 'NaClShuttle: [ERROR] '
    echo $1
    error=true
}
function ip_forward
{
    echo 1 > /proc/sys/net/ipv4/ip_forward
}
if [[ $1 ]]; then
    if [ $1 = '-S' ]; then
        echo 'NaClShuttle: Server mode.'
        echo 'NaClShuttle: Setting up network settings...'
        { echo 1 > /proc/sys/net/ipv4/ip_forward; } 2>/dev/null ||
            echo_error 'Failed to enable IP forwarding.'
        iptables -t nat -A POSTROUTING -j MASQUERADE >/dev/null 2>&1
            || echo_error 'Failed to setup iptables.'
        echo 'NaClShuttle: Creating TAP device... '
        ip tuntap add tap0 mode tap > /dev/null 2>&1 || echo_error '
            Failed to create TAP device. Does it already exist?'
        ifconfig tap0 10.8.0.1/24 > /dev/null 2>&1 || echo_error '
            Failed to configure TAP device. Does the TAP device exist
            ?'
        echo -n 'NaClShuttle: NaClShuttle setup completed'
        if [ "$error" = true ]; then
            echo ' with errors, check output above.'
        else
            echo '.'
        fi
    elif [ $1 = '-C' ]; then
        echo 'NaClShuttle: Client mode.'
        echo 'NaClShuttle: Creating TAP device... '
        ip tuntap add tap0 mode tap > /dev/null 2>&1 || echo_error '
            Failed to create TAP device. Does it already exist?'
        ifconfig tap0 10.8.0.2/24 > /dev/null 2>&1 || echo_error '
            Failed to configure TAP device. Does the TAP device exist
            ?'
        echo 'NaClShuttle: Adjusting IP routing table...'
        echo -n 'NaClShuttle: NaClShuttle setup completed'
        if [ "$error" = true ]; then
            echo ' with errors, check output above.'
        else
            echo '.'
        fi
    fi
else
    echo 'NaClShuttle: [ERROR] No parameter given. Use -S for server
        mode, -C for client mode.'
fi
```

A.1.3 tunnel.py

```
import errno
import fcntl
import os
import select
import socket
import sys
import struct
import threading
import tweetnacl as nacl
```

```
class Tunnel(threading.Thread):
    def __init__(self, taddr, tmask, tmtu, laddr, lport,
        remote_address, remote_port):
        super(Tunnel, self).__init__()
        self._tmtu = tmtu
        self._sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self._sock.bind((laddr, lport))
        self._remote_address = remote_address
        self._remote_port = remote_port

        if not remote_address:
            try:
                print 'NaClShuttle: Waiting for client...'
                msg, addr = self._sock.recvfrom(65535)
                print 'NaClShuttle: Client (' + addr[0] + ') connected
                    .'
                self._remote_address = addr[0]
            except KeyboardInterrupt:
                print u'\u0008\u0008NaClShuttle: Closed.'
                sys.exit(0)
        else:
            print 'NaClShuttle: Sending initialisation message to ' +
                str(self._remote_address) + ':' + str(self.
                _remote_port)
            init_sock = socket.socket(socket.AF_INET, socket.
                SOCK_DGRAM)
            init_sock.sendto('init', (self._remote_address, self.
                _remote_port))
            os.system('route add default gw 10.8.0.1')

        print 'Connecting to TAP'
        tap = os.open('/dev/net/tun', os.O_RDWR | os.O_NONBLOCK)
        ifr = struct.pack('16sH', 'tap0', 2 | 4096)
        fcntl.ioctl(tap, 0x400454ca, ifr)
        self._tap = tap

    def run(self):
        mtu = self._tmtu
        files = [self._tap, self._sock]
        to_tap = None
        to_sock = None

        while True:
            try:
                r, w, x = select.select(files, files, [])
                if self._tap in r:
                    to_sock = os.read(self._tap, mtu)

                if self._sock in r:
                    to_tap, addr = self._sock.recvfrom(65535)
                    key = 'ThisKeyIsNotSoSecretThisKeyIsNot' # Debug,
                        can be replaced with keyfile
                    nonce = 'NonceNonceNonceNonceNonc'
                    to_tap_decrypted = nacl.crypto_secretbox_open(
                        to_tap, nonce, key)
                    to_tap = to_tap_decrypted

                if to_tap and self._tap in w:
                    os.write(self._tap, to_tap)
                    to_tap = None

                if to_sock and self._sock in w:
```

```
                key = 'ThisKeyIsNotSoSecretThisKeyIsNot' # Debug,
                    can be replaced with keyfile
                nonce = 'NonceNonceNonceNonceNonc'
                to_sock_encrypted = nacl.crypto_secretbox(to_sock,
                     nonce, key)
                to_sock = to_sock_encrypted
                self._sock.sendto(
                    to_sock, (self._remote_address, self.
                        _remote_port))
                to_sock = None
            except (select.error, socket.error) as e:
                if e[0] == errno.EINTR:
                    continue
                sys.stderr.write(str(e))
                break
            except KeyboardInterrupt:
                print u'\u0008\u0008NaClShuttle: Closed.'
                sys.exit(0)
```

A.2 TweetNaCl (as tweeted)

```
#include "tweetnacl.h"
#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void
typedef unsigned char u8;typedef unsigned long u32;typedef unsigned
    long long u64;typedef long long i64;typedef i64 gf[16];extern void
randombytes(u8*,u64);static const u8 _0[16],_9[32]={9};static const gf
     gf0,gf1={1},_121665={0xDB41,1},D={0x78a3,0x1359,0x4dca,0x75eb,0
    xd8ab,
0x4141,0x0a4d,0x0070,0xe898,0x7779,0x4079,0x8cc7,0xfe73,0x2b6f,0x6cee
    ,0x5203},D2={0xf159,0x26b2,0x9b94,0xebd6,0xb156,0x8283,0x149a,0
    x00e0,
0xd130,0xeef3,0x80f2,0x198e,0xfce7,0x56df,0xd9dc,0x2406},X={0xd51a,0
    x8f25,0x2d60,0xc956,0xa7b2,0x9525,0xc760,0x692c,0xdc5c,0xfdd6,0
    xe231,
0xc0a4,0x53fe,0xcd6e,0x36d3,0x2169},Y={0x6658,0x6666,0x6666,0x6666,0
    x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0
    x6666,
0x6666,0x6666},I={0xa0b0,0x4a0e,0x1b27,0xc4ee,0xe478,0xad2f,0x1806,0
    x2f43,0xd7a7,0x3dfb,0x0099,0x2b4d,0xdf0b,0x4fc1,0x2480,0x2b83};
    static
u32 L32(u32 x,int c){return(x<<c)|((x&0xffffffff)>>(32-c));}static u32
     ld32(const u8*x){u32 u=x[3];u=(u<<8)|x[2];u=(u<<8)|x[1];return(u
    <<8)|
x[0];}static u64 dl64(const u8*x){u64 i,u=0;FOR(i,8)u=(u<<8)|x[i];
    return u;}sv st32(u8*x,u32 u){int i;FOR(i,4){x[i]=u;u>>=8;}}sv
    ts64(u8*x,
u64 u){int i;for(i=7;i>=0;--i){x[i]=u;u>>=8;}}static int vn(const u8*x
    ,const u8*y,int n){u32 i,d=0;FOR(i,n)d|=x[i]^y[i];return(1&((d-1)
    >>8))
-1;}int crypto_verify_16(const u8*x,const u8*y){return vn(x,y,16);}int
     crypto_verify_32(const u8*x,const u8*y){return vn(x,y,32);}sv
    core(u8
*out,const u8*in,const u8*k,const u8*c,int h){u32 w[16],x[16],y[16],t
    [4];int i,j,m;FOR(i,4){x[5*i]=ld32(c+4*i);x[1+i]=ld32(k+4*i);x[6+i
    ]=
ld32(in+4*i);x[11+i]=ld32(k+16+4*i);}FOR(i,16)y[i]=x[i];FOR(i,20){FOR(
    j,4){FOR(m,4)t[m]=x[(5*j+4*m)%16];t[1]^=L32(t[0]+t[3],7);t[2]^=L32
    (t[1
]+t[0],9);t[3]^=L32(t[2]+t[1],13);t[0]^=L32(t[3]+t[2],18);FOR(m,4)w[4*
    j+(j+m)%4]=t[m];}FOR(m,16)x[m]=w[m];}if(h){FOR(i,16)x[i]+=y[i];FOR
```

```
    (i,4
){x[5*i]-=ld32(c+4*i);x[6+i]-=ld32(in+4*i);}FOR(i,4){st32(out+4*i,x[5*
    i]);st32(out+16+4*i,x[6+i]);}}else FOR(i,16)st32(out+4*i,x[i]+y[i
    ]);}
int crypto_core_salsa20(u8*out,const u8*in,const u8*k,const u8*c){core
    (out,in,k,c,0);return 0;}int crypto_core_hsalsa20(u8*out,const u8*
    in,
const u8*k,const u8*c){core(out,in,k,c,1);return 0;}static const u8
    sigma[16]="expand 32-byte k";int crypto_stream_salsa20_xor(u8*c,
    const u8
*m,u64 b,const u8*n,const u8*k){u8 z[16],x[64];u32 u,i;if(!b)return 0;
    FOR(i,16)z[i]=0;FOR(i,8)z[i]=n[i];while(b>=64){crypto_core_salsa20
    (x,z
,k,sigma);FOR(i,64)c[i]=(m?m[i]:0)^x[i];u=1;for(i=8;i<16;++i){u+=(u32)
    z[i];z[i]=u;u>>=8;}b-=64;c+=64;if(m)m+=64;}if(b){
    crypto_core_salsa20(x
,z,k,sigma);FOR(i,b)c[i]=(m?m[i]:0)^x[i];}return 0;}int
    crypto_stream_salsa20(u8*c,u64 d,const u8*n,const u8*k){return
crypto_stream_salsa20_xor(c,0,d,n,k);}int crypto_stream(u8*c,u64 d,
    const u8*n,const u8*k){u8 s[32];crypto_core_hsalsa20(s,n,k,sigma);
    return
crypto_stream_salsa20(c,d,n+16,s);}int crypto_stream_xor(u8*c,const u8
    *m,u64 d,const u8*n,const u8*k){u8 s[32];crypto_core_hsalsa20(s,n,
    k,
sigma);return crypto_stream_salsa20_xor(c,m,d,n+16,s);}sv add1305(u32*
    h,const u32*c){u32 j,u=0;FOR(j,17){u+=h[j]+c[j];h[j]=u&255;u
    >>=8;}}
static const u32 minusp[17]={5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,252};int
    crypto_onetimeauth(u8*out,const u8*m,u64 n,const u8*k){u32 s,i,j,u
    ,x[
17],r[17],h[17],c[17],g[17];FOR(j,17)r[j]=h[j]=0;FOR(j,16)r[j]=k[j];r
    [3]&=15;r[4]&=252;r[7]&=15;r[8]&=252;r[11]&=15;r[12]&=252;r
    [15]&=15;
while(n>0){FOR(j,17)c[j]=0;for(j=0;(j<16)&&(j<n);++j)c[j]=m[j];c[j]=1;
    m+=j;n-=j;add1305(h,c);FOR(i,17){x[i]=0;FOR(j,17)x[i]+=h[j]*((j<=i
    )?r[
i-j]:320*r[i+17-j]);}FOR(i,17)h[i]=x[i];u=0;FOR(j,16){u+=h[j];h[j]=u
    &255;u>>=8;}u+=h[16];h[16]=u&3;u=5*(u>>2);FOR(j,16){u+=h[j];h[j]=u
    &255;u
>>=8;}u+=h[16];h[16]=u;}FOR(j,17)g[j]=h[j];add1305(h,minusp);s= -(h
    [16]>>7);FOR(j,17)h[j]^=s&(g[j]^h[j]);FOR(j,16)c[j]=k[j+16];c
    [16]=0;
add1305(h,c);FOR(j,16)out[j]=h[j];return 0;}int
    crypto_onetimeauth_verify(const u8*h,const u8*m,u64 n,const u8*k){
    u8 x[16];
crypto_onetimeauth(x,m,n,k);return crypto_verify_16(h,x);}int
    crypto_secretbox(u8*c,const u8*m,u64 d,const u8*n,const u8*k){int
    i;if(d<32)
return-1;crypto_stream_xor(c,m,d,n,k);crypto_onetimeauth(c+16,c+32,d
    -32,c);FOR(i,16)c[i]=0;return 0;}int crypto_secretbox_open(u8*m,
    const u8
*c,u64 d,const u8*n,const u8*k){int i;u8 x[32];if(d<32)return-1;
    crypto_stream(x,32,n,k);if(crypto_onetimeauth_verify(c+16,c+32,d
    -32,x)!=0)
return-1;crypto_stream_xor(m,c,d,n,k);FOR(i,32)m[i]=0;return 0;}sv
    set25519(gf r,const gf a){int i;FOR(i,16)r[i]=a[i];}sv car25519(gf
     o){int
i;i64 c;FOR(i,16){o[i]+=(1LL<<16);c=o[i]>>16;o[(i+1)*(i<15)]+=c-1+37*(
    c-1)*(i==15);o[i]-=c<<16;}}sv sel25519(gf p,gf q,int b){i64 t,i,c
    =~(b-
1);FOR(i,16){t=c&(p[i]^q[i]);p[i]^=t;q[i]^=t;}}sv pack25519(u8*o,const
     gf n){int i,j,b;gf m,t;FOR(i,16)t[i]=n[i];car25519(t);car25519(t)
```

```
    ;
car25519(t);FOR(j,2){m[0]=t[0]-0xffed;for(i=1;i<15;i++){m[i]=t[i]-0
    xffff-((m[i-1]>>16)&1);m[i-1]&=0xffff;}m[15]=t[15]-0x7fff-((m
    [14]>>16)&1)
;b=(m[15]>>16)&1;m[14]&=0xffff;sel25519(t,m,1-b);}FOR(i,16){o[2*i]=t[i
    ]&0xff;o[2*i+1]=t[i]>>8;}}static int neq25519(const gf a,const gf
    b){
u8 c[32],d[32];pack25519(c,a);pack25519(d,b);return crypto_verify_32(c
    ,d);}static u8 par25519(const gf a){u8 d[32];pack25519(d,a);return
     d[0
]&1;}sv unpack25519(gf o,const u8*n){int i;FOR(i,16)o[i]=n[2*i]+((i64)
    n[2*i+1]<<8);o[15]&=0x7fff;}sv A(gf o,const gf a,const gf b){int i
    ;FOR
(i,16)o[i]=a[i]+b[i];}sv Z(gf o,const gf a,const gf b){int i;FOR(i,16)
    o[i]=a[i]-b[i];}sv M(gf o,const gf a,const gf b){i64 i,j,t[31];FOR
    (i,
31)t[i]=0;FOR(i,16)FOR(j,16)t[i+j]+=a[i]*b[j];FOR(i,15)t[i]+=38*t[i
    +16];FOR(i,16)o[i]=t[i];car25519(o);car25519(o);}sv S(gf o,const
    gf a){M(
o,a,a);}sv inv25519(gf o,const gf i){gf c;int a;FOR(a,16)c[a]=i[a];for
    (a=253;a>=0;a--){S(c,c);if(a!=2&&a!=4)M(c,c,i);}FOR(a,16)o[a]=c[a
    ];}sv
pow2523(gf o,const gf i){gf c;int a;FOR(a,16)c[a]=i[a];for(a=250;a>=0;
    a--){S(c,c);if(a!=1)M(c,c,i);}FOR(a,16)o[a]=c[a];}int
crypto_scalarmult(u8*q,const u8*n,const u8*p){u8 z[32];i64 x[80],r,i;
    gf a,b,c,d,e,f;FOR(i,31)z[i]=n[i];z[31]=(n[31]&127)|64;z[0]&=248;
unpack25519(x,p);FOR(i,16){b[i]=x[i];d[i]=a[i]=c[i]=0;}a[0]=d[0]=1;for
    (i=254;i>=0;--i){r=(z[i>>3]>>(i&7))&1;sel25519(a,b,r);sel25519(c,d
    ,r);
A(e,a,c);Z(a,a,c);A(c,b,d);Z(b,b,d);S(d,e);S(f,a);M(a,c,a);M(c,b,e);A(
    e,a,c);Z(a,a,c);S(b,a);Z(c,d,f);M(a,c,_121665);A(a,a,d);M(c,c,a);M
    (a,d
,f);M(d,b,x);S(b,e);sel25519(a,b,r);sel25519(c,d,r);}FOR(i,16){x[i
    +16]=a[i];x[i+32]=c[i];x[i+48]=b[i];x[i+64]=d[i];}inv25519(x+32,x
    +32);M(x+
16,x+16,x+32);pack25519(q,x+16);return 0;}int crypto_scalarmult_base(
    u8*q,const u8*n){return crypto_scalarmult(q,n,_9);}int
crypto_box_keypair(u8*y,u8*x){randombytes(x,32);return
    crypto_scalarmult_base(y,x);}int crypto_box_beforenm(u8*k,const u8
    *y,const u8*x){u8 s
[32];crypto_scalarmult(s,x,y);return crypto_core_hsalsa20(k,_0,s,sigma
    );}int crypto_box_afternm(u8*c,const u8*m,u64 d,const u8*n,const
    u8*k)
{return crypto_secretbox(c,m,d,n,k);}int crypto_box_open_afternm(u8*m,
    const u8*c,u64 d,const u8*n,const u8*k){return
    crypto_secretbox_open(m
,c,d,n,k);}int crypto_box(u8*c,const u8*m,u64 d,const u8*n,const u8*y,
    const u8*x){u8 k[32];crypto_box_beforenm(k,y,x);return
crypto_box_afternm(c,m,d,n,k);}int crypto_box_open(u8*m,const u8*c,u64
     d,const u8*n,const u8*y,const u8*x){u8 k[32];crypto_box_beforenm(
    k,y,
x);return crypto_box_open_afternm(m,c,d,n,k);}static u64 R(u64 x,int c
    ){return(x>>c)|(x<<(64-c));}static u64 Ch(u64 x,u64 y,u64 z){
    return(x&
y)^(~x&z);}static u64 Maj(u64 x,u64 y,u64 z){return(x&y)^(x&z)^(y&z);}
    static u64 Sigma0(u64 x){return R(x,28)^R(x,34)^R(x,39);}static
    u64
Sigma1(u64 x){return R(x,14)^R(x,18)^R(x,41);}static u64 sigma0(u64 x)
    {return R(x,1)^R(x,8)^(x>>7);}static u64 sigma1(u64 x){return R(x
    ,19)^
R(x,61)^(x>>6);}static const u64 K[80]={0x428a2f98d728ae22ULL,0
    x7137449123ef65cdULL,0xb5c0fbcfec4d3b2fULL,0xe9b5dba58189dbbcULL,
```

```
0x3956c25bf348b538ULL,0x59f111f1b605d019ULL,0x923f82a4af194f9bULL,0
    xab1c5ed5da6d8118ULL,0xd807aa98a3030242ULL,0x12835b0145706fbeULL,
0x243185be4ee4b28cULL,0x550c7dc3d5ffb4e2ULL,0x72be5d74f27b896fULL,0
    x80deb1fe3b1696b1ULL,0x9bdc06a725c71235ULL,0xc19bf174cf692694ULL,
0xe49b69c19ef14ad2ULL,0xefbe4786384f25e3ULL,0x0fc19dc68b8cd5b5ULL,0
    x240ca1cc77ac9c65ULL,0x2de92c6f592b0275ULL,0x4a7484aa6ea6e483ULL,
0x5cb0a9dcbd41fbd4ULL,0x76f988da831153b5ULL,0x983e5152ee66dfabULL,0
    xa831c66d2db43210ULL,0xb00327c898fb213fULL,0xbf597fc7beef0ee4ULL,
0xc6e00bf33da88fc2ULL,0xd5a79147930aa725ULL,0x06ca6351e003826fULL,0
    x142929670a0e6e70ULL,0x27b70a8546d22ffcULL,0x2e1b21385c26c926ULL,
0x4d2c6dfc5ac42aedULL,0x53380d139d95b3dfULL,0x650a73548baf63deULL,0
    x766a0abb3c77b2a8ULL,0x81c2c92e47edaee6ULL,0x92722c851482353bULL,
0xa2bfe8a14cf10364ULL,0xa81a664bbc423001ULL,0xc24b8b70d0f89791ULL,0
    xc76c51a30654be30ULL,0xd192e819d6ef5218ULL,0xd69906245565a910ULL,
0xf40e35855771202aULL,0x106aa07032bbd1b8ULL,0x19a4c116b8d2d0c8ULL,0
    x1e376c085141ab53ULL,0x2748774cdf8eeb99ULL,0x34b0bcb5e19b48a8ULL,
0x391c0cb3c5c95a63ULL,0x4ed8aa4ae3418acbULL,0x5b9cca4f7763e373ULL,0
    x682e6ff3d6b2b8a3ULL,0x748f82ee5defb2fcULL,0x78a5636f43172f60ULL,
0x84c87814a1f0ab72ULL,0x8cc702081a6439ecULL,0x90befffa23631e28ULL,0
    xa4506cebde82bde9ULL,0xbef9a3f7b2c67915ULL,0xc67178f2e372532bULL,
0xca273eceea26619cULL,0xd186b8c721c0c207ULL,0xeada7dd6cde0eb1eULL,0
    xf57d4f7fee6ed178ULL,0x06f067aa72176fbaULL,0x0a637dc5a2c898a6ULL,
0x113f9804bef90daeULL,0x1b710b35131c471bULL,0x28db77f523047d84ULL,0
    x32caab7b40c72493ULL,0x3c9ebe0a15c9bebcULL,0x431d67c49c100d4cULL,
0x4cc5d4becb3e42b6ULL,0x597f299cfc657e2aULL,0x5fcb6fab3ad6faecULL,0
    x6c44198c4a475817ULL};int crypto_hashblocks(u8*x,const u8*m,u64 n)
    {u64 z[
8],b[8],a[8],w[16],t;int i,j;FOR(i,8)z[i]=a[i]=dl64(x+8*i);while(n
    >=128){FOR(i,16)w[i]=dl64(m+8*i);FOR(i,80){FOR(j,8)b[j]=a[j];t=a
    [7]+Sigma1
(a[4])+Ch(a[4],a[5],a[6])+K[i]+w[i%16];b[7]=t+Sigma0(a[0])+Maj(a[0],a
    [1],a[2]);b[3]+=t;FOR(j,8)a[(j+1)%8]=b[j];if(i%16==15)FOR(j,16)w[j
    ]+=w[
(j+9)%16]+sigma0(w[(j+1)%16])+sigma1(w[(j+14)%16]);}FOR(i,8){a[i]+=z[i
    ];z[i]=a[i];}m+=128;n-=128;}FOR(i,8)ts64(x+8*i,z[i]);return n;}
    static
const u8 iv[64]={0x6a,0x09,0xe6,0x67,0xf3,0xbc,0xc9,0x08,0xbb,0x67,0
    xae,0x85,0x84,0xca,0xa7,0x3b,0x3c,0x6e,0xf3,0x72,0xfe,0x94,0xf8,0
    x2b,
0xa5,0x4f,0xf5,0x3a,0x5f,0x1d,0x36,0xf1,0x51,0x0e,0x52,0x7f,0xad,0xe6
    ,0x82,0xd1,0x9b,0x05,0x68,0x8c,0x2b,0x3e,0x6c,0x1f,0x1f,0x83,0xd9
    ,0xab,
0xfb,0x41,0xbd,0x6b,0x5b,0xe0,0xcd,0x19,0x13,0x7e,0x21,0x79};int
    crypto_hash(u8*out,const u8*m,u64 n){u8 h[64],x[256];u64 i,b=n;FOR
    (i,64)h[i
]=iv[i];crypto_hashblocks(h,m,n);m+=n;n&=127;m-=n;FOR(i,256)x[i]=0;FOR
    (i,n)x[i]=m[i];x[n]=128;n=256-128*(n<112);x[n-9]=b>>61;ts64(x+n-8,
    b<<3
);crypto_hashblocks(h,x,n);FOR(i,64)out[i]=h[i];return 0;}sv add(gf p
    [4],gf q[4]){gf a,b,c,d,t,e,f,g,h;Z(a,p[1],p[0]);Z(t,q[1],q[0]);M(
    a,a,t
);A(b,p[0],p[1]);A(t,q[0],q[1]);M(b,b,t);M(c,p[3],q[3]);M(c,c,D2);M(d,
    p[2],q[2]);A(d,d,d);Z(e,b,a);Z(f,d,c);A(g,d,c);A(h,b,a);M(p[0],e,f
    );M(
p[1],h,g);M(p[2],g,f);M(p[3],e,h);}sv cswap(gf p[4],gf q[4],u8 b){int
    i;FOR(i,4)sel25519(p[i],q[i],b);}sv pack(u8*r,gf p[4]){gf tx,ty,zi
    ;
inv25519(zi,p[2]);M(tx,p[0],zi);M(ty,p[1],zi);pack25519(r,ty);r[31]^=
    par25519(tx)<<7;}sv scalarmult(gf p[4],gf q[4],const u8*s){int i;
set25519(p[0],gf0);set25519(p[1],gf1);set25519(p[2],gf1);set25519(p
    [3],gf0);for(i=255;i>=0;--i){u8 b=(s[i/8]>>(i&7))&1;cswap(p,q,b);
    add(q,p)
```

```
;add(p,p);cswap(p,q,b);}}sv scalarbase(gf p[4],const u8*s){gf q[4];
    set25519(q[0],X);set25519(q[1],Y);set25519(q[2],gf1);M(q[3],X,Y);
scalarmult(p,q,s);}int crypto_sign_keypair(u8*pk,u8*sk){u8 d[64];gf p
    [4];int i;randombytes(sk,32);crypto_hash(d,sk,32);d[0]&=248;d
    [31]&=127;
d[31]|=64;scalarbase(p,d);pack(pk,p);FOR(i,32)sk[32+i]=pk[i];return
    0;}static const u64 L[32]={0xed,0xd3,0xf5,0x5c,0x1a,0x63,0x12,0x58
    ,0xd6,
0x9c,0xf7,0xa2,0xde,0xf9,0xde,0x14,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0x10
    };sv modL(u8*r,i64 x[64]){i64 carry,i,j;for(i=63;i>=32;--i){carry
    =0;for
(j=i-32;j<i-12;++j){x[j]+=carry-16*x[i]*L[j-(i-32)];carry=(x[j]+128)
    >>8;x[j]-=carry<<8;}x[j]+=carry;x[i]=0;}carry=0;FOR(j,32){x[j]+=
    carry-(x
[31]>>4)*L[j];carry=x[j]>>8;x[j]&=255;}FOR(j,32)x[j]-=carry*L[j];FOR(i
    ,32){x[i+1]+=x[i]>>8;r[i]=x[i]&255;}}sv reduce(u8*r){i64 x[64],i;
    FOR(i
,64)x[i]=(u64)r[i];FOR(i,64)r[i]=0;modL(r,x);}int crypto_sign(u8*sm,
    u64*smlen,const u8*m,u64 n,const u8*sk){u8 d[64],h[64],r[64];i64 i
    ,j,x[
64];gf p[4];crypto_hash(d,sk,32);d[0]&=248;d[31]&=127;d[31]|=64;*smlen
    =n+64;FOR(i,n)sm[64+i]=m[i];FOR(i,32)sm[32+i]=d[32+i];crypto_hash(
    r,sm
+32,n+32);reduce(r);scalarbase(p,r);pack(sm,p);FOR(i,32)sm[i+32]=sk[i
    +32];crypto_hash(h,sm,n+64);reduce(h);FOR(i,64)x[i]=0;FOR(i,32)x[i
    ]=(
u64)r[i];FOR(i,32)FOR(j,32)x[i+j]+=h[i]*(u64)d[j];modL(sm+32,x);return
     0;}static int unpackneg(gf r[4],const u8 p[32]){gf t,chk,num,den,
    den2
,den4,den6;set25519(r[2],gf1);unpack25519(r[1],p);S(num,r[1]);M(den,
    num,D);Z(num,num,r[2]);A(den,r[2],den);S(den2,den);S(den4,den2);M(
    den6,
den4,den2);M(t,den6,num);M(t,t,den);pow2523(t,t);M(t,t,num);M(t,t,den)
    ;M(t,t,den);M(r[0],t,den);S(chk,r[0]);M(chk,chk,den);if(neq25519(
    chk,
num))M(r[0],r[0],I);S(chk,r[0]);M(chk,chk,den);if(neq25519(chk,num))
    return-1;if(par25519(r[0])==(p[31]>>7))Z(r[0],gf0,r[0]);M(r[3],r
    [0],r[1]
);return 0;}int crypto_sign_open(u8*m,u64*mlen,const u8*sm,u64 n,const
     u8*pk){int i;u8 t[32],h[64];gf p[4],q[4];*mlen= -1;if(n<64)return
    -1;
if(unpackneg(q,pk))return-1;FOR(i,n)m[i]=sm[i];FOR(i,32)m[i+32]=pk[i];
    crypto_hash(h,m,n);reduce(h);scalarmult(p,q,h);scalarbase(q,sm+32)
    ;add
(p,q);pack(t,p);n-=64;if(crypto_verify_32(sm,t)){FOR(i,n)m[i]=0;return
    -1;}FOR(i,n)m[i]=sm[i+64];*mlen=n;return 0;}
```