„Implementing the Phantom Protocol”

Diploma Thesis in Computer Science

written by

Johannes Schlumberger

born 25.01.1983

Chair for Distributed Systems and Operating Systems

Friedrich-Alexander-University at Erlangen-Nürnberg

Advisors: **Prof. Dr. W. Schröder-Preikschat**
Dipl.-Inf. Michael Gernoth

Begin: 01.06.2010
End: 08.10.2010

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen,

Abstract

This work describes the prototypical implementation of the Phantom Anonymization Protocol designed by Magnus Bråding. The protocol exists to this date only as a theoretical concept. As it is much needed as a usable program, implementing a prototype provided a good way to start the transition from theory to practice.

Implementing a theoretical design backs up the theoretical concept and proves its usability in practice.

I found the design implementable and noted some problems with it. In this thesis I will discuss my implementation, my findings and problems and then recommend some further improvements. The work should be seen as the first of a series of steps bringing Phantom from a theoretical design to a usable tool in the real world.

Contents

Chapter 1

A Short Overview Over The Phantom Protocol

1.1 Motivation

The Phantom Protocol[2] was designed in 2008 by Magnus Bråding to provide anonymity for the average Internet user. During the last years there has been a massive upswing in surveillance and censorship laws not only by unfree states like the Peoples Republic of China but also in the so-called free western democracies like the United States of America or various states of the European Union. Together with the fact that more and more traditional communication forms are replaced by new ones using the Internet, the need for better anonymization tools becomes more pressing. There have been such tools in the past, the maybe best known one is the TOR-Project[12] initially released in late 2002. However these tools have never seen widespread use and there have also been problems[8, 10], where the leakage of information out of those networks have led to the anonymization being broken or where people hosting TOR-services have seen themselves confronted with massive lawsuits aimed at taking down the TOR-nodes.

With these Problems in mind Magnus Bråding specifically designed the Phantom protocol to overcome these flaws.

1.2 Design assumptions

The following three assumptions were made before designing the Phantom protocol:

1. All traffic of every node in the network is assumed to be eavesdropped on, but never all nodes involved in the whole network by one party at the same time.

2. Arbitrary nodes participating in the network are assumed to be adverse or compromised by an attacker.

3. The protocol must be free from any trusted or central entity in order to avoid that this entity can be taken down by lawsuits or other means.

These three design assumptions seem both reasonable and necessary for a large scale anonymized network, where there are no (or very little) personal contacts between the participants and so no knowledge about which participants one might trust. Those three design assumptions lead to some important consequences regarding the Phantom protocol. First, the resulting network must be completely decentralized and distributed (if possible among e.g. different jurisdictions or countries), secondly the protocol must allow to distrust any particular node in the network and never assume any single chosen node to be "good". Last but not least, probabilistically secure algorithms must be used instead of deterministically secure ones.
Since CPU-power and network bandwidth become increasingly cheaper and more available, any tradeoff decisions between those two factors on the one hand and better security or anonymity on the other hand have been made in favor of the latter two.

1.3 Design goals

With the above mentioned design assumptions and consequences in mind the Phantom protocol was designed to meet the following eight design goals:

1.3.1 Complete decentralization

If there is a weak point (or weak node for that matter) in the design of the protocol, it is always the likeliest point to be attacked by anyone interested to harm the network in any way. Thus all nodes must be equally important and equally unimportant, no node must be irreplaceable in the network or exposed in any other way as an attack goal, for that will surely be the point an attacker will concentrate his resources on. This is not only important from a technical point of view, but also from a legal point of view. There may be no person more or less important than any other participant of the network, because this person will then be chosen as the attacking point for lawsuits or other means of attack.

1.3.2 Maximum resistance against all kinds of DoS-attacks

If Design goal 1. achieves its purpose there will be no single point to attack in the network, and any attacker will be forced to attack the network as a whole to be successful. The network must therefore be able to resist these attacks.

1.3.3 Theoretically secure anonymization

The security of the anonymization provided by the network must be theoretically provable. There can still be errors in any implementation, but they are usually easier to fix than a problem with the underlying theoretical design.

1.3.4 Theoretically secure end-to-end encryption

If anyone can eavesdrop the unencrypted communication between two communication partners, it is highly likely that he can deduce information leading to the identification of the communication parties. Therefore the communication has to be encrypted from sender to recipient and back, in order to prevent breaking of anonymity by an eavesdropper.

1.3.5 Complete virtual isolation from the "normal" Internet

The TOR-project uses so called exit-nodes as proxies to the normal Internet. The owners of these proxies can be identified and sued if someone from the TOR-network behind that proxy attacks a target in the "normal" Internet. That poses a problem for the exit-node providers. Phantom therefore completely separates its network from the normal Internet. The Phantom network forms an overlay network, using the infrastructure of the normal Internet, but strictly separates the contents, so it is not possible for an anonymized vandal to reach any service in the normal Internet and have some unrelated exit-node provider taking the blame for it. Anyone is free to make only those select services available to the anonymized network he wants to. He should do so knowing that no culprit can be found (by design) if those services are damaged in any way. He is responsible for offering this service to the anonymized network, if he does not want to take that chance, he should not make the service reachable from an anonymized network.

1.3.6 Maximum protection against protocol identification

Several ISPs are filtering or shaping down traffic they do not like or shaping up traffic if they are paid for it. Therefore if a protocol is easily identifiable and unwanted, it could be dropped by any ISP. This is a potential threat to an anonymization network, since if a participant cannot be identified and therefore cannot be coerced to stop using that network, the network can just be dropped by some/most ISPs in e.g. a certain country and thus be made unavailable to all people (including the one person one wishes to prevent from using the network). Therefore the Phantom protocol must not be easily identifiable from the outside, as no one can selectively drop traffic he cannot distinguish from traffic he does not want to drop.

1.3.7 High traffic volume and throughput capacity

Poor performance is a negative trait on any network protocol and must be eliminated as far as possible, to make it interesting for many users and applications. If someone cannot use the anonymization network for the things he wants to do anonymized, it is useless for him. Furthermore, a distributed design gets better with more users joining in, so the drawbacks of using the network should be as small as possible, in order to make the user base big.

1.3.8 Generic, well-abstracted and backward compatible design

A generic system is in the long run always superior to a specific one (be it only in numbers of users). A well-abstracted design makes sure that a design mistake in one part does not make the whole system useless but might be fixed. A backward compatible design makes it easier for a new protocol to be used with already existing applications, greatly boosting the acceptance rate and variety of services offered over the protocol. Phantom aims for complete IP-compatibility here.

1.4 High level technical view

The Phantom protocol as described in the original paper [2] consists of three different parts. First a routing path which can be seen as a set of nodes relaying incoming and outgoing traffic to and from an anonymized node, with a designated entry or exit node acting as a proxy for incoming or outgoing requests to or from the anonymized node. Second the routing tunnels which form the logical equivalent of a point to point connection. It consists of the nodes forming the entry routing path and the exit routing path of two communication endpoints. Third a distributed hash table, containing various information, e.g. certificates and information about the current entry nodes for a given service on a given anonymized node. Magnus Bråding's paper gives the full details if the reader is interested, I will only give a very short overview about the different parts in the next couple of subsections. Readers of this document are strongly advised to familiarize themselves with Magnus' design in order to become able to understand the remainder of this thesis.

1.4.1 Routing paths

A routing path is the central anonymization concept of the Phantom design. It consists of a number of nodes relaying traffic for the anonymized node who is also called the owner of the routing path. Routing paths come in two flavors, exit

paths and entry paths. An exit path is used for outgoing connections into the Phantom network, an entry path is used to accept incoming connections from the Phantom network. It is expected that someone interested in content within the Phantom network would own an exit path, someone providing content an entry path.

Nodes participating in a Phantom routing path construction process are divided into two groups called X and Y, and according to their membership in one of these groups called X- or Y-nodes. The X nodes will be in the finished routing path, the Y nodes are only used to help in construction. A finished routing path can be thought of as a number of ordered X-nodes like pearls on a string. On the one end there is the owner of the routing path, then some intermediate X-nodes and on the other end the terminating X-node which is called the exit node or entry node depending on the path's flavor. The string between the individual pearls (X-nodes) are SSL-connections.

Path construction is done in three steps initiated and controlled by the path's owner to be. The first step is preparation, the owner selects other Phantom nodes he wants to participate in the path and prepares information called setup packages for these nodes. In the second stage he sends this information to the selected nodes in a special way as to not disclose his anonymity. The receiving nodes will act on the received packages by starting to form a routing path. This step is repeated once with slightly different information. These steps are fully controlled and strictly verified by the owner of the constructed path. If all goes well and as expected, the Y-nodes are kicked out of the path and the X-node form into the string of pearls explained above. The routing path is then finished and routing tunnels can be created over it. A finished example path can be seen in 1.1.

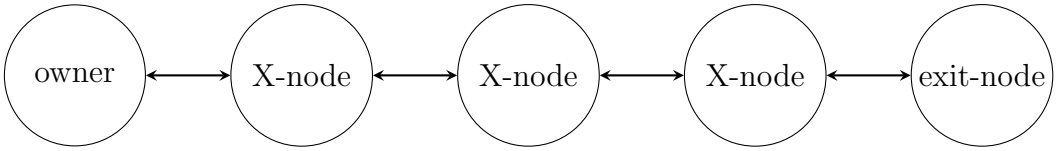

Figure 1.1: Example of a finished exit routing path of length three

1.4.2 Routing tunnels

Routing tunnels are used to communicate between two Phantom nodes. They are built along paths, using them as pre-built infrastructure. So a tunnel can not exist without a path. Tunnel construction is always initiated by the owner of an exit path. The request for a new tunnel travels along the exit path until it reaches the exit node. After receiving this request, the exit node tries to connect to an entry node. The entry node receiving this connection then sends the request

along the corresponding entry path whose entry point it is, back to the entry path's owner.

The basic connection between the two Phantom nodes has been made. Some additional data is then exchanged via the paths between the owners and their terminating nodes to make the tunnel ready for use. After this data exchange, the tunnel can be seen as a bidirectional point-to-point connection between the two path owners. Those two can now start to exchange arbitraty data over it, which will be onion encrypted while travelling through the tunnel. An example of a finished tunnel can be seen in 1.2. The strings in this picture connecting the nodes are not the original SSL-connections resulting from the path creation process but completely new ones, just involving the same nodes.

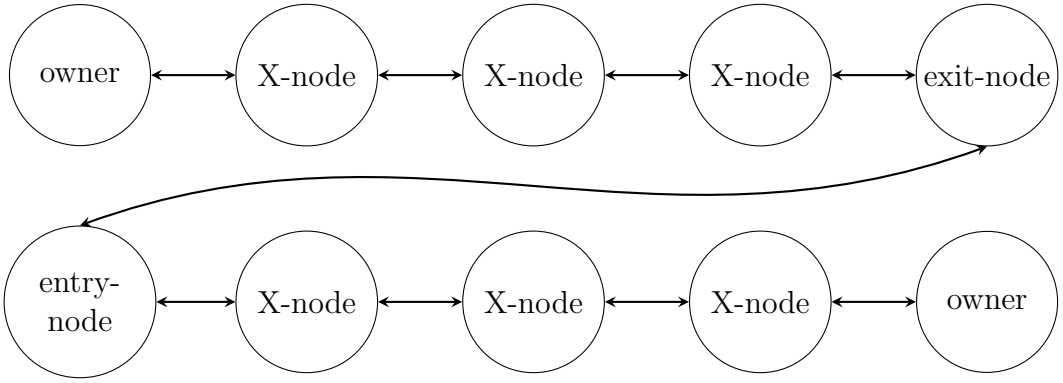

Figure 1.2: Example of a finished routing tunnel

1.4.3 Distributed hash table

The distributed hash table (DHT) is used to store information needed by the protocol. This part is very abstract, yet crucial to the success of the protocol. At the very least, two kind of information has to be stored within the DHT. A mapping that makes it possible to get the entry nodes for an AP-address[1] and information describing other Phantom nodes, specifically their IP-addresses and ports along with their certificates.

[1] An AP-address is like an IP-address, but anonymized

Chapter 2

Implementation

2.1 Basic design decisions

2.1.1 AP-addresses in IPv6 form

IPv6 defines unique local addresses in RFC4193 [6]. These addresses are specified as site-local-addresses so they are not routed globally, which does not pose a problem. Two prefixes are defined fc00::/8 and fd00::/8. The fc-prefix is proposed to be managed by a central authority. Once this authority exists and Phantom starts to see use outside of research laboratories, a block of addresses should be registered for Phantom with this authority.

In my current implementation I use a /48-block randomly chosen from the fd-prefix block. This provides enough AP-addresses for testing. Choosing IPv6 addresses as AP-addresses makes Phantom integrateable with all IPv6-enabled operating systems and IPv6-enabled applications, of which there are already a lot today and their number is steadily growing.

2.1.2 One server and one port

I have decided to use only one TCP-port for all Phantom related traffic. This induces difficulties because different traffic types have to be analyzed on arrival and then dispatched to the different modules handling them. It, however, reduces the complexity of using Phantom behind packet filters and makes traffic analysis attacks much harder.

2.1.3 Programming language

I have chosen to implement the prototype in C.

2.1.4 Operating system

I implemented the prototype for the Linux operating system since I am most comfortable and familiar with it. I have tried to match certain standards discussed in the next section to make porting it to other operating systems easier.

2.2 Standards

I have tried to stick to the POSIX.1-2001-standard. This standard does not include certain often used functions found within most C-libraries. I have implemented some of these functions myself, except `snprintf`, I am instead using `sprintf` for now. This should be changed before the prototype sees use outside of controlled environments.

Furthermore I have taken care to write ANSI-C.

I have been using the GNU C compiler[1] in version 4.3.2 as my compiler with the following quite restrictive flags:

Listing 2.1: Compiler flags used

```
1 -O2 -ansi -D_POSIX_C_SOURCE=200112L
2 -Wall -Werror -Wextra -Wbad-function-cast
3 -Wcast-align -Wcast-qual -Wdeclaration-after-statement
4 -Wmissing-prototypes -Wpointer-arith -Wshadow
5 -Wstrict-prototypes -Wformat -Wformat-security -Wunused
6 -Wwrite-strings -Waggregate-return -pedantic
```

In addition, I used the clang static ananlyzer in version 2.8. This is a static code analysis tool from the llvm project[2]. The code passes both the compile runs and the static analysis without any warnings or errors.

2.2.1 Encryption algorithms and hashes

For the prototype implementation I have chosen to use AES256[7] in cipher block chaining mode, whenever the need for a symmetric block cipher arises. The onion encryption used when forwarding application data over the tunnels can obviously not be done with a block cipher, so AES256 in output feedback mode has been used. Output feedback mode transforms the AES256 block cipher into a synchronous stream cipher. The SSL-certificates used for testing were 2048-bit RSA certificates, other sizes should work, too. The path building certificate generated specifically to be used with one path by the owner node to be, during path construction is an exception from this rule. It has a fixed size of 2048 bit. If this size is changed, the implementation of the path module would have to be changed, too.

[1] http://gcc.gnu.org
[2] http://llvm.org/

All hashes used are SHA-1[1].

2.3 Used libraries

2.3.1 libopenssl

OpenSSL[3] is an open source implementation of the SSL and TLS protocols. It also implements various cryptographic algorithms like symmetric and asymmetric ciphers or digests. Since it is never a good idea to implement those security critical algorithms anew, I have chosen to use libopenssl in my implementation. OpenSSL is available on most UNIX-Systems including the most commonly used ones (Linux, *BSD, Solaris), Mac OS X and Microsoft Windows. It is the de facto standard library for open source SSL-Applications. It is available under an Apache-like license[4].

2.3.2 libprotobuf-c

Protobuf-c provides runtime libraries and a code generator for "Protocol Buffers" in pure C. "Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data - think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the old format."[5] Protobuf-c is is available under the "Apache License 2.0"[6] The original protocol buffers implementation under the "New BSD License"[7].

This library is used to serialize and deserialize the messages sent among the participating nodes both during the creation of routing paths and when communicating as a DHT-Node. Using the Protocol Buffers library has several advantages:

1. Its easy to use (see the example below).
2. The message format can later be changed if necessary and the library will handle it correctly.
3. The protocol buffers are rather lightweight and fast.
4. It avoids the need for "manual" parsing by having the parser code automatically generated.

[3]http://www.openssl.org
[4]http://www.openssl.org/source/license.html
[5]http://code.google.com/apis/protocolbuffers/docs/overview.html
[6]http://www.apache.org/licenses/LICENSE-2.0
[7]http://www.opensource.org/licenses/bsd-license.php

5. It should be quite well tested since it is used for most of Google's internal RPC-Code.

However the protocol buffers may also introduce a loophole. During the creation of the routing paths it is essential that no uncontrolled in band communication whatsoever can take place between non-neighboring nodes in the routing path. It might be possible for an attacker to use the protocol buffers to slip some information to nodes later on in the routing path in order to coordinate a possible attack. Should this be the case, an alternative to the protocol buffers must be found. As this is only a proof of concept implementation so far, I have decided not to burden myself with complex serialization and deserialization and tedious checks for overflows and such, but left those things to the protocol buffers library.

With protobuf-c one writes a description of the message one wishes to send in a simple text file and then uses the protoc-c compiler on the file to generate C-source and header files to compile with the program. Those files mostly declare and define simple `typedef`-ed types for the individual messages, as well as pack-, unpack-, init- and free-functions for these types.

Listing 2.2: Example of a protobuf description file and protobuf-c

```
1  $ cat protos/kademlia.proto
2  message node_info {
3          required bytes id = 1;
4          required uint32 port = 2;
5          required bytes cert = 3;
6          required string ip = 4;
7  };
8
9  message store {
10         required bytes key = 1;
11         required bytes data = 2;
12         required node_info self = 3;
13 };
14
15 message store_reply {
16         required bool success = 1;
17 };
18
19 message find_close_nodes {
20         required bytes id = 1;
21         required node_info self = 2;
22 };
23
24 message find_close_nodes_reply {
25         repeated node_info nodes = 1;
26 };
27
28 message find_value {
29         required bytes key = 1;
30         required node_info self = 2;
```

```
31 };
32
33 message find_value_reply {
34         required bool success = 1;
35         repeated node_info nodes = 2;
36         optional bytes data = 3;
37 };
38
39 $ protoc-c --c_out=/tmp -I ../protos/ ../protos/kademlia.proto
40 $ ls /tmp/kademlia.pb-c.*
41 /tmp/kademlia.pb-c.c /tmp/kademlia.pb-c.h
```

The generated not yet `typedef`-ed structs look like this and are used in the obvious way.

Listing 2.3: Example struct generated by protobuf-c

```
1 struct _FindValueReply
2 {
3   ProtobufCMessage base;
4   protobuf_c_boolean success;
5   size_t n_nodes;
6   NodeInfo **nodes;
7   protobuf_c_boolean has_data;
8   ProtobufCBinaryData data;
9 };
```

In my current implementation I assume that protobuf-c never changes any data passed to it in one of the generated structs (which seems to be the case so far). However the generated code does not declare the passed-in parameters `const`, as one would expect. If later revisions of this library would start to change the data passed to it, the current implementation will have to be changed to handle this correctly.

2.4 Module overview

I have decided to describe the implementation on a per module basis, which can also serve as a documentation of the existing modules and their interactions for people interested in continuing my work. My implementation is roughly divisible in six parts, three of those are the parts of the Phantom protocol as described before, namely routing paths, routing tunnels and the DHT. The other three parts are the Phantom server which is used by all three Phantom modules, the operating system specific integration or frontend part providing the necessary code to make Phantom usable on a Linux machine and last a group of helper-modules providing various mostly simple functions that can be used by any other module. I will try to present the modules in an order in which they can be easily read one by one, but it lies in the nature of interacting modules that they cannot be fully explained one after the other and fully independently from each other.

2.5 Small helper modules

The helper modules are easily described since they just offer simple functionality or abstract from ugly or uninteresting things. They do not use any of the more important modules and so make a good starting point to describe the implementation.

2.5.1 config

The config module reads a configuration file for Phantom and stores its contents in a struct that is used to present the configuration options at runtime. The struct is passed to several other modules to provide them with the information needed. Typically this struct should be a singleton. The module exports two functions, one to read a configuration from file and store the contents to the struct and one to free the allocated contents within the struct. Since the struct itself can be allocated statically (as a singleton), I decided not to allocate it dynamically but to pass only a pointer to statically preallocated memory to the functions using it. Consequently the struct itself is not freed by the free-function.

Listing 2.4: The config module's interface

```
1  struct config {
2          char *ip;
3          char *kad_node_file;
4          char *kad_data_dir;
5          uint16_t port;
6          uint8_t nxnodes;
7          uint8_t nynodes;
8          uint8_t nkeys;
9          X509 *construction_certificate;
10         struct X509_flat *construction_certificate_flat;
11         EVP_PKEY *private_construction_key;
12         X509 *communication_certificate;
13         struct X509_flat *communication_certificate_flat;
14         EVP_PKEY *private_communication_key;
15         X509 *routing_certificate;
16         struct X509_flat *routing_certificate_flat;
17         EVP_PKEY *private_routing_key;
18 };
19 void read_config(char *path, struct config *config);
20 void free_config(struct config *config);
```

The members of the struct have the following meaning:

- ip - the ipv4 address of the interface, the server will listen on
- kad_node_file - the filename of previously stored kademlia contacts
- kad_data_dir - the directory used to store contents of the DHT locally

- port - the tcp port the server will listen on
- nxnodes - the number of X nodes used to create a routing path
- nynodes - the number of Y nodes used to create a routing path
- nkeys - the number of keys used per x-node for tunnels
- various certificates and their corresponding private keys stored in both serialized (flat) form and also in their runtime-representation (`X509`/`EVP_PKEY`) used by libopenssl. This is done to avoid converting too often from one format to the other. The filenames of the certificate files as generated by the openssl tool are found in the config file. The functions used for reading certificates from file and those to serialize and deserialize them are found in the helper module.

The config file itself is provided as an XML-file, which makes it parseable very easily using libxml2. This library is not listed with the used libraries because it should be simple to read the config manually from a file and fill in the struct – so it is not really a dependency for the implementation.

2.5.2 helper

The helper module is a collection of various functions written to provide simple operations not covered by the libc or abstracting from libopenssl and simplifying its use in the other modules. The following functions are exported from the module:

Listing 2.5: The helper module's interface

```
1  void randomize_array(void *base, size_t nmemb, size_t size);
2  void reverse_array(void *base, size_t nmemb, size_t size);
3  X509 *read_x509_from_file(const char *path);
4  void hexdump(const void *buf, int size);
5  EVP_PKEY *rsa_to_pkey(RSA *rsa);
6  void serialize_32_t(uint32_t t, uint8_t *buf);
7  uint32_t deserialize_32_t(const uint8_t *buf);
8  void serialize_16_t(uint16_t t, uint8_t *buf);
9  uint16_t deserialize_16_t(const uint8_t *buf);
10 char *ip4_to_char(uint32_t ip);
11 struct ssl_connection *create_ssl_connection(const char *ip,
       uint16_t port, X509 *cert, EVP_PKEY *privkey);
12 struct ssl_connection *create_ssl_connection_tmout(const char *ip,
       uint16_t port, X509 *cert, EVP_PKEY *privkey, uint32_t tmout);
13 void free_ssl_connection(struct ssl_connection *s);
14 void xor(uint8_t *s1, const uint8_t *s2, int len);
15 void rand_bytes(uint8_t *buf, int len);
16 int ssl_read(SSL *ssl, uint8_t *buf, uint32_t len);
17 int ssl_write(SSL *ssl, const uint8_t *buf, uint32_t len);
```

```
18 uint32_t rand_range(uint32_t min, uint32_t supremum);
19 char *parse_ip4_to_char(const struct in_addr *in);
20 char *bin_to_hex(const uint8_t *bin, int len);
21 char *strdup(const char *s);
22 uint8_t *read_package(SSL *ssl, uint32_t *outsize);
23 int write_package(SSL *ssl, uint8_t *data, uint32_t len);
```

Except for the *`ssl`* and *`package`*-functions their use and purpose should be obvious. The *`ssl`*-functions facilitate the creation and deletion of SSL-connections as well as blocking reading and writing from or to them. An SSL-connection is at runtime represented by a `struct ssl_connection`. This struct holds the SSL-context for libopenssl, the socket for the connection itself and (if available) the connection-peer's certificate.

Listing 2.6: struct ssl_connection

```
1 struct ssl_connection {
2         SSL *ssl;
3         int socket;
4         X509 *peer_cert;
5 };
```

The `create_ssl_connection*`-functions create an SSL-connection to the specified IPv4-address and TCP-port, using the given certificate and corresponding private key. The `_tmout`-function can be used to specify a shorter timeout than the usual TCP-timeout. Unfortunately there is no way to set the timeout directly for TCP-connection-attempts in Linux. So the socket has to be made nonblocking before the call to `connect`, then polled for the given timeout. The return value of `poll` can be used to determine if the connection attempt succeeded before the given timeout or if it timed out. After the `poll`-call the socket is made nonblocking again, before the SSL-handshake is started.

The write and read functions write `size` bytes from or read `size` bytes to the given buffer. They do either fail or read/write the amount of bytes. In case of failure this should be considered a hard error mostly and there is no way to know how much data has been read or written. If this information is necessary, these functions should not be used. They provide however a nice way to simply receive a known amount of data or send a known amount of data in a single call that hides all the quirkiness typical for libopenssl-I/O.

The *`package`*-functions provide a way to send or receive a given chunk of data via an SSL-connection. The data is transmitted as is with a prefixed 32-bit length.

2.5.3 openssl_locking

Libopenssl requires multi-threaded applications to provide at least two callback functions in order to become thread safe. This is what this module does. It

exports two functions which register or deregister the needed functions and locks with the library. The start-function should be called after library initialization but before any calls to libopenssl. The deregister function is used to clean up once the library is no longer used.

Listing 2.7: The openssl_locking module's interface

```
1 int init_locks(void);
2 void kill_locks(void);
```

2.5.4 thread_pool

The thread pool is an optimization since my implementation is heavily multi-threaded with many threads only having a short expected lifetime. So to get around the costly creation of threads, I wrote this module. It creates a number of threads once and then dispatches functions to idle threads. Arguments can be passed to this functions in a similar matter than with `pthread_create`. An additional, optional `free_func`-parameter can be passed that is called on the passed-in argument once the function has returned. An additional benefit of the thread pool is, that since no thread creation is done after the initialization, the creation can not fail. The exported interface of the thread_pool module is quite simple:

Listing 2.8: The thread_pool module's interface

```
1 struct thread_pool *new_thread_pool(int nthreads);
2 int thread_pool_dispatch(struct thread_pool *t, void *arg, void
     (*free_func)(void *), void (*start_func)(void *));
3 void free_thread_pool(struct thread_pool *t);
```

Typically there should be one thread pool for the whole application, but it is also possible to have several fully separated pools. The pool is created by calling `new_thread_pool` passing the number of threads. There is currently no way to increase or decrease this number later on, so it should be chosen carefully. A call to `free_thread_pool` waits for all threads to terminate and will cause the thread pool to no longer accept new functions. After the return of all threads they are terminated and their resources freed.

The `thread_pool_dispatch` function is roughly equivalent to `pthread_create` it must be passed at least the `thread_pool` context to use and a start-function to be executed by a free thread. Passing a single argument and free-function to the dispatch-function is optional.

2.5.5 x509_flat

It is necessary, due to the protocol design, to pass X509-certificates between nodes. Therefore those have to be serialized. As the internal representation of

X509-certificates within libopenssl is not exported to the user and there are no functions provided to serialize or deserialize a `struct X509` I have chosen to use the libopenssl-functions which are usually used to write and read certificates to and from disk. Instead of writing to disk, I write into memory and represent the serialized or flat certificates as a simple tuple of (bytes, length) which can then be transmitted over the wire and again transformed into an X509-certificate using the disk-read functions for certificates. Additional functions are made available to do various operations on the flat certificates. A total of twelve functions are exported from this module:

Listing 2.9: The x509_flat module's interface

```
1  struct X509_flat *new_X509_flat(void);
2  uint8_t *serialize_X509_flat(const struct X509_flat *x);
3  struct X509_flat *deserialize_X509_flat(const uint8_t *serialized);
4  void free_X509_flat(struct X509_flat *x);
5  struct X509_flat *read_x509_from_file_flat(const char *path);
6  X509 *read_x509_from_x509_flat(const struct X509_flat *fx);
7  int X509_serialized_size(const struct X509_flat *x);
8  struct X509_flat *flatten_X509(X509 *x);
9  int X509_compare(X509 *a, X509 *b);
10 int X509_compare_mixed(struct X509_flat *a, X509 *b);
11 int X509_compare_flat(struct X509_flat *a,  struct X509_flat *b);
12 int X509_hash(X509 *c, uint8_t *buf);
13 X509 *clone_cert(X509 *x);
```

The `compare`, `clone`, `new` and `free` functions should be obvious. The `hash` function stores the SHA-1 hash of the certificate in the passed-in buffer. The `read_x509_from_file_flat`-function is used by the config module to read the certificates from disk. The typical calls to serialize and deserialize a given X509-certificate `x` would be:

Listing 2.10: Sample code for certificate serialization and deserialization

```
1  uint8_t *serialized;
2  X509_flat *f;
3  /*on sending host */
4  f = flatten_X509(x);
5  serialized = serialize_X509_flat(f);
6  send_to_wire(serialized, X509_serialized_size(f));
7  free_X509_flat(f);
8  free(serialized);
9  /*on receiving host */
10 read_from_wire(serialized);
11 f = deserialize_X509_flat(serialized);
12 free(serialized);
13 x = read_x509_from_x509_flat(f);
14 free_X509_flat(f);
```

2.5.6 list

The list module is not a module but simply a header file providing a simple and efficient list data type purely via preprocessor macros. The list is implemented as a ring, which makes it easy to append or prepend to the list and saves some instructions because it eliminates the need to check for the typical cornercases usually found in list implementations (e.g. Where is the tail pointer?, Is the list empty right now?, etc.). The two major drawback with this implementation are that an element can only be in one list at a time and that one needs to declare and pass two additional pointers of the correct type to the iterator-macros. This could in theory be avoided but would effectively render the compilers type checking useless, which seems unwise.

Listing 2.11: Macros defined by the list header file

```
1 #define LIST_init(x) {(x)->next=(x)->prev=(x);}
2 #define LIST_is_empty(x) ((x)->next==(x))
3 #define LIST_insert(x,y) {((y)->next=(x)->next)->prev=(y);
      ((x)->next=(y))->prev=(x);}
4 #define LIST_insert_before(x,y) {((y)->prev=(x)->prev)->next=(y);
      ((x)->prev=(y))->next=(x);}
5 #define LIST_remove(x) {((x)->prev)->next=(x)->next;
      ((x)->next)->prev=(x)->prev;}
6 #define LIST_for_all(x,y,z) for
      (y=(x)->next,z=(y)->next;y!=(x);y=z,z=(y)->next)
7 #define LIST_for_all_backwards(x,y,z) for
      (y=(x)->prev,z=(y)->prev;y!=(x);y=z,z=(y)->prev)
8 #define LIST_clear(x,y) while ((x)->next!=(x)) { y = *x.next;
      LIST_remove(y); free(y);}
```

A short example program and its output should clarify how this list implementation is used:

Listing 2.12: Sample code for the list macros

```
1  #include <stdio.h>
2  #include "list.h"
3
4  struct element {
5          struct element *prev;
6          struct element *next;
7          int data;
8  };
9
10 static void
11 dump_list(const struct element *list, int backwards)
12 {
13         const struct element *help1, *help2;
14         if (backwards)
15                 LIST_for_all_backwards(list, help1, help2)
16                         printf("%d ", help1->data);
```

```
17          else
18                  LIST_for_all(list, help1, help2)
19                          printf("%d ", help1->data);
20          putchar('\n');
21  }
22
23  int
24  main(int argc, char **argv)
25  {
26          struct element head;
27          struct element one;
28          struct element two;
29          struct element three;
30          LIST_init(&head);
31          one.data = 1;
32          two.data = 2;
33          three.data = 3;
34          LIST_insert(&head, &one);
35          LIST_insert(&head, &two);
36          LIST_insert(&head, &three);
37          dump_list(&head, 0);
38          dump_list(&head, 1);
39          while (! LIST_is_empty(&head)) {
40                  LIST_remove(head.prev);
41                  dump_list(&head, 0);
42          }
43          LIST_insert(&head, &one);
44          dump_list(&head, 0);
45          LIST_insert_before(&head, &two);
46          LIST_insert_before(&head, &three);
47          dump_list(&head, 0);
48          return 0;
49  }
```

Listing 2.13: Sample run of the previous program

```
1  3 2 1
2  1 2 3
3  3 2
4  3
5
6  1
7  1 2 3
```

This concludes the helper modules. The other modules are much more complex and will not only have their interfaces described but I will also discuss their internal structure and implementation.

2.6 The Phantom server

The Phantom server is the core module putting all communicating modules together. I have chosen an approach, where every incoming traffic arrives at the same listening port, which makes some kind of dispatching necessary and also poses some problems, which I will discuss later. The Phantom server is implemented as a single module.

2.6.1 server

The basic interface to the server module is quite simple:

Listing 2.14: The server module's base interface

```
1 int start_server(const struct config *config);
2 void stop_server(void);
```

The `start`-function starts the server process, the listening socket information and certificates are taken from the `struct config`. There should of course only be one running server per Phantom-application, not necessarily per Phantom node though. The `stop`-function stops the server and with it any network communication.

In order to dispatch incoming packets to the DHT the server has to know if the DHT is currently running or not. There are two functions called by the `start` and `stop` functions of the DHT to inform the server of its state. If the DHT is running and has informed the server process by calling the `kad_running` function, the server will inspect each incoming connection if it is meant for the DHT and dispatch it accordingly. 32-bit magic numbers have been randomly chosen that are stated first on any incoming DHT-connection. According to these numbers, the server calls the right DHT-functions and does not process this connection any further in case the DHT-functions return successfully. If they return with an error, the server tries to interpret the incoming connection as a request by another Phantom node to participate in the construction of a new routing path and handles it accordingly. If the DHT is not running, the server will just throw incoming DHT-packets away. It is critical to inform the server when the DHT is stopped.

Listing 2.15: The server module's interface to register or deregister a running DHT

```
1 void kad_running(void);
2 void not_kad_running(void);
```

In addition, the Phantom server module offers the functionality to register and wait for a specific incoming connection, which is needed by the path and tunnel module. The interface used for this purpose consists of three functions:

Listing 2.16: The server module's interface for awaiting connections

```
1 struct awaited_connection *register_wait_connection(const char *ip,
      const uint8_t *id);
2 int wait_for_connection(struct awaited_connection *w, int timeout);
3 void free_awaited_connection(struct awaited_connection *w);
```

The `register_wait_connection`-function is used to inform the server of an incoming awaited connection from another node. The information passed in is used to identify the connection on arrival. It has to come from a certain IPv4-address and state a certain 160 bit id. The `wait_for_connection`-function is used from a calling thread to block itself for a certain time until the required connection has been established or the timeout occurs. At this point the server will have filled in the connection details into the `struct awaited_connection` and pass control of the connection on to the calling thread. After the calling thread has received the information it needed, it should call `free_awaited_connection` to free the connection and related data. If a thread does not want specific members of the struct to be freed, it can set those pointers to `NULL` and has to free them by calling the appropriate functions later. It is important not to send a message to another node before registering the awaited connection for the reply, in order to avoid race conditions. Should the other node have replied before the awaited response connection was registered, the server may just have thrown it away already.

The following code excerpt taken from the path module shows the typical use of this interface:

Listing 2.17: Sample code for awaiting a connection

```
1  struct awaited_connection *wait;
2  wait = register_wait_connection(path->sps[path->nnodes - 2].next_ip,
       path->sps[path->nnodes - 1].next_id);
3  if (wait == NULL) {
4          /*error*/
5  }
6  ret = send_setup_array(path, config, array, outsize, 1);
7  if (ret != 0) {
8          /*error*/
9  }
10 ret = wait_for_connection(wait, TMOUT);
11 if (ret != 0) {
12         /*error*/
13 }
14 /* handle the incoming connection and data */
15 free_awaited_connection(wait);
```

First the information identifying the incoming connection is passed to the server. Then the packet triggering the reply on the other node is sent and the reply is awaited. After processing, the awaited connection is freed. In order to wait for another connection with the same details again, `register_wait_connection` would have to be called again.

The `struct awaited_connection` has the following members, most of them are for internal use only and should not be used from other modules without fully understanding their purpose:

Listing 2.18: struct awaited_connection

```
1  struct awaited_connection {
2          struct awaited_connection *next;
3          struct awaited_connection *prev;
4          uint8_t id[SHA_DIGEST_LENGTH];
5          int permanent;
6          char *ip;
7          sem_t sem;
8          sem_t entry_ok;
9          X509 *incoming_cert;
10         struct ssl_connection *incoming_conn;
11         uint8_t *incoming_package;
12         uint32_t len;
13         struct timespec timeout;
14 };
```

- id - the awaited id
- ip - the awaited ip
- incoming_cert the X509 certificate stated by the connecting node
- incoming_conn the incoming SSL-connection
- incoming_package the first packet that was stated through the connection it is not changed by the server but has to be received and inspected to see if the right id was stated or if it is to be thrown away
- len - the length of the incoming packet
- next, prev - list interface - used internally
- sem - used to block the calling thread in `wait_for_connection` - used internally
- permanent - marks this `awaited_connection` for multiple use if set. This means multiple connections with the same details are awaited, without having to reregister the connection - used only internally
- entry_ok - used for premanent `awaited_connection`s internally. the thread registering the awaited connection posts on this semaphore to signal it has gotten the data out of the last `awaited_connection` and the data in the struct can now be overwritten

- timeout - the timeout for an awaited connection - used internally

So far for the exported functionality and interfaces of the server module. The central datastructure of the server module is a statically allocated `struct server` which is a module-global variable, so it is easily accessible by all functions inside the server module. It holds different status information about the whole module, for example a list of running threads that need to be joined when the server is stopped, the server's credentials, etc.

Listing 2.19: struct server

```
1  struct server {
2          struct worker_pid pid_list;
3          struct awaited_connection awaited_list;
4          pthread_mutex_t awaited_mutex;
5          pthread_mutex_t worker_pid_mutex;
6          const struct config *config;
7          int init;
8          int quit;
9          int kad_running;
10         uint16_t port;
11         const char *ip;
12         pthread_t thread;
13         int listen_sd;
14         SSL_CTX *ctx;
15         X509 *certificate;
16         EVP_PKEY *privkey;
17 };
```

Internally the server module handles all kind of traffic and dispatches it to various other modules. It is also responsible to start the threads which perform the Phantom node tasks, i.e. they construct paths and tunnels or forward traffic.

The main loop of the server simply waits for an incoming connection, accepts it and dispatches it to a thread in the thread pool before waiting for the next connection. The dispatched function is called `worker`. This function does the SSL-handshake with the connecting node and gets its certificate. If no client-certificate is provided, the connection is closed immediately and the function returns. I have made it a prerequisite that every node participating in the Phantom network must have a valid certificate that is stated with every connection. If a certificate was provided, the worker reads a packet from the connection. The contents of this packet are used to decide how to proceed further. If the packet is prefixed with a DHT-magic number and the DHT is known by the server to be running, the appropriate DHT-handle function is called and if it returns successfully the connection is closed and the function returns. If there is no DHT-magic number or the handle-function fails (this usually indicates a falsely identified DHT-packet. i.e. a packet that looks like a DHT-packet but is truly something else, so the magic number is there only by coincidence) the worker goes through

the list of awaited connections to see if this new connection is part of a path or tunnel creation process in progress. If so, there is a thread waiting for this connection and blocked on the semaphore associated with the corresponding `struct awaited_connection`. The packet and connection details are then inserted into the struct and the semaphore is posted on. The worker will return. The previously blocked thread will carry on with the awaited connection and proceed with the path or tunnel creation. This will be discussed in more detail later on.

If the packet was not awaited and not a DHT-packet it is either invalid or requests the receiving node's participation in a new routing path to be. The worker will then call to the path-module passing in the received packet. The path-module will decide if the request is valid and process it accordingly. This process will be discussed in detail later in the description of the path module. If the packet was not a valid first round package, the connection is closed and the function returns. Otherwise the worker will read the second round package from the connection and hand it to the path module again for verification and processing. If the second round package was also valid, the node now knows the role it has to play in the routing path. If it is just a Y-node it is not participating in the finished routing path and all work is done. The worker returns.

If the node is to be an X-node however the `become_x_node` function is called and the required data is passed to it. These are namely the contents of the routing path setup package and the connections needed for the routing path. These steps are visualized in figure 2.1.

If the `become_x_node`-function is called, the function starts by waiting on the passed in `awaited_connection` from the previous X-node and receives the X-package from this node. The connection to the next X-node is then created and the package is passed along. There are two possibilities now, either the node is the terminating X-node or an intermediate X-node. The former case will be discussed soon, in the latter case the X-node spawns two more threads, which try to read a tunnel initialization package from the neighbouring X-nodes (this has to be done on both sides, because there is no way of knowing from which side this package will come. Once one of the threads has received this package, the other one is canceled and the tunnel creation process is started. The tunnel creation process and its implementation will be discussed in the description of the tunnel module. Receiving tunnel initialization packages is done in a loop until one of the X-node connections is closed down, which closes the path.

If the node is the terminating X-node, another function is called instead of waiting for tunnel initialization packages. The `become_tx_node`-function is again twofold. If the created path is an entry path, the thread calls upon the netdb-module to update the routing table information within the DHT in order to make itself known as the entry node for a certain AP-address. Once this is done, it starts waiting for incoming entry tunnel creation requests. Here the infrastructure to await connections plays a vital role again. Two things are different though. First the awaiting of entry tunnel requests can come from any IP, and

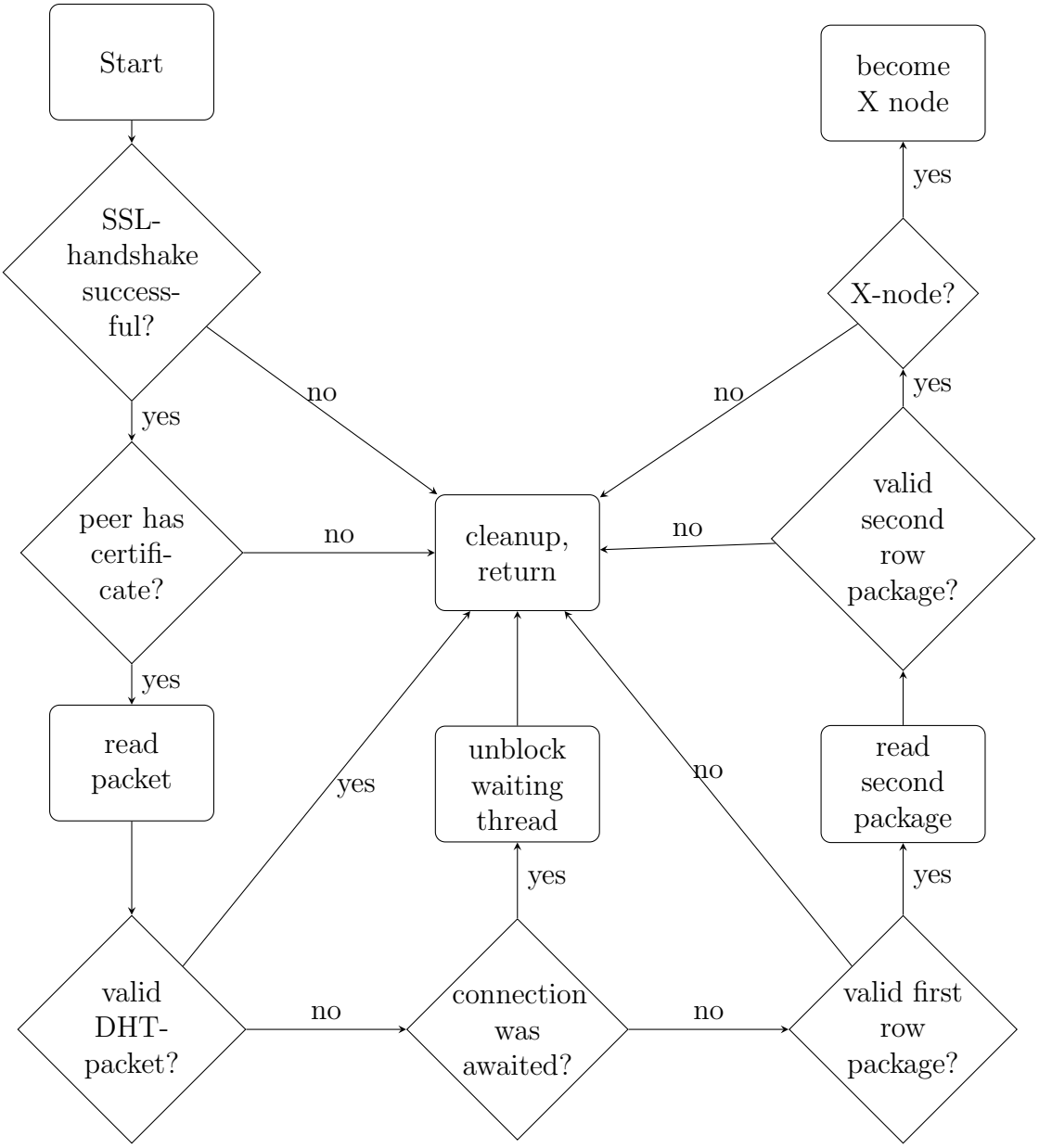

Figure 2.1: Flowchart of the worker thread

so the IP to expect is not known in advance and cannot be used for identifying the package. So the `id` alone is responsible to identify the incoming connection as an entry tunnel creation request. Second the connection is not awaited just once but usually multiple times. That is what the additional members of the `struct awaited_connection` (marked for internal use before) are used for. If the permanent-member is true, the struct will not be removed from the awaited

list once its awaited connection has come but will instead be locked until the responsible thread (the thread currently discussed) has gotten the required information and unlocks it again so new information of another incoming request may be entered by a worker thread. After an entry tunnel creation request has come, a new thread called entry worker is started which builds the tunnel and forwards traffic over it afterwards. After the entry worker has been created, the thread will block again and wait for the next incoming entry tunnel request.

If the terminating X-node is to be an exit node, it tries to read a tunnel initialization package from the previous X-node (originally initiated by the owner of the routing path) and spawns an exit worker to create the tunnel and then relay the traffic through the tunnel.

2.7 Routing paths

The implementation of the routing paths consists mainly of the path module. In addition there are four other small modules involved. The conn_ctx module, the setuppackage.pb-c module, the node_info module and the rc4rand module. I will first describe the four simpler modules and then the main module.

2.7.1 node_info

This is not a module but simply a header file. It declares a struct used for the internal representation of Phantom nodes during path creation and some flags which are used in the setup packages to tell the other nodes their role in the path creation process.

Listing 2.20: struct node_info and related defines

```
1  struct node_info {
2          X509 *construction_certificate; /* path building certificate
               */
3          struct X509_flat *construction_certificate_flat;
4          X509 *communication_certificate; /* communication
               certificate */
5          struct X509_flat *communication_certificate_flat;
6          char *ip;
7          uint16_t port;
8          uint32_t flags;
9  };
10
11 #define X_NODE (0x01)
12 #define Y_NODE (0x02)
13 #define T_NODE (0x04)
14 #define ENTRY_NODE (0x08)
15 #define SUCCESS_FLAG (0x10)
16 #define RESERVE_AP (0x20)
```

2.7.2 conn_ctx

The conn_ctx module mainly declares the `struct conn_ctx` and provides an allocation and free-function for it. This struct is used mainly by the server module and stores the relevant information extracted from the setup packages during path creation. This information is however filled in by the path module since this is where the validation and extraction of the setup packages happens. That is why it is described in this section rather than in the server section.

Listing 2.21: struct conn_ctx, related structs and functions

```
1  struct xkeys {
2          int nkeys;
3          uint8_t *keys;
4          uint8_t *ivs;
5  };
6
7  struct rte {
8          uint32_t len;
9          uint8_t *data;
10 };
11
12 struct conn_ctx {
13         uint8_t prev_id[SHA_DIGEST_LENGTH];
14         uint8_t next_id[SHA_DIGEST_LENGTH];
15         char *prev_ip;
16         char *next_ip;
17         uint16_t prev_port;
18         uint16_t next_port;
19         uint32_t flags;
20         uint8_t *peer_id; /* used for terminating nodes */
21         char *peer_ip; /* used for terminating nodes */
22         uint16_t peer_port; /* used for terminating nodes */
23         X509 *peer_cert; /* used for terminating nodes */
24         struct xkeys *keys;
25         struct ssl_connection *to_next;
26         X509 *prev_communication_certificate;
27         X509 *next_communication_certificate;
28         X509 *routing_certificate;
29         RSA *construction_certificate;
30         /* optional */
31         struct in6_addr ap;
32         struct rte rte;
33 };
34
35 struct conn_ctx *new_conn_ctx(void);
36 void free_conn_ctx(struct conn_ctx *conn);
```

The two small structs declared first represent the keys and ivs used by an X-node to encrypt the traffic passed along through the tunnels. The rte-struct represents an already serialized routing table entry as len bytes of data.

The members of `struct conn_ctx` have the following meaning:

- prev_id - the unique id of the previous node in the path
- next_id - the unique id of the next node in the path
- prev_ip - the ip of the previous node in the path
- next_ip - the ip of the next node in the path
- prev_port - the port of the previous node in the path
- next_port - the port of the next node in the path
- flags - flags which identify the role of a node (these are the flags defined in the node_info module
- peer_{id,ip,port,cert} - are only set in terminating X-nodes and point to the X-node connected to the terminating node, since there is no clear previous and next node anymore in a finished routing path.
- keys - the xkeys used by an X-node (unused in Y-nodes)
- to_next - the connection to the next X or Y node, assuming there is one
- {prev,next}_communication_certificate - the communication certificates of the next X- or Y-node
- routing_certificate - the routing certificate of the routing path owner
- construction_certificate - the construction certificate for this specific path
- ap - the AP-address of the routing path owner (in case of an entry path)
- rte - the routing table entry of the routing path owner (in case of an entry path)

The `free`-function frees the struct and its members. If the caller does not wish certain members to be freed, he can set those pointers to `NULL` and has to free them later.

2.7.3 setuppackage.pb-c

This module is automatically generated by libprotobuf-c which has been described earlier on. The prototype-file used to generate this module describes four message types. A dummy setup package consisting of the seed used to generate the package, its size and flags which describe if the package is to be inserted or deleted. Second a routing table entry which consists of an AP-address of the anonymized node and a list of entry node IPs and ports. The `setup_package`-message holds the contents of a setup package. Last but not least, the `setup_array`-message is a full setup array passed around during the creation of a routing path. It simply consists of a number of already serialized setup and dummy packages.

Listing 2.22: Message formats used inside a setup array

```
1  message dummy_setup_package {
2          required bytes seed = 1;
3          required fixed32 size = 2;
4          required fixed32 flags = 3;
5  }
6
7  message routing_table_entry {
8          required bytes ap_address = 1;
9          repeated string ip_addresses = 2;
10         repeated fixed32 ports = 3;
11 };
12
13 message setup_package {
14         required string prev_ip = 1;
15         required string next_ip = 2;
16         required uint32 prev_port = 3;
17         required uint32 next_port = 4;
18         required bytes prev_id = 5;
19         required bytes next_id = 6;
20         required bytes prev_communication_certificate_flat = 7;
21         required bytes next_communication_certificate_flat = 8;
22         required bytes construction_certificate_flat = 9;
23         repeated dummy_setup_package dummies = 10;
24         required uint32 nkeys = 11;
25         required bytes key_seed = 12;
26         required bytes replacement_seed = 13;
27         required bytes key_salt = 14;
28         required uint32 flags = 15;
29         required bytes hash = 16;
30         required bytes external_hash = 17;
31         optional bytes ap_address = 18;
32         optional routing_table_entry rte = 19;
33 }
34
35 message setup_array { repeated bytes slots = 1; }
```

2.7.4 rc4rand

The creation of the dummy packages during path creation procedure needs a deterministic, precalculatable but secure pseudo random number generator. The openssl-PRNG is not deterministic and can therefore not be used for this purpose. After implementing the BBS-generator[9] I found it to be much too slow for this purpose and went with the bit stream generated from a symmetric stream cipher (RC4) as PRNG. This module implements the RC4-based PRNG used to create and precalculate the dummy packages given a seed and a salt. The interface is quite simple and so this module can easily be replaced by some other PRNG. The PRNG should be well chosen since it cannot easily be altered once the protocol is used in the real world. The interface to the PRNG is quite simple and should need no further explanation:

Listing 2.23: The rc4rand module's interface

```
1 struct rc4_rand *rc4_rand_init(const uint8_t *seed, int len);
2 void rc4_rand_bytes(struct rc4_rand *r, uint8_t *buf, int len);
3 void rc4_rand_free(struct rc4_rand *r);
```

Internally I simply call to the RC4-cipher functions of libopenssl passing all zero bits as input. Since symmetric stream ciphers work by xoring the input with a stream of pseudorandom bits (generated predictably and therefore reproducibly given the starting key and salt), passing zero bits as the input simply gives the pseudorandom bit stream as the output. These bits are then used as the PRNG's pseudo-random bit stream.

2.7.5 path

The path module is the main module of the routing path implementation. It plans and constructs routing paths. The exported interface is again quite simple:

Listing 2.24: The path module's interface

```
1 struct path *construct_entry_path(const struct config *config);
2 struct path *construct_exit_path(const struct config *config);
3 struct path *construct_reserve_ap_path(const struct config *config);
4 void free_path(struct path *path);
5 uint8_t *handle_first_round_setup_array(const struct config *config,
      const uint8_t *sa, int sa_len, const uint8_t *id, const char
     *from_ip, struct conn_ctx *conn, uint32_t *outsize);
6 uint8_t *handle_second_round_setup_array(const struct config
     *config, const uint8_t *sa, int sa_len, const uint8_t *id, const
     struct conn_ctx *oldconn, struct conn_ctx *conn, uint32_t
     *outsize);
```

The first three functions construct an entry path, an exit path or a special path which I call AP-reservation path. The AP-reservation path is basically an exit path that is solely constructed and used to reserve an AP-address from the DHT.

This is necessary, because the DHT node reserving the AP-address would otherwise know the reservers true identity. The other two functions are used by the server module in case a possible first or second round setup array has been received. The server calls these functions to unpack the setup packages and to validate them. These two functions take a lot of arguments which should be explained:

- config - a pointer to the `struct config`
- sa - the packed setup array
- sa_len - `sa`'s length in bytes
- id - the unique id stated by the node along with the setup array
- conn - the connection context to be filled in
- outsize - the size of the new setup array after the modifications on it have been done
- oldconn - in case of the second round the connection context filled in by the previous call in the first round

The central data structure for the path module is `struct setup_path`. I will present this data structure first and then talk about the internal structure and functions of the module.

Listing 2.25: struct setup_path

```
1  struct setup_path {
2          struct node_info *nodes;
3          uint8_t nxnodes;
4          uint8_t nynodes;
5          uint8_t nnodes;
6          uint32_t *sizes;
7          uint8_t **contents;
8          struct setup_package *sps;
9          int construction_certificate_len;
10         uint8_t *construction_certificate_data;
11         uint8_t endhash[SHA_DIGEST_LENGTH];
12         uint32_t entrypath;
13         struct ssl_connection *ssl_conn;
14         /* is_reverse_path == 1 if we start with many y nodes */
15         int is_reverse_path;
16         int reserve_ap_address;
17         char *entry_ip;
18         struct in6_addr ap;
19         RSA *construction_certificate;
20         const X509 *routing_certificate;
21         struct X509_flat *routing_certificate_flat;
22 };
```

- nodes - information of the X- and Y-nodes chosen to participate in the routing path
- nxnodes - the number of X-nodes in the path
- nynodes - the number of Y-nodes in the path
- nnodes - `nxnodes + nynodes`
- sizes - the sizes in byte of the setup packages for the chosen nodes
- contents - the packed setup packages
- sps - the unpacked or not yet packed setup packages
- construction_certificate_len - the length of the construction certificate in bytes
- construction_certificate_data - the construction certificate for this path
- endhash - the precalculated hash over the setup array after receiving it from the last node in the construction process round, including all modifications and dummy packages
- entrypath - flag set if the struct belongs to an entry path
- ssl_conn - the SSL-connection to the first node of the path
- is_reverse_path - flag set if the circle of nodes constructed during the setup phase has many Y-nodes in the front[8]
- reserve_ap_address - flag set if the path constructed is an address-reservation-path
- entry_ip - the IP-address of the terminating X-node in an entry path
- ap - the AP-address reserved for this this path
- construction_certificate - the construction certificate used for this path
- routing_certificate - the routing certificate of the anonymized node
- routing_certificate_flat - and its serialized form

[8]There are either many consecutive Y-nodes at the front or at the end of a path during the construction phase.

The three exported unary path-construction functions are internally mapped to the same ternary function: `static struct path *construct_path(const struct config *config, int want_entrypath, int reserve_ap)`. The two additional flags are used to signal which flavor of a path is to be constructed. I have chosen this approach, because huge parts of the construction-process are identical and so it is easier to have some branches taken or not taken depending on the passed in flags than to implement the three variations independently. This adds complexity within the functions but reduces the number of code lines drastically.

The path building process is divided into two parts. First, the planning of the path and preparation of the setup process, which takes part solely in the node initiating the construction process. Second, the actual construction of the path, which mainly consists of passing around setup arrays. The initiating node starts by creating the `struct setup_path` and filling in the required information. It therefore queries the DHT for the contact information of other Phantom nodes until it has the required number of nodes to construct the path. Those contact information consist of the IP, the port, the communication certificate and the construction certificate for each selected node and are represented by a `struct node_info`.

The node then calls the `build_xy_path`-function. This function flags the nodes according to the role they should play in the creation process. If there are more Y-nodes than those strictly needed by the protocol, they are mixed randomly in between the other nodes. Finally the order of all the nodes in the path is reversed randomly with a 50% chance as required by the protocol. The layout of the path to be is now finished and the planning phase ends.

The next step is to prepare the setup array. This is done by first generating the path-construction-key as a new RSA key-pair. This key-pair is solely used for a single path. The `generate_setup_packages`-function fills in the required information for the setup package for each node. It generates random seeds for the xkeys to be used in later tunnels and the dummy package creation process, if required. Additionally it fills in the previous node's and next node's contact information, unique id and the path construction keys. All that is missing now is the information and precalculation of the dummy packages and modifications carried out by each participating node. These precalculations are rather tedious and I came up with an alternate method to do this than the one proposed in the original design. This will be discussed later in more detail.

With all this information composed, the setup array consisting of the packed setup packages can now be created. The setup packages itself can be serialized using libprotobuf-c. The protocol requires however for the individual packages to be symmetrically and asymmetrically encrypted, signed and to carry a hash over the full setup array in its expected state of arrival at each node within its setup package. These requirements make the generation of the setup array a hard process. In order to be able to insert the hash of the expected state of arrival

for the setup array as a whole, the setup array has to be packed backwards (i.e. starting with the slot in the array that corresponds to the last node receiving the array, then its previous node and so on).

The packing process for each setup package starts by adding the expected hash of the state of arrival into the setup package. The setup package is then packed (with its own hash set to all zero bytes), the resulting bytes are then hashed again and this hash is filled into the setup package as its own hash. The package is then repacked (because the new hash has been entered) and the previously packed version is thrown away. The package is asymmetrically encrypted with its construction certificate and signed with the private construction key of the anonymized node to be. Since asymmetric cryptography is expensive, the encryption is done using an enveloping technique. The package itself is encrypted using a symmetric cipher whose key is encrypted asymmetrically with the recipients public key. The asymmetrically encrypted symmetric cipher key is then prepended to the message along with the symmetric ciphers iv. The recipient recovers the symmetric key by decrypting it asymmetrically using its private key and then decrypts the package itself using the symmetric cipher with given key and iv. The symmetric cipher's key and iv are prepended to the actual setup package. The described triple is then encrypted symmetrically using the recipient's preprocessor's unique id as a symmetric key. Finally the iv for this encryption is prepended to the package again. All resulting data is then signed with the path owners private key and the signature prepended. The length of the RSA-signature is currently for 2048-bit certificates. If a different size or mixed size certificates should be used by the protocol, the length of the signature should be prepended. Once the slots have all been serialized in the described way, they are packed as as a message consisting of a number of (byte, length) pairs using libprotobuf-c. The unique id expected by the first node is prepended and the total length of the setup package data and unique id prepended again as a 32-bit big endian value. The "wire format" for a full setup array can be seen in figure 2.2.

The initiating node registers the awaited connection on which the setup array will return to him and sends the data to the first node in the setup chain. The unpacking and modification actions taken by each node along the way will be discussed a little further down in this section. After the path owner has received the array back, it hashes it and checks to see if it is in the expected state. If so, a second round setup array is created in very much the same way as the first round setup array and sent away again. If the second round setup array is also received in the expected state after passing through all the other nodes, the X-package is sent as described in the paper to finish the construction process.

When a setup array is received by a node's server, it calls to the path module for verification, unpacking and extraction of the setup package and receives the extracted information in a `struct conn_ctx`.

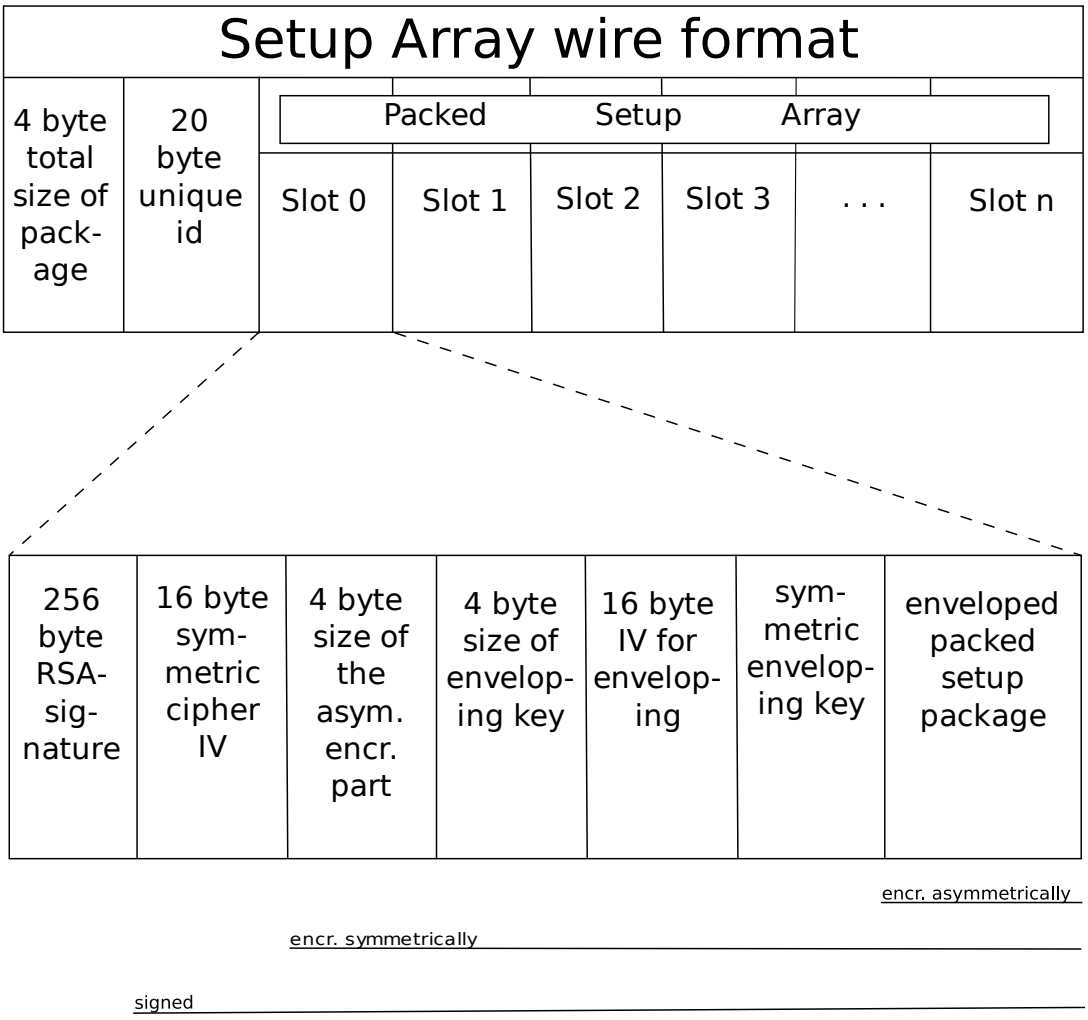

Figure 2.2: Wire format of a full setup array and an example slot

The path module starts the verification process by splitting the received chunk back into individual slots. It then tries to find its own package candidate by symmetrically decrypting the slots with the stated unique connection id. If the symmetric decryption is successful can not be decided before the padding at the end of the package has been decrypted. After successfully decrypting a package, it tries to open the enveloped asymmetrically encrypted setup package itself. If the contents can be successfully recovered, the setup package is unpacked and the contents are checked if they are complete. The internal hash of the package is then set to zero, the package is repacked and the hash is compared to make sure the contents have not been modified. To make sure the right package has definitely been found, further checks are made. The connection id, stated by the sending node is compared to the expected one from the setup package. Next the

IP from which the connection was initiated is checked to see if it matches the previous IP from the setup package. If so, the array is hashed in full (excluding our own setup package slot) to make sure it has not been tampered with by a previous node. If all these checks are successful, the setup packages contents are extracted into a `struct conn_ctx` and the node, now knowing its role, decides if it wants to participate in the path being constructed. If it participates, it modifies the setup array according to the instructions received within the setup package, repacks it and sends it along to the next node, which will go through the same process again. The servers thread dealing with this connection will block and try to read the second round setup array from the connection. Once it arrives, it is checked by the path module in very much the same way as the first round setup array. The main difference in the second round is that the RSA-signature is now also checked using the construction certificate passed along in the first round, and it is checked if the `SUCCESS_FLAG` is set in the flags of the second round setup package.

The result of a successful path construction process is a newly constructed entry or exit path ready to create the first tunnel over it. The information needed for tunnel creation are passed out to the caller in a `struct path`:

Listing 2.26: struct path

```
1  struct path {
2          uint8_t nkeys;
3          struct xkeys **xkeys;
4          int is_entrypath;
5          struct in6_addr ap;
6          struct ssl_connection *conn;
7          uint8_t peer_id[SHA_DIGEST_LENGTH];
8          char *peer_ip;
9          uint16_t peer_port;
10 };
```

The members of the struct have the following meaning:

- nkeys - the number of xkeys available at the X-nodes in the path
- xkeys - the xkeys themselves
- is_entrypath - flag set if the path is an entry path
- ap - the AP-address reserved for this path
- conn - the SSL-connection to the first X-node in the path
- peer_id - the unique id used for the communication between the anonymized node and the first X-node during path creation
- peer_{ip, port} - the port and IP of the first X-node in the path

2.8 Routing tunnels

The Routing creation is mainly implemented in the tunnel and server module. Tunnels can only be constructed once a path has been successfully constructed.

2.8.1 tunnel

The tunnel module exports five functions to be used by other modules and an additonal five functions used by the server module:

Listing 2.27: The tunnel module's interface

```
1  struct tunnel *create_tunnel(struct in6_addr *ap, const struct path
       *path);
2  struct tunnel *await_entry_tunnel(const struct in6_addr *own_ap,
       struct in6_addr *remote_ip, const struct path *path, const struct
        config *config);
3  int tunnel_read(struct tunnel *t, uint8_t *buf, int num);
4  int tunnel_write(struct tunnel *t, const uint8_t *buf, int num);
5  void free_tunnel(struct tunnel *t);
6  /* needed by server */
7  struct tunnel *create_ap_reservation_tunnel(const struct path
       *path);
8  struct tunnel_dummy_package *create_tunnel_dummy_package(const
       uint8_t *received, const struct conn_ctx *conn);
9  uint8_t *decrypt_tunnel_block(const struct tunnel_dummy_package *dp,
        const uint8_t *data);
10 int extract_exit_init_reply_package(const uint8_t *received, struct
       in6_addr *ap);
11 int extract_entry_init_reply_package(const uint8_t *received,
       uint32_t *flags);
```

The `create_tunnel`-function is used to create a new outgoing tunnel to the passed-in AP-address over an exit path. The `await_entry_tunnel`-function provides a way to wait for an incoming tunnel over an entry path. The `free_tunnel`-function shuts down a previously constructed tunnel and frees its allocated resources. The read and write functions are used to read to and write from a tunnel. They handle the onion encryption and decryption. So the data written to the write-function on one end of the tunnel is identical to the data received by a call to the read-function on the other end of the tunnel.

The result of a successful tunnel creation process is a `struct tunnel`:

Listing 2.28: struct tunnel

```
1  struct tunnel {
2          int nkeys;
3          int is_entry_tunnel;
4          pthread_t tid;
5          int quit;
6          EVP_CIPHER_CTX *ectxs;
```

```
7         EVP_CIPHER_CTX *dctxs;
8         struct ssl_connection *conn;
9 };
```

- nkeys - the number of xkeys. Once a tunnel has been successfully created the xkeys will have been stored within the cipher contexts listed below, so there are no `struct xkeys` necessary anymore
- is_entry_tunnel - flag set if the tunnel is an entry tunnel
- tid - used by the frontend implementation, not strictly part of a tunnel
- quit - used by the frontend implementation, not strictly part of a tunnel
- ectxs, dctxs - the cipher contexts used for the onion en- and decryption of this tunnel, the contexts are updated by calls to the `tunnel_read` and `tunnel_write` functions.
- conn - the SSL-tunnel-connection to the first X-node in the routing path

During the implementation of the routing tunnels, I did not encounter many problems, so I will not discuss it here broadly. It is implemented exactly as described in Magnus' paper. I have chosen the size of the tunnel initialization and dummy packages as 48 bytes. It is the smallest multiple of the chosen cipher's block size providing enough room for the contents that need to be transported in this packages as shown in figures 2.3 and 2.4. The remaining bytes are used to store the first bytes of the SHA-1 hash over the contents for verification. This are 8 bytes for an entry tunnel and 16 bytes in case of an exit tunnel. The crypto initialization block consists of two 32 bit integers a and b. a is chosen at random and b is chosen as $\neg a$ (its logical inverse). This makes it possible to quickly check if the brute-forcing step has successfully recovered the used xkeys. For exit tunnels this initialization block is doubled, because there was space available inside the package and this further reduces the chance of false positives during the brute force phase.

Content	size (bytes)
crypto key initialization block	2 * 4
ip address of connecting node	16
AP-address of the node to connect to	16
part of SHA-1 hash over the previous contents	8
total	48

Figure 2.3: Data transported for an entry tunnel

Content	size (bytes)
crypto key initialization block	4 * 4
AP-address of the node to connect to	16
part of SHA-1 hash of the previous contents	16
total	48

Figure 2.4: Data transported for an exit tunnel

2.9 Distributed kademlia-like hash table

Compared to the other two parts of the original protocol-description, the description of the distributed hash table is rather vague and generic. After doing a bit of reading, I found the kademlia design as proposed by Petar Maymounkov and David Mazieres[11] suitable.

2.9.1 The kademlia design

Kademlia is a communications protocol for peer-to-peer networks and can be used to implement a DHT. A specific implementation is characterized mainly by three parameters:

- α - used to describe the degree of parallelization in RPC-calls (3 in my implementation, which seems to be optimal[3])
- B - the size in bits of the unique id used to identify nodes and pieces of data (160 in my implementation - the length of a SHA-1-hash)
- k - the maximum number of contacts stored in a bucket (20 in my implementation)

A kademlia network consist of cooperating nodes which exchange and store data and communicate with each other. Each node is identified by a quasi-unique B-bit id called the node id. Pieces of information stored in the DHT are also identified by an B-bit id called its key. Data is stored and searched for at the nodes nearest to the data's key. To decide if a key is more or less distant from a node id, a metric is needed. Kademlia uses the xor-metric for this purpose, which is defined as follows:

The distance between two ids x and y is the result of the operation $x \oplus y$ (x xor y) interpreted as a big endian binary number of length B. This means if two nodes are close to each other the most significant bits of the resulting distance vector will be zero, i.e. the resulting number will be small.

Each kademlia node organizes the contacts known to it in B buckets, each having up to k entries. These buckets are called k-buckets. The k-buckets are organized by the distance between the node's id and the contact's id, If the node

ids differ first in the n-th bit, the node is inserted into bucket n. The following code snippet taken from the kademlia module shows how this is implemented. `own_id` is the nodes own id, the passed in `id` is the contacts id. `logs` is a 256-byte table holding the floored binary logarithms for the values 0 – 255. The function returns the index of the k-bucket the new contact has to be stored in or -1 if the passed in id is the node's own id, which should theoretically not happen.

Listing 2.29: The bucket_idx function

```
1  static int
2  bucket_idx(const uint8_t *id)
3  {
4          int i, tmp;
5          for (i = 0; i < 160 >> 3; i++) {
6                  if (! (tmp = id[i] ^ kad->own_id[i])) {
7                          continue;
8                  }
9                  return 160 - ((i + 1) << 3) + logs[tmp];
10         }
11         return -1;
12 }
```

Furthermore contacts are sorted within the buckets by the time of the most recent successful communication with them.

To prevent Sybil attacks[4], I require each node's node id to be the SHA-1 hash over its serialized communication certificate. Sybil attacks are possible if a node can choose its node id freely and therefore position itself at or near a specific position within the overlay network. For example an attacker could try to remove certain data from the DHT by clustering fake nodes around the data (i.e. nodes with node ids near to the data's key) and then accept every incoming store request but never return the data stored if requested to do so.

The kademlia protocol specifies four rpc messages:

- ping - used to see if a node is still alive
- store - to store data at a specific node
- find node - to find a node given a specific key
- find value - to find data given a specific key

I have decided to implement the kademlia design not via UDP but with SSL-connections. So the DHT traffic is also encrypted and since it uses the Phantom server who expects all incoming traffic to be SSL, it made integration a lot easier.

In the original kademlia design all rpc messages have to carry a quasi-unique id that is repeated in the reply to make it possible for the sender of the rpc call to match an incoming reply to an outstanding request. By using TCP as the underlying transport protocol, this is no longer necessary since the reply can

simply be sent back via the already established connection that was used to send the request.

Three of the four kademlia rpcs are implemented as rpcs within the kademlia_rpc module. Ping is not implemented as a rpc. A successful ping in my implementation is done by creating an SSL-connection to the ping target and requesting its communication certificate. If the stated certificate matches the node id the ping is successful.

The other three rpcs are required to work as follows: the sender of the store rpc sends a key and a block of data to the recipient and asks for it to be stored for later retrieval. The find node rpc includes a B-bit key. The recipient then returns up to k contact informations which are the closest it knows to the key. The node may return only less than k contacts if it returns all the nodes within its k-buckets. A find value rpc includes also a B bit key and the recipient replies either with the data belonging to the key if he has it in his local store or handles the find value rpc as a find node rpc on the given id. All this operations are primitive.

The procedure finding the k closest nodes for a given id is described as recursive in the original kademlia paper. It is in fact iterative and works as follows:

1. Select up to α contacts from the closest non-empty k-bucket responsible for the id into a shortlist

2. If fewer than α contacts are within this k-bucket, add other contacts

3. note the closest node found so far as `closest-node`

4. Send parallel find value or find node rpc requests to those nodes

5. Remove dead contacts from the shortlist

6. Improve the shortlist with the nodes returned by the find node rpc by adding the closer contacts to it

7. update `closest-node`

8. Repeat the sequence of parallel searches until either no node closer to id than `closest-node` is returned by any of the α contacted nodes or the shortlist holds k probed and valid contacts

9. If the operation requested was a find value operation and data has been returned, return the data to the caller. If not, return the k nodes from the shortlist.

This algorithm is called iterative-find node and is the core routine used to implement the DHTs functionality. The store-operation does an iterative-find node

first and then sends a primitive store rpc to the k returned nodes. The find node and find value operations use the algorithm as is.

The kademlia design also describes certain rules for data expiration and refreshing, since I have not implemented these I will not discuss them here.

2.9.2 Kademlia module overview

DHT-design and implementation is not for the faint of heart and since a good design and choice of a DHT-model is crucial for the success of the Phantom protocol, this part of the implementation definitely needs improvement and a lot of more work which simply was beyond the scope of my thesis. So what I have implemented works to a certain point and makes the implementation of the rest testable and usable in a controlled environment. Nonetheless I will discuss my basic implementation here and the problems with it will make up a good part of the later problems section.

My implementation consists of six modules. The main module implementing the algorithms and logic of the kademlia design is the kademlia module. The kademlia_rpc-module contains the functions processing rpc calls made by the DHT to other Phantom nodes using the protobuf-c messages from the automatically generated kademlia.pb-c-module. netdb provides the interface and some glue code used to communicate with the DHT from the modules which do not belong to the DHT-implementation. Finally the diskcache module provides a simple way to store the data inserted into the DHT on a participating nodes disk.

2.9.3 diskcache

The diskcache module is mainly for testing. It simply stores the data on a given directory at disk, using its key as a filename. The data can also be retrieved. I am not describing this module in detail because it should be rewritten and changed to something better suited.

2.9.4 kad_contacts

This module implements the functionality to save contacts to disk and to restore them. Some initial contact has to be provided if the DHT is started for the first time in order to make it possible to contact another node to help with the bootstrapping process of kademlia. The module exports two functions used by the main kademlia module:

Listing 2.30: The kad_contacts module's interface

```
1 int save_contacts(const char *filename, struct kad_table *table);
2 int restore_contacts(const char *filename, struct kad_node_info
     *contacts);
```

The first function serializes the contacts in a simple way and stores them on disk to the given file, the `kad_table` is part of the kademlia main module and holds the contacts known by the node. It will be described later on. The second function can be used to restore the contacts from this file. They are passed out to the caller as a list in the `contacts`-parameter.

2.9.5 Kademlia

The kademlia module implements the kademlia logic and algorithms and declares the central datastructures. Those are the `struct kad_node_info`, the `struct kad_table` and the `struct kad`.

Listing 2.31: struct kad_node_info

```
1  struct kad_node_info {
2          struct kad_node_info *prev;
3          struct kad_node_info *next;
4          uint8_t id[SHA_DIGEST_LENGTH];
5          X509 *cert;
6          X509 *pbc;
7          uint16_t port;
8          struct timespec last_seen;
9          char *ip;
10         int ponged;
11         sem_t sem;
12 };
```

The `struct kad_node_info` is the datastructure representing the contact information of a kademlia-contact as well as the information needed from a node to use it in Phantom's path-building process. The members of the struct are the following:

- prev, next - `kad_node_info` structs can be collected into a list
- id - the node id of the contact
- cert - the communication certificate of the contact
- pbc - the path building certificate of the contact
- port - the listening port of the contact's Phantom server
- last_seen - timestamp when the last sucessful communication with this contact took place

- ip - the contacts IPv4 address
- ponged, sem - used internally by the ping thread (discussed later)

The `struct kad_table` represents the k-buckets:

Listing 2.32: struct kad_table

```
1  struct kad_table {
2          struct kad_node_info buckets[NBUCKETS];
3          int entries[NBUCKETS];
4          struct timespec last_action[NBUCKETS];
5          pthread_mutex_t bucket_mutexes[NBUCKETS];
6  };
```

- buckets - the buckets itself. A bucket is implemented as a list of kademlia contacts
- entries - the number of entries for each bucket (i.e. the number of contacts in it)
- last_action - timestamp for when a contact in this bucket was last communicated with
- bucket_mutexes - one mutex for each buckets used to synchronize operations on the underlying list

The most important structure is the `struct kad` which puts it all together. How the members are used in detail will become obvious during the discussion of the modules internal implementation.

Listing 2.33: struct kad

```
1  struct kad {
2          struct kad_table *table;
3          struct ping_nodes *ping;
4          struct disk_cache *cache;
5          const struct config *config;
6          uint8_t own_id[SHA_DIGEST_LENGTH];
7          struct updates updates;
8          int quit;
9          NodeInfo self;
10         struct thread thread_list;
11         char *nodefile;
12 };
```

- table - the kademlia table presented above
- ping - list of nodes enqueued for the ping rpc

- cache - the diskcache used to store data
- config - the config of the Phantom application
- own_id - the node's node id
- updates - holds contacts which need to be entered into the k-buckets
- quit - flag set if the kademlia module has been asked to shut down
- self - the node's own contact information in protobuf-c form
- thread_list - list of running threads within the module
- nodefile - the file where the contact information are stored

Once the kademlia module is started, it creates a `struct kad` along with all its members and starts two threads called the ping-worker and the update-worker. It then calls into the server module to register itself as ready and running and tries to join the DHT. The procedure for joining the network is quite simple. First the previously stored contacts are retrieved using the kad_contacts module. One by one, the node tries to send these contacts find node rpc messages for its own id, which is a quick way to get to know some other nodes near to it. If the first of these rpc-calls returns successfully it has joined the DHT.

A third thread is then spawned called the housekeeping-worker. The housekeeping-worker iterates over the k-buckets in certain intervals and refreshes their contents if the last action seen on the bucket is longer than KAD_T_REFRESH seconds in the past. Refreshing a bucket is accomplished by calling iterative find node on a randomly generated id falling into the buckets range. Since all buckets will be refreshed shortly after joining the DHT, a certain number of new and valid contacts should soon be available for the node to work with.

The update-worker is responsible for inserting contacts into the k-buckets. It waits on a semaphore inside `struct updates` until another thread relays an update request for it. Contact information should be updated whenever a rpc-call was made to another node successfully. Updates can simply be relayed using the `update_table_relay`-function, this function clones the passed contact information inserts it in a list and posts on the semaphore the update-worker is blocked on. The update worker will then perform the following action:

1. Wait on updates-semaphore
2. Dequeue the first element of the update list
3. Get the bucket index for it

4. Iterate over the bucket's contents to see if the contact is already in the bucket. If so, timestamp this contact and insert it as the first element of the bucket list. Upate the `last_action` timestamp on the bucket and goto 1.

5. Check if the bucket already holds k entries. If not, insert the contact at the head of the bucket's list, increment the nentries counter. Timestamp the `last_action` timestamp and the contacts `last_seen` timestamp, then goto 1.

6. If the bucket was full, remove the current list head of the bucket list and give it to the ping worker to see if it is still alive.

7. Wait for the ping-worker's reply.

8. If the old list head was still alive keep it, if not exchange it with the new contact. Timestamp the bucket. Goto 1.

The ping-worker is responsible for pinging nodes, to see if they are still alive. It works similar to the update-worker. It has a list containing ping-requests and a semaphore to block on if no work is to be done. If a ping request has been relayed, the thread will establish an SSL-connection to the other node and get its certificate. If the hash over the certificate is matching the node's node id, the contact is considered valid and ponged successfully. If not, it has not ponged successfully. The result of the ping process is made available to the requestor via the original request struct.

The kademlia module exports the following functions to other modules. Some are intended to be used only from within other modules implementing the DHT functionality others are for the rest of the Phantom application also:

Listing 2.34: The kademlia module's interface

```
1  /* interface functions for kademlia */
2  int start_kad(const struct config *config);
3  void stop_kad(void);
4  int kad_store(uint8_t *key, uint8_t *data, uint32_t len);
5  int kad_find(const uint8_t *key, uint8_t **data, uint32_t *len);
6  struct kad_node_list *get_n_nodes(int n);
7  void free_kad_node_list(struct kad_node_list *l);
8  void get_free_ap_address(struct in6_addr *ap);
9
10 /* Functions needed from other kademlia modules */
11 int local_find(const uint8_t *key, uint8_t **data, uint32_t *len);
12 int local_store(const uint8_t *key, const uint8_t *data, uint32_t
       len);
13 struct kad_node_list *get_k_closest_nodes(const uint8_t *id, const
       uint8_t *requestor);
14 struct kad_node_info *new_kad_node_info(const uint8_t *id, const
       char *ip, uint16_t port, X509 *cert, X509 *pbc);
```

```
15 void free_kad_node_info(struct kad_node_info *n);
16 void update_table_relay(const struct kad_node_info *n);
```

The start and stop functions bring the kademlia module up and register it with the server module as running or shut it down and deregister it. The store and find functions are used to store data to or retrieve data from the DHT. The `get_n_nodes`-function restores n contacts to other Phantom nodes collected from the k-buckets. Its main use is getting nodes to use in a path creation process. These nodes can then be freed using the `free_kad_node_list`-function. The `get_free_ap_address`-function is used to reserve an unused AP-address for a newly created path. This is currently not much more than a stub, because the DHT (as said above) is not fully implemented.

The `local-{find, store}`-functions provide a way to store data to and retrieve data from the diskcache. The `get_k_closest_nodes`-function returns the k closest contacts for a given id, without the contact whose node id is passed in `requestor`. `update-table-relay` is used to relay a node to the update-worker as described before, this function is called from the kademlia_rpc module after a rpc-call has been handled successfully. The two functions not mentioned are used to create or free a `struct kad_node_info` as the need arises.

2.9.6 kademlia.pb-c

The kademlia.pb-c module is autogenerated via protobuf-c from the following descriptions:

Listing 2.35: The DHT rpc-message formats

```
1  message node_info {
2          required bytes id = 1;
3          required uint32 port = 2;
4          required bytes cert = 3;
5          required bytes pbc = 4;
6          required string ip = 5;
7  };
8
9  message store {
10         required bytes key = 1;
11         required bytes data = 2;
12         required node_info self = 3;
13 };
14
15 message store_reply {
16         required bool success = 1;
17 };
18
19 message find_close_nodes {
20         required bytes id = 1;
21         required node_info self = 2;
```

```
22 };
23
24 message find_close_nodes_reply {
25         repeated node_info nodes = 1;
26 };
27
28 message find_value {
29         required bytes key = 1;
30         required node_info self = 2;
31 };
32
33 message find_value_reply {
34         required bool success = 1;
35         repeated node_info nodes = 2;
36         optional bytes data = 3;
37 };
```

The most important message is the `node_info` message. It is the protobuf-equivalent to the `struct kad_node_info` and is used to send contacts from one node to the other. The other three messages are for the find value, find node and store rpcs. Each message has a request and a reply form. Along with the request, a node sends its own contact, so it can be entered into the k-buckets of the node receiving the rpc-call via `update_table_relay`. The ping-rpc call has no message because it is not implemented as a rpc.

2.9.7 kademlia_rpc

The kademlia_rpc module exports three functions for the kademlia rpc-calls. These three functions have nearly the same interface. They all return a `struct rpc_return` holding the result of the rpc-call back to the caller. As their arguments they take the id, or the data and its length (in case of store), needed for the procedure, the contact information of the node to connect to, the credentials for creating the SSL-connection to this node and their own contact information. The returned struct can be freed by a call to `free_rpc_return` once the needed information has been extracted by the caller.

In addition, three functions are exported to be used by the server module to handle incoming rpc-requests. The server will call these functions as described in 2.6.1.

Listing 2.36: The kademlia_rpc module's interface

```
1 struct rpc_return {
2         int success;
3         int nnodes;
4         struct kad_node_info *nodes[KADEMLIA_K];
5         uint32_t len;
6         uint8_t *data;
7 };
```

```
8
9  struct rpc_return *rpc_find_node(uint8_t *id, const struct
       kad_node_info *n, X509 *cert, EVP_PKEY *privkey, NodeInfo *self);
10 struct rpc_return *rpc_find_value(uint8_t *key, const struct
       kad_node_info *n, X509 *cert, EVP_PKEY *privkey, NodeInfo *self);
11 int rpc_store(uint8_t *key, uint8_t *data, uint32_t len, const
       struct kad_node_info *store_to, X509 *cert, EVP_PKEY *privkey,
       NodeInfo *self);
12 void free_rpc_return(struct rpc_return *r);
13
14 int handle_rpc_find_node(SSL *from, X509 *cert, uint8_t *package,
       int size);
15 int handle_rpc_find_value(SSL *from, X509 *cert, uint8_t *package,
       int size);
16 int handle_rpc_store(SSL *from, X509 *cert, uint8_t *package, int
       size);
```

The three handle-functions are passed the SSL-context from the incoming connection and the matching certificate, as well as the packets received by the server via the connection. They start by trying to extract the received packet as a rpc-message using libprotobuf-c. If this succeeds, the message is validated and if found OK, the kademlia module is called to fulfill the request. For example to find nodes close to a given id or store data. A reply packet containing the result of the requested operation is generated and sent back over the SSL-connection. The connection is then closed and the requestor's contact information is relayed to the update-worker. The return value of these functions provided to the server module is always successful if the packet could be unpacked sucessfully - since the server only cares about if the received packet was indeed a kademlia request or not.

2.9.8 netdb

The netdb module implements the interface to the DHT as described in the Phantom paper. It can be seen as glue code between the main Phantom implementation and the DHT. Changing the DHT implementation therefore only requires changes to small parts of the server module and this module.

Listing 2.37: The netdb module's interface

```
1 int register_my_node_in_the_network(char *ip, uint8_t
      *communicationcertificate, uint8_t *path_building_certificate);
2 int extend_ap_address_lease(struct in6_addr *ap_address, uint8_t
      *signed_lease_request, uint8_t *routing_certificate);
3 int reserve_new_ap_address(const struct config *config, struct
      in6_addr *ap);
4 int update_routing_table_entry(const struct in6_addr *ap_address,
      struct rte *signed_routing_entry, uint16_t port, uint8_t
      *routing_certificate);
```

```
5 int get_random_node_ip_addresses(char **addresses, uint16_t *ports,
    X509 **communication_certificates, X509
    **path_building_certificates, int num);
6 int get_entry_nodes_for_ap_address(char ***ip_addresses, uint16_t
    **ports, int *num, const struct in6_addr *ap_address);
```

The first two functions are not implemented because the DHT lacks support for these operations. The `reserve_new_ap_address`-function creates an AP-reservation-path and one tunnel over it which is used to reserve an AP-address. The AP-address reservation itself is implemented in the main kademlia module and is only a stub returning a valid but random AP-address. `update_routing_table_entry` makes the reserved AP-address known to the DHT, `get_entry_nodes_for_ap_address` can then be used by other nodes to retrieve this information. The id for this piece of data is the SHA-1-hash of the AP-address in binary form. `get_random_node_ip_addresses` is used by the path module to get the required information of a number of other Phantom nodes required for path construction.

2.10 Integration/frontend

To test the prototype implementation, I have programmed a frontend for Linux machines. By using the Linux tun/tap interface, the integration was straightforward. The tun interface provides a virtual network interfaces at OSI layer three. It is therefore possible to send arbitrary IP-frames over the Phantom network which are then dispatched by the kernel to the right application. All higher level protocols using the IP-protocol should therefore be compatible with the prototype.

The frontend consists of a tun-device which has at least one reserved AP assigned as its IP and a corresponding route set. Creating the tun-device requires superuser-privileges. I have written a small shell script which creates a tun-device and changes the permissions on it so that the user running the Phantom protocol can communicate with the device. Furthermore adding and deleting IP (in this case AP) addresses and routes needs superuser privileges too. Since it does not seem like a good idea to have a prototype implementation running as a privileged user, I have instead written a small daemon which has to run as superuser and accepts requests from the main Phantom application to add or remove addresses from the tun-device and to set the routes on the tun-device. This technique is called privilege separation and makes it possible for the main Phantom application to run as a non privileged user.

2.10.1 phantomd

The phantomd module is used for privilege separation. It sets and removes IP (AP) addresses on the tun-device, accepting request from the main Phantom application. The communication interface between the two processes is a unix domain socket. The messages itself have been kept as simple as possible. There are two kinds of messages. One requesting to add a new AP-address to the tun device and one requesting the removal of a previously set AP-address. Phantomd will try to process the request after some basic validation and answer with a message signalizing success or failure of the requested operation to the requestor. The message format is shown in 2.5 and 2.6.

Figure 2.5: Request message format for phantomd

The request message is always exactly 17 bytes long. The first byte is either 'a' for adding an address or 'd' for removing an address. The remaining bytes are the 16 byte AP-address.

Figure 2.6: Reply message format for phantomd

A '1' signalizes failure - the operation was not performed, a '0' signalizes success.

2.10.2 addr

The addr module communicates with phantomd by sending the messages described in the previous subsection and receiving the reply messages. It exports two self explaining unary functions, each taking a previously reserved AP-address as its argument:

Listing 2.38: The addr module's interface

```
1 int set_addr(struct in6_addr *addr);
2 int del_addr(struct in6_addr *addr);
```

2.10.3 tun

The tun module is the most important part of the frontend. Given an exit or entry path, it creates tunnels over it as required and forwards traffic over them. The exported interface consists of two functions:

Listing 2.39: The tun module's interface

```
1 struct tun_dev *start_forwarding(struct path *path, const struct
    config *config);
2 void stop_forwarding(struct tun_dev *t);
```

After successfully constructing an entry or exit path it should be handed to the `start_forwarding`-function which will take care of the tunnels and communication running via this path as explained below. The `stop_forwarding`-function is used to shut down all tunnels and stop the communication along the path. In the current implementation (which is mainly for testing and simple demonstration) only one path should be created by each Phantom node. Since the tun module is implemented with the assumption that it is the only user of the tun-device. Later it would be desirable to have multiple paths. Communication to the tun-device will then have to be synchronized between the different path instances and their tunnels.

The `start_forwarding`-function starts one thread responsible for reading IPv6-frames from the tun-device. A blocking read from the tun-device will either fail or block and return one full IPv6-frame as soon as one is available. If a packet has been received from the tun-device, the thread extracts its destination address (the AP-address of another Phantom node). It maintains a list of previously created tunnels and sends the packet to the matching tunnel if one such tunnel exists. If no tunnel can be found and the underlying path is an entry path, the packet is thrown away. If the underlying path is an exit path it will try to create a new tunnel over it to the stated AP-address by calling the tunnel modules `create_tunnel`-function. If the tunnel was created successfully, the tunnel is entered into the list of running tunnels and the packet is dispatched into it. Since tunnels are bidirectional, for each tunnel created a reader-thread is spawned which reads IPv6-frames coming from the tunnel and writes the frames to the tun-device. Since multiple tunnels can exist and therefore also multiple reader-threads, these writes have to be synchronized. The format for transporting IPv6-frames through a tunnel is shown in 2.7.

If the path passed to the `start_forwarding`-function is an entry path, another thread is spawned in addition to the reader-thread. This threads waits for incoming tunnels via the tunnel module's `await_entry_tunnel`-function. If an entry-tunnel has arrived, it spawns a thread (running the same function as for exit-tunnels) to read packets from the tunnel and write them to the tun-device. Before doing so, the tunnel is entered into the tunnel list, so the reader thread for this path can find it there after reading the first reply frame logically belonging to the incoming connection from the tun-device and can successfully dispatch it back. The next incoming tunnel is awaited immediately after this.

The design using the tun-device has two main advantages. First, it works on the IP-layer which makes it possible to relay arbitrary IP-traffic through the Phantom network. This is basically all and every protocol used via the Internet.

Second, the packets are injected into the kernel's IP-stack by the tun-device and so the kernel or applications using the Phantom network handle everything related to higher protocol levels, the Phantom implementation itself does not have to care about it.

32 bit big endian frame-size	IP-frame

Figure 2.7: Format of an IP-frame send through a tunnel

2.10.4 main

This module is very simple and strictly for testing. The main function sanitizes the environment, blocks `SIG_PIPE`, reads the configuration file, brings up and initializes the other modules. It prepares the locks needed for libopenssl, initializes its PRNG and starts the Phantom server as well as the DHT. If everything comes up successfully, it creates an exit or entry path at random, prints the reserved AP-address for it and starts the `start_forwarder`-function of the tun module. This makes testing and demonstration of the implementation very simple using tools like cluster-ssh [9].

2.11 Things not implemented

2.11.1 Non anonymized participation

The Phantom protocol design makes it possible for members of the Phantom network to join without anonymization by being their own entry or exit node if they wish to do so. This functionality is not implemented. The prototype implementation requires every participant in the network to have a routing path of at least length one.

Implementing this functionality should however not pose a big problem.

2.11.2 DHT

As indicated in the previous section, the DHT is not fully implemented. What is there is basically enough to make the implementation testable, however a lot is still missing. First of all, a lot of thought has to be put into deciding which specific design would be most suitable to meet Phantom's requirements. It is not said at all that kademlia is the most suitable one of the existing designs or that it may be required to think of a whole new design.

Data in the DHT should be organized within tables, there is no such thing in the prototype yet. The only data stored within the DHT so far is the contact

[9] http://sourceforge.net/projects/clusterssh/

information for AP-addresses, i.e. the contact information of their entry nodes. The key for this data is the SHA-1 hash of the AP-address.

Since the DHT has to reserve AP-addresses for nodes, it needs to have some means of global synchronization as to not give out the same AP-address twice. Currently I simply select an AP-address at random and hope that there will not be conflicts. This showed to be sufficient for testing purposes.

In short, most of the DHT, better all, should be reimplemented and before doing so thought and discussed about a lot by good researchers with both good knowledge of DHT designs and a good understanding of the Phantom protocol. I found this to be out of scope of my diploma thesis.

2.12 Problems

2.12.1 Dummy package creation

The most important requirement of a dummy package used during routing path construction is not to be distinguishable from a real setup package destined for another node. A setup package is however not simply a lump of arbitrary data of arbitrary size but has certain properties that should also be resembled by a dummy package. Assuming that setup packages look from the outside like random data with a certain size, creating this data with a good determinstic PRNG is OK. The sizes of dummy packages however have to be chosen in a way to match those of the setup packages. Setup packages can have different sizes depending on their contents, not all sizes must be valid though. In my current implementation I choose the size of a dummy package as the size of another randomly selected setup package within the array. This is probably not optimal. Since the best solution of how to create dummy packages depends on the format of setup packages in the final protocol implementation it can not be decided upon yet, but is a topic to keep in mind.

2.12.2 Deallocation of tunnels

Since tunnels are created on demand and do not care about the protocol tunneled via them as long as it is IPv6, there is no way of knowing if a tunnel is no longer used and can be deallocated. At the moment I am therefore not deallocating created tunnels at all until they are either shut down explicitly or one of the underlying paths breaks down. For tunneled TCP connections it would be possible to inspect the packets for FIN-flags and shut down a tunnel once the TCP-connection has come to an end. This would however mean, that a tunnel is only ever used for one logical connection or even more information has to be tracked. This is also not possible for other protocols like UDP and therefore seems like the wrong way to go.

A possibility would be to timestamp tunnels every time a packet is received or sent through them and expire (deallocate) them after some timeout. If the connection was not really dead at this point, it would be possible to simply create a new tunnel if another packet belonging to this connection is received from the tun-device. This is at least possible if the underlying path is an exit path.

2.12.3 Missing AP-address for exit nodes

The original design does not stipulate AP-addresses for exit paths. The implementation of the forwarding via tun-devices however is a lot simpler if the exit node has also an AP-address from the Phantom AP-range. So I have decided to also give exit paths an AP-address.

2.13 Deviations from the original design

Due to the problems discussed in the previous section, I have chosen to deviate from the original design in two main points. First, I have changed and greatly simplified the way the setuparray is hashed and precalculated during routing path construction. Second, I have decided that not only entry nodes but also exit nodes should have an AP-address. This simplifies the tunneling of arbitrary IP-traffic via Phantom a great deal.

2.13.1 Verification and precalculation of setup arrays

The precalculation of the setup arrays is a crucial point to avoid piggybacking any information during path creation. If an attacker can piggyback any information during path creation, this could improve his chances to coordinate an attack to break or weaken the anonymity of the protocol.

This process is however very tedious, time consuming and I have failed twice when trying to implement it correctly. So I came up with a different solution. The hash over the setup array is not calculated across the array as a whole but across the individual slots within the array instead. These hashes are then xored and the result of this operation is the hash of the array.

This means the order in which the slots in the array are hashed is no longer important, which greatly simplifies the calculations. It however makes it possible to piggyback information of some kind. An attacker can rearrange the slots within the array freely and therefore transport information. He can for example hide information by ordering the slots in a way that the first bit in each slot concatenated gives information. This can even be improved. To weaken this possibility I require each node to randomly mix the slots in the array before sending it to the next node. So if there are valid nodes in between two attacker

nodes the information will be destroyed. This is still a lot weaker than the original design, and should be subject to further research.

2.14 Further improvements

I have thought of various further improvements to the implementation, however this project is only a diploma thesis and so I did not find time to implement all the cool stuff that came to mind. In addition there are some improvements that are obligatory if the prototype should be used in the wild.

The next subsections are roughly sorted by descending urgency and then by ease of implementation. They can be seen as possible starting points for anyone willing to improve the prototype implementation.

2.14.1 Getting rid of the ping thread

The ping thread (or ping worker) within the kademlia module is useless. The only one relaying ping-requests to it is the update worker. The update worker thread could just ping the nodes himself and therefore render the ping worker superfluous. It should be thrown out.

2.14.2 Getting rid of `cleanup_stack` macros

When implementing the prototype I found it tedious to have most lines of code I wrote followed by a check for some error condition and then a list of free or cleanup calls in the unlikely event that there was an error. So I tried to solve the problem with some macros. If some resource is allocated it can be pushed on a stack together with a free-function. If the resource is no longer needed it can be popped from the stack which causes the unary free-function to be called on the resource. In case of an error, there is a cleanup-all macro, deallocating all resources pushed on the stack within the current function. It made some code easier, but overall is not a good solution and I stopped using it after a while but have not yet found time to get it out of the code in every place. The two main problems with this approach were firstly that it allocates a large chunk of stack space for each function it is used in and that wastes memory, secondly, due to the implications of Rice's theorem[5] it is not possible to predict exactly how often a loop is executed. So if the push function is called within a loop, there may not be enough space available on the stack if it is called too often. Dynamically reallocating the needed space is not an option, since this reallocation can fail, too, and then one would have to handle this error again, not using these macros. The second reason makes this instrument a very bad choice for this implementation. To get rid of it should be easy but it takes some time to go through the code and change the deallocation of allocated resources in case of an error.

2.14.3 DPRNG used for dummy package creation

The deterministic random number generator used for the creation of dummy packages should probably be exchanged with another, better implementation. It has to be kept in mind though, that the main purpose of this PRNG is not to produce excellent random numbers but to produce a data stream reflecting the properties of a packed and encrypted setup package. Producing good random numbers should be good enough for this theoretically but, maybe not in practice.

2.14.4 Setup array precalculation

If my method presented in 2.13 should be found inferior to Magnus's originally prpoposed method, it should be changed to match the original method.

2.14.5 Exchange protobuf-c

I am pretty sure it is possible to piggyback additional information within libprotobuf messages without the library complaining about it. Since the piggybacking of information must be made impossible under all circumstances, libprotobuf-c is not the right choice to use here. I have chosen it anyway, because writing own serialization and deserialization routines is cumbersome and takes time, especially if the message format is changed from time to time during the early stages of implementation. Once the message format is relatively stable, own serialization and deserialization functions for the messages should be written along with very strict checking and validation functions. This is a lot of work, yet it is listed early on in this section, as failing to do so will make it possible to break Phantom's anonymity and that would just render the whole implementation useless in the first place.

2.14.6 Participation decision

The current implementation for deciding if a node wants to participate in a path construction process is just a dummy, always returning true. Arbitrary complex decision processes could be implemented here to decide if a node wants to participate. Normally the only valid factor should be if the node has enough ressources to support another path and a number of tunnels over it. Interesting factors for this would include usage of CPU, memory or bandwidth.

2.14.7 Selection of X- and Y-nodes

The possibility to select X- and Y-nodes for constructing a path is one of the great advantages of the Phantom design. The current implementation simply takes the first n nodes returned by the netdb module. There are a lot of possibilities to

strengthen Phantom's anonymity by selecting those nodes carefully. An example would be to select X-nodes in a way that they are most probably in different jurisdictions or countries using GeoIP-services. Another way to select the nodes could be by setting up a web of trust similar to the one used by GnuPG[10], where nodes are preferred if they have a high trust ratio, provide high bandwidth or have a high uptime and provide stable service. A blacklist could be implemented where the user can blacklist nodes if he finds them supsicious for some individual reason. Many more schemes come to mind and implementing some of them or combinations of them is an interesting option to put the possibilities provided by Magnus' design to good use.

2.14.8 IPv6 support

Currently the implementation uses IPv6 addresses for AP-addresses and only IPv4-addresses for normal communication. Implementing IPv6 support for normal communication would be a nice feature. IPv6 is coming and applications should be ready for it, the sooner the better. Since Phantom cannot work easily through network address translation and NAT is deprecated when using IPv6 Phantom would benefit greatly if IPv6 would be more widespread.

2.14.9 Getting rid of `system` in phantomd

The interface provided by Linux to add or delete routes or addresses on a network interface is called netlink. The interface is horribly complex and confusing, at least if one has no experience using it. I had none, so after trying for three days to get the interface to do what I wanted, I had a look at other opensource projects to see how they handle this stuff. The interface does not seem to enjoy widespread use and most other projects use the libc function `system` to call on the `ip` utility. I decided to do the same. This is however not good style and `system` is known for causing problems, so this should be changed, preferably by someone comfortable with the netlink interface.

2.14.10 Better use of the thread_pool-module

I have implemented the thread pool module once I was halfway done with the implementation. Therefore it is used only within the server module. The advantages of having a thread pool discussed in 2.5.4 could also be used in other modules. So modules using threads should be checked to see if they can make use of the thread pool implementation and if so, be changed to use it.

[10] http://www.gnupg.org

2.14.11 Poll on SSL-sockets

Using `poll` on SSL-sockets poses a problem. The `poll` system-call checks only if there is data available from the kernel for the userland, it has no way of knowing if this data is meant for libopenssl or the application using the library. So if the application tries to see if there is data available on the socket, `poll` will not block and tell that there is some data available, even so this data is only data concerning libopenssl. The application will then fail to get the data and maybe use `poll` again, with the same implications. This causes a `poll`-storm, similar to busy waiting. To get around this problem, I am using blocking SSL-read and write functions instead of polling the socket directly. This however further increased the number of threads necessary to run Phantom, which seems unnecessary.

If there is a way to use `poll` together with libopenssl it should be found, implemented and put to use.

2.14.12 Dynamic module support

For various things discussed in this section, some kind of loadable module support would be great. For example, a user could decide in which way he wants to select X and Y nodes by choosing from a set of community provided modules implementing these functions. Different user groups will have different needs and adapting to this needs can be made easier if potential user groups could implement these needs themselves, encapsulated in some kind of loadable module programmed against a clean and easy to use interface. This would make it easier on users to match Phantom to their needs without having to put all that code inside the Phantom core application, making it more complex. I would suggest using the dynamic linking loader's `dlopen` interface for this. Module functionality could be implemented as a set of functions with a fixed signature and then provided as a dynamic library file.

2.14.13 Different ciphers, hashes and DPRNGs

Some people may not be comfortable with my choice of ciphers, hashes and the DPRNG used to create dummy packages. For them it would be nice if the ciphers they wish to use for their traffic going through Phantom or for constructing their routing paths could be changed. This would make it necessary to contain information about the ciphers used within the setup packages. If the setup packages themselves should be encrypted with different algorithms it might be possible to implement a three-round path construction process, Where the first round sends a default package with the flags signaling which ciphers and hashes to use for the setup process. The second round would then become the normal first round (with other ciphers) and the third round the normal second round. It would also be possible that not all nodes support all kinds of hashes or ciphers, the contact

information within the DHT could then be updated to contain a list of cryptographic routines supported on this node. For the onion encryption it would even be feasible to have different ciphers on different X-nodes.

Implementing this functionality in this or some other way would also make it possible to easily update the Phantom implementation to use new cryptographic routines should the old ones be broken or considered insecure in the future.

2.14.14 Overall stability

The overall stability of the implementation has to be improved. This means there should be careful reviews of the code for bugs and correctness issues. There should be a number of rigorous testcases to protect from unforeseen consequences when changing some part of the code. This prototpye is nowhere near being ready for production use and for a protocol like Phantom that some people may be tempted to trust with their lives making it ready can not be accomplished by a single person, certainly not during a diploma thesis. Having said this clearly, I hope a community, interested in my work and the Phantom protocol will form and help to shape and expand the prototype into a usable protocol implementation.

Chapter 3

Evaluation

For Evaluation purposes I did some measurements on two sets of machines A and B. The machine's specifications are listed in 3.1. All test machines were connected via a 100Mbit switched network. Measured values in this chapter are in seconds.

	Type A	Type B
CPU	AMD Opteron 148 @ 2.2 GHz	Intel Core 2 Quad Q6600 @ 2.40 GHz
RAM	1024 MiB	8192 MiB
NIC	Nvidia CK804 Gigabit Ethernet	Intel 82566DM-2 Gigabit Ethernet

Figure 3.1: Technical data for test machines

3.1 Path creation

The timings for the path creation process were taken as the time of a full successful run if the `construct_entry_path` or `construct_exit_path`-function. Since these functions also create a second path and a single tunnel over it to reserve an AP-address, this is actually the timing for two path creation processes and one tunnel creation process. So the time to create a single path should be roughly half or slightly less than the times measured in 3.2, 3.3, 3.4 and 3.5. The path creation processes all used three X-nodes and twelve Y-nodes.

The mean, variance, standard deviation, minimum and maximum of the measurements are l isted in 3.6.

Path creation takes roughly between two and seven seconds using todays machines. This is however on a local 100 Mbit network. Via the Internet times will probably be higher due to bandwidth limitations and greater latency between nodes. The reason for the differences in times is probably how long it took the X- and Y-nodes to find their individual setup packages within the arrays and how many dummy packages they had to create. As expected the type B machines

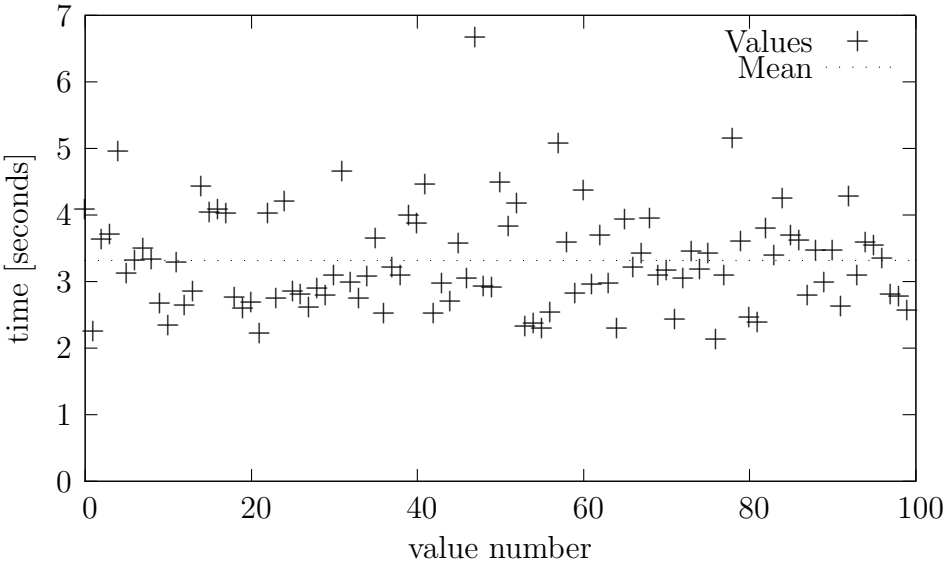

Figure 3.2: Times for `construct_exit_path` on type A machines

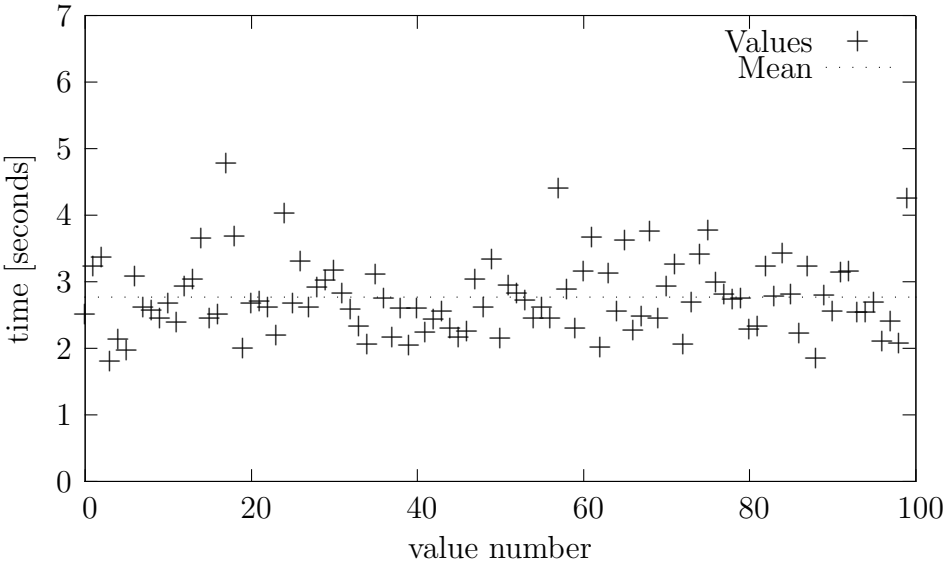

Figure 3.3: Times for `construct_exit_path` on type B machines

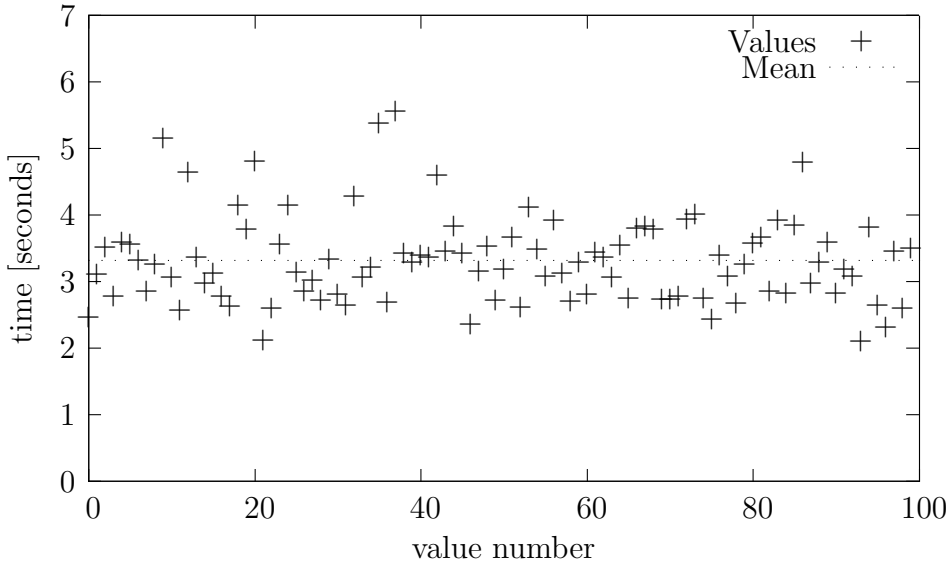

Figure 3.4: Times for `construct_entry_path` on type A machines

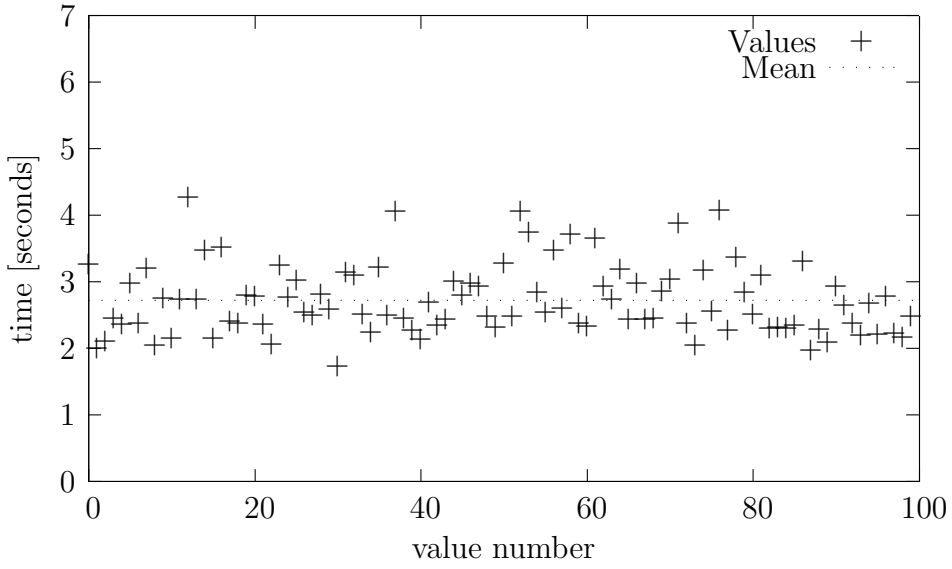

Figure 3.5: Times for `construct_entry_path` on type B machines

	3.2	3.3	3.4	3.5
Mean	3.318674	2.769496	3.317035	2.724664
Variance	0.582186	0.316752	0.443819	0.279035
Standard deviation	0.763011	0.562807	0.666197	0.528237
Minimum	2.132818	1.811392	2.108837	1.736680
Maximum	6.669100	4.776687	5.554295	4.271500

Figure 3.6: Further evaluation of the path creation measurements

can profit from their better specifications. This could be further exploited by parallelizing dummy package creation and finding the setup package inside the setup array.

3.2 Tunnel creation

The main time for tunnel creation is spent by the owner node trying to brute force the combination of xkeys used along a newly created tunnel. The formula for the maximum number of key combinations possible given the number of X-nodes N and the number of xkeys for each node x is: x^N. Since the keys are chosen at random, one can expect to find the right combination in half the time needed to try all the combinations. I have measured the times for 100 tunnel creation brute force attempts on paths using three or four X-nodes with 15 or 30 xkeys each. The results can be seen in 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13 and 3.14. The number of combinations, mean, maximum and minimum values are listed in 3.15.

If we assume 100 measured values are enough to calculate a good mean, we could estimate that the time used to bruteforce one combination is roughly 2.04 us on type A machines and 1.5 us on type B machines. The system clock's granularity kept me from getting usable values from direct measurements.

These times could be improved by writing specialized bruteforcing functions using for example SSE-instructions to test multiple key-combinations at once or by exploiting thread-level parallelism. The search space can be divided trivially and so a big number of CPUs could be used to do the brute forcing. The expected speed-up would be roughly the number of CPUs used minus the time used to divide the searchspace and start/join the threads. As expected, the type B machines outperformed the type A machines during these tests, even so the prototype's bruteforcing implementation is not parallelized.

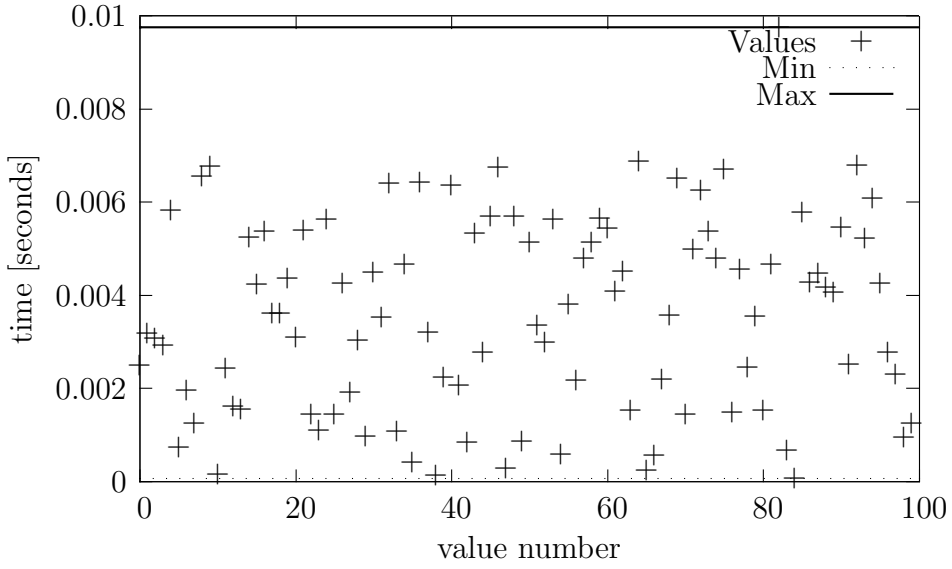

Figure 3.7: Brute forcing 15 xkeys on a path of 3 X-nodes on type A machines

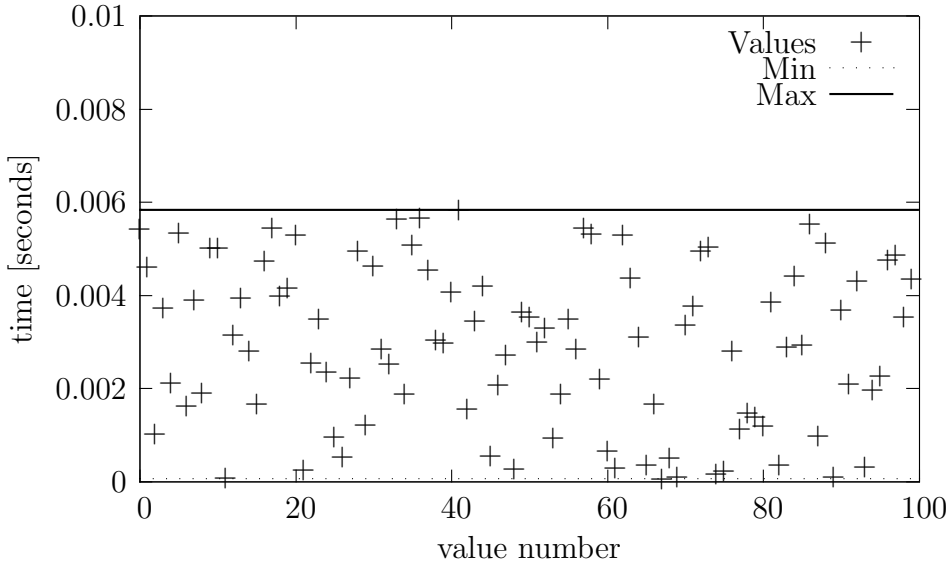

Figure 3.8: Brute forcing 15 xkeys on a path of 3 X-nodes on type B machines

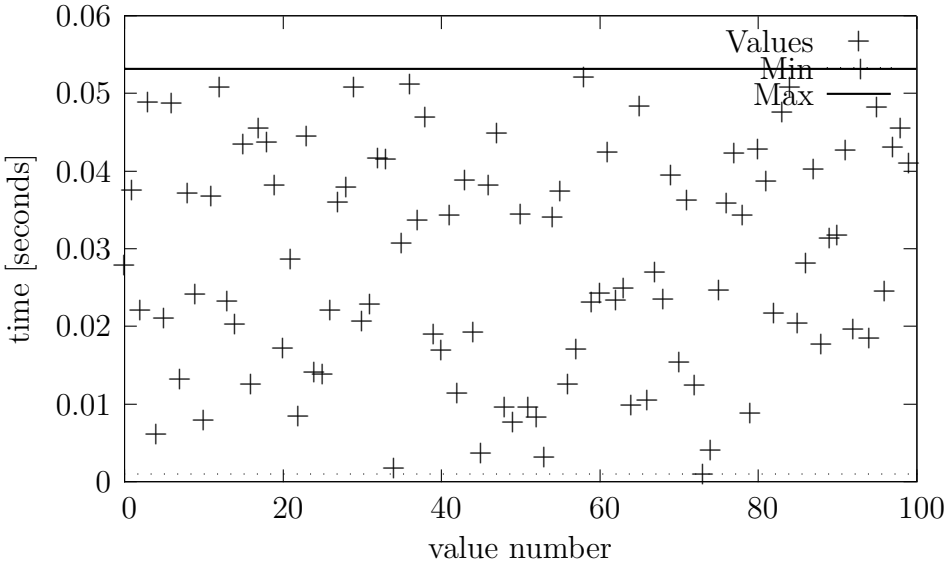

Figure 3.9: Brute forcing 30 xkeys on a path of 3 X-nodes on type A machines

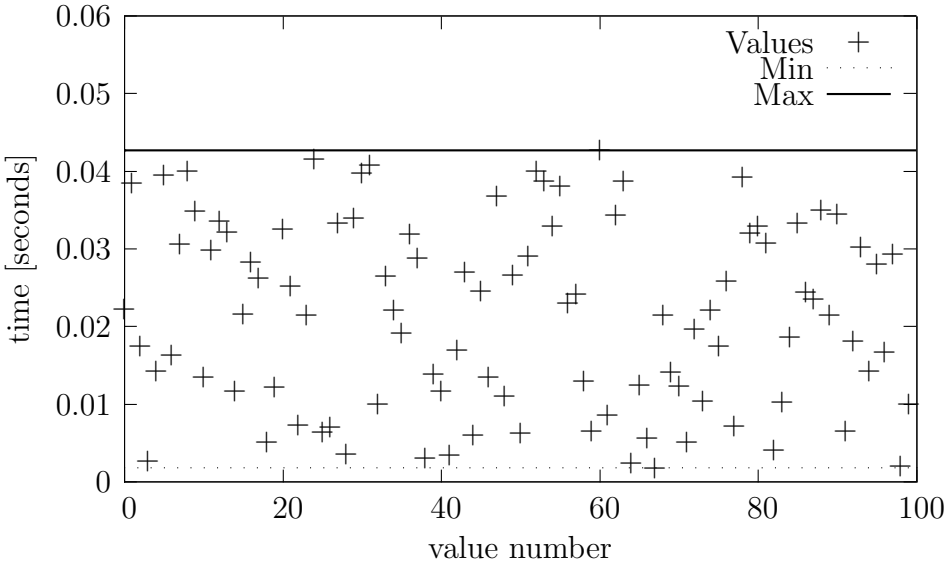

Figure 3.10: Brute forcing 30 xkeys on a path of 3 X-nodes on type B machines

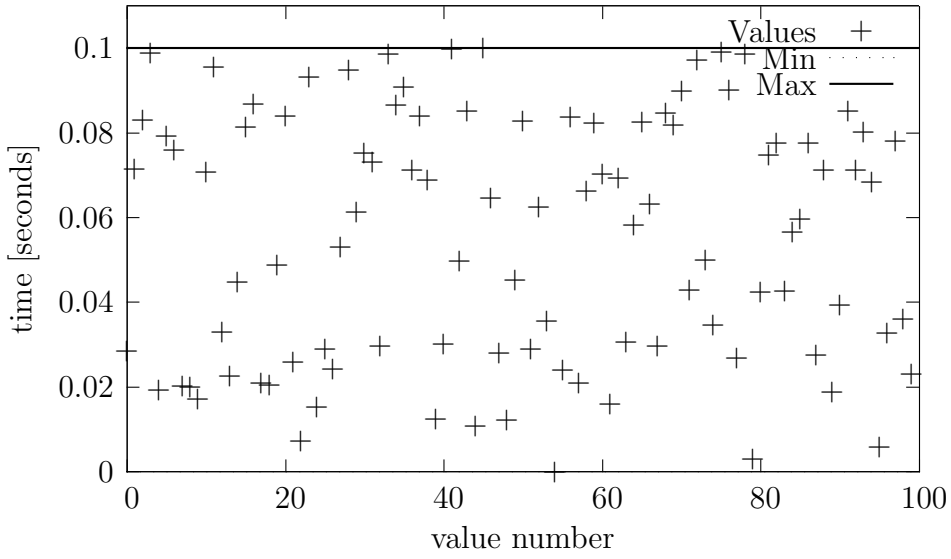

Figure 3.11: Brute forcing 15 xkeys on a path of 4 X-nodes on type A machines

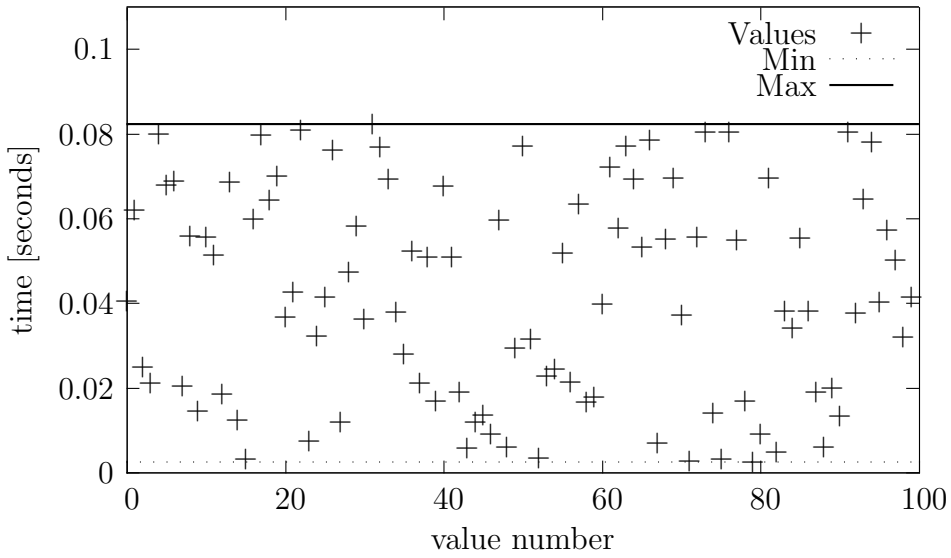

Figure 3.12: Brute forcing 15 xkeys on a path of 4 X-nodes on type B machines

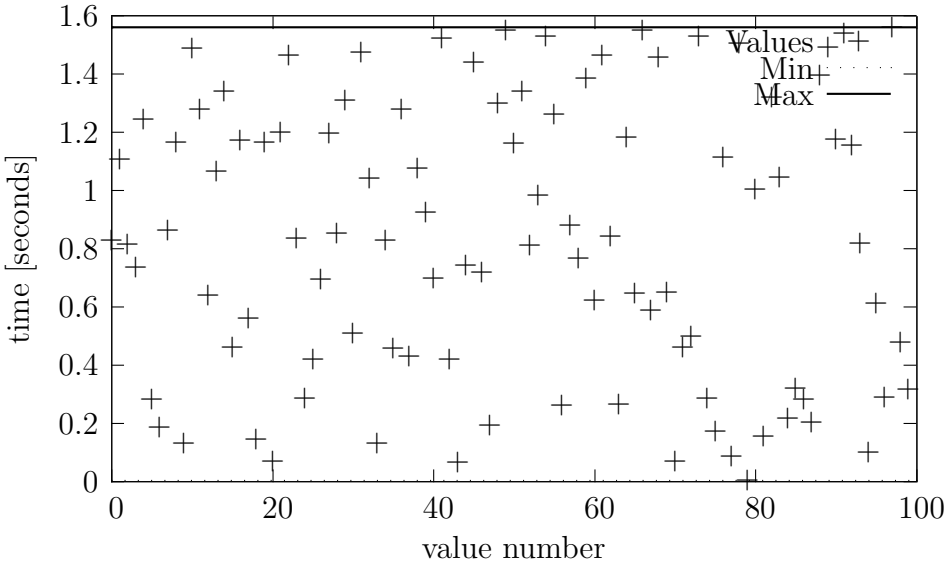

Figure 3.13: Brute forcing 30 xkeys on a path of 4 X-nodes on type A machines

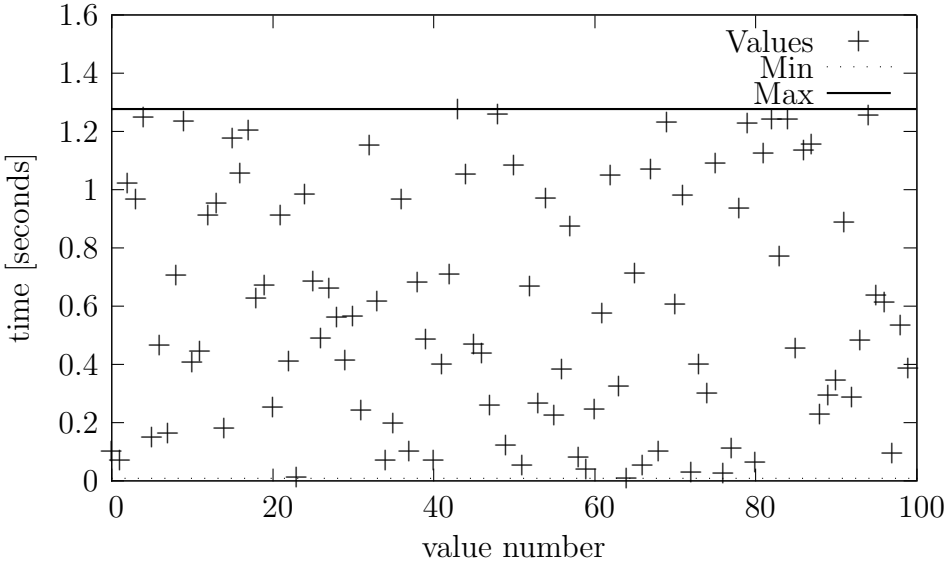

Figure 3.14: Brute forcing 30 xkeys on a path of 4 X-nodes on type B machines

	3.7	3.8	3.9	3.10	3.11	3.12	3.13	3.14
Combinations	3375	3375	27000	27000	50625	50625	810000	810000
Mean	0.003589	0.002932	0.028479	0.021476	0.055166	0.041735	0.827930	0.593491
Minimum	0.000064	0.000068	0.001012	0.001775	0.000062	0.002628	0.003547	0.009626
Maximum	0.009748	0.005832	0.053188	0.042704	0.100077	0.082332	1.559712	1.276336

Figure 3.15: Further evaluation of brute force measurements

3.3 Throughput

Finally I measured the throughput via a precreated tunnel. For these measurements an exit and an entry path, consisting of three X-nodes each, have been used. So the total path length between the two owner nodes was six. I did three sets of measurements, the round trip time for 1024 bytes of data (3.16, 3.17), the round trip time for 8192 bytes of data (3.18, 3.19) and finally the time to transmit 100 MiB of data from the exit path owner to the entry path owner(3.20, 3.21). The mean, variance, standard deviation, minimum and maximum values for these measurements can be seen in 3.22.

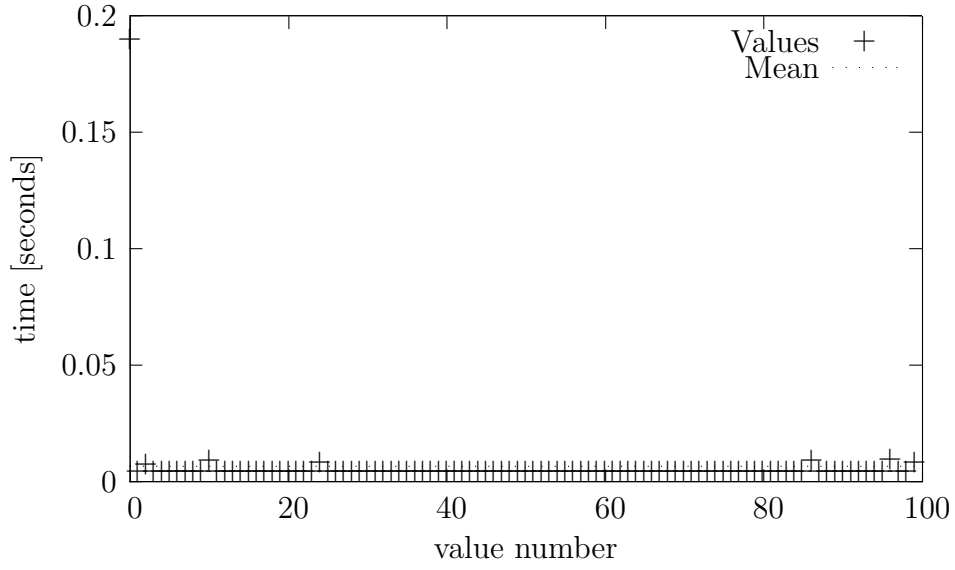

Figure 3.16: 1 KiB round trip time on type A machines

Two interesting observations can be made from the round trip time measurements. First the weaker type A machines performed a little better than the type B machines. Second the first measured value is a lot larger than the other values who are very close to the mean. This is probably because libopenssl is doing some initialization for the cipher contexts on each node when the first data arrives over the tunnel.

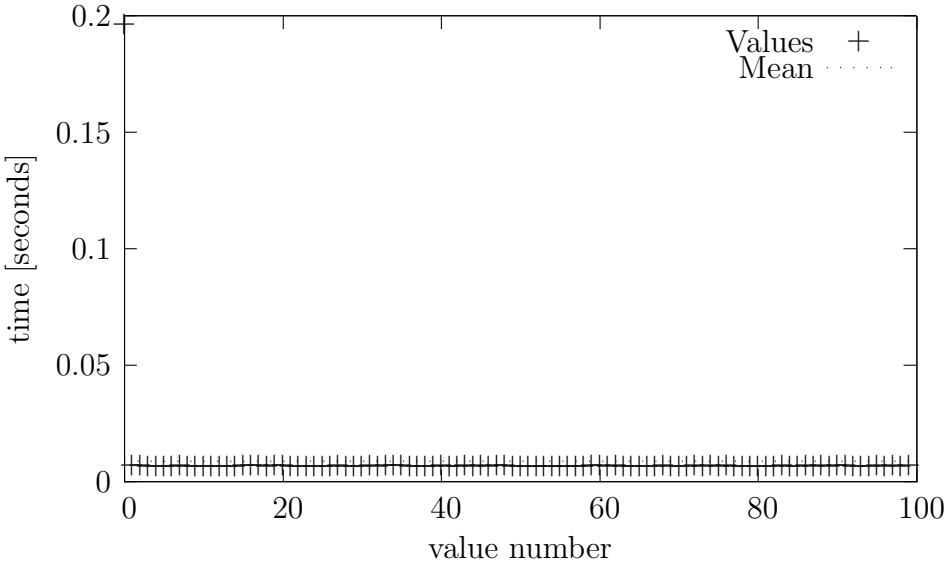

Figure 3.17: 1 KiB round trip time on type B machines

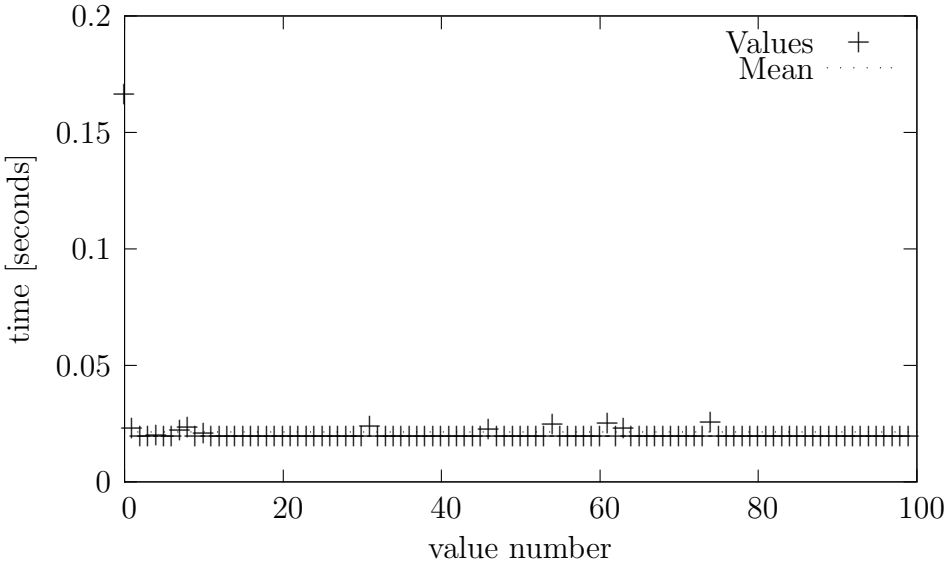

Figure 3.18: 8 KiB round trip time on type A machines

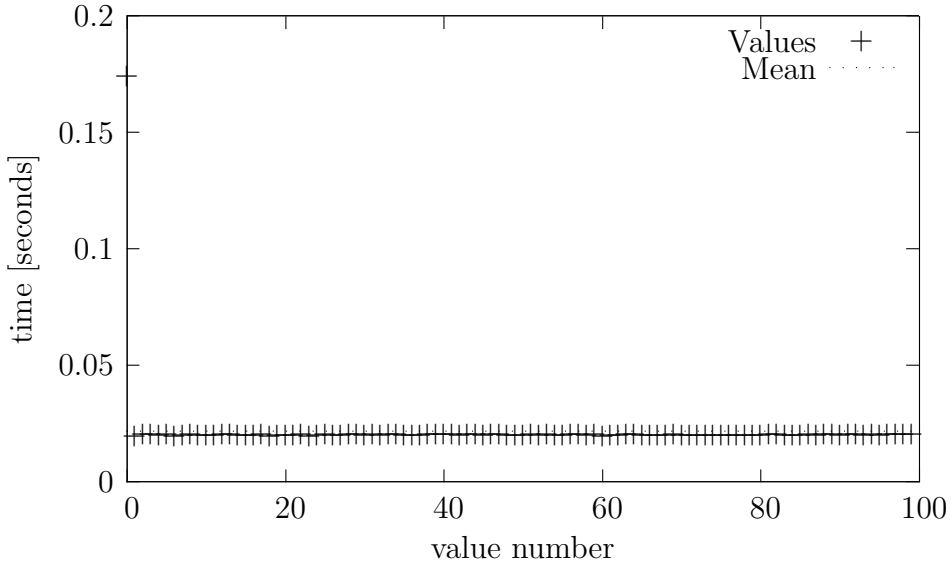

Figure 3.19: 8 KiB round trip time on type B machines

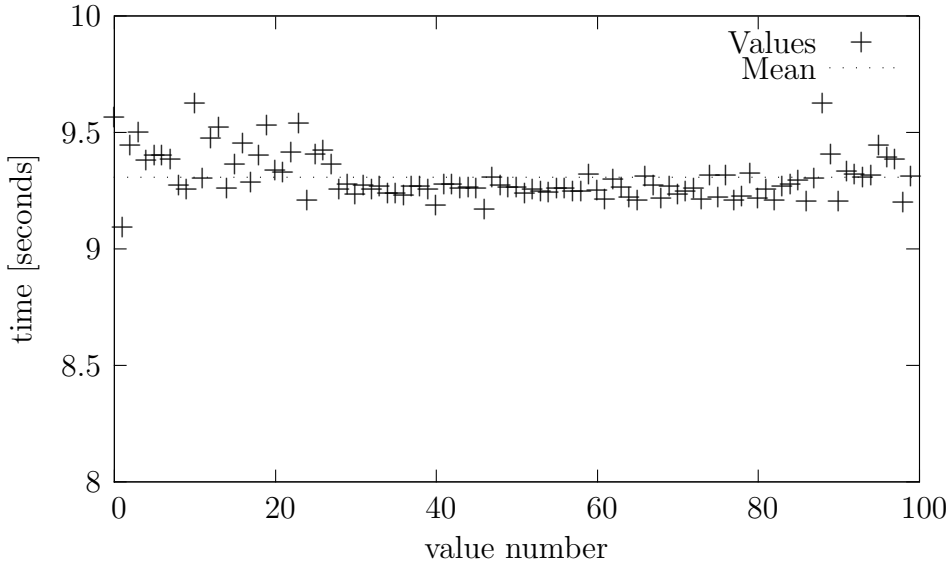

Figure 3.20: 100 MiB transmit time on type A machines

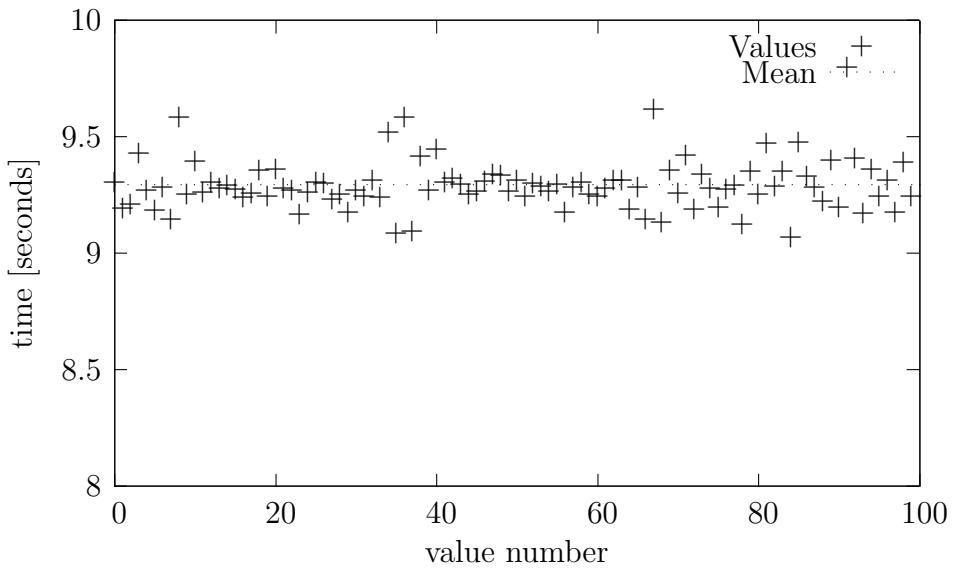

Figure 3.21: 100 MiB transmit time on type B machines

While transmitting 100 MiB from the exit path owner node to the entry path owner the theoretical throughput limit of the underlying 100 Mbit network seemed to be the limiting factor on both machine types. The theoretical throughput limit for a 100 Mbit network is 12.5 MiB/s. This includes however overhead like ethernet, TCP and IP headers which sum up to 110 bytes per ethernet frame transmitted in the current setup. In the best case each ethernet frame transmitted carries a measured payload of 1426 bytes of data and has a size of 1526 bytes. On a totally free and perfect network I would expect to see 8589 ethernet frames per second carrying a combined payload of roughly 11.68 MiB/s. In practice this value is even further reduced by interframe gaps, which require the ethernet device to be silent for 96 bit times after each ethernet frame. Deducting the interframe gaps leaves a payload of 11.58 MiB/s. Further deductions have to be made for CSMA/CD after collisions, TCP flow control and TCP congestion avoidance. Additionally processing the data in each node through onion encryption and decryption causes delays. The measurements show values of 10.74 MiB/s for the type A machines and 10.75 MiB/s for the type B machines. Both types of machines were not at their CPU-capacity limit by far which shows that the prototype implementation can achieve very high bandwith saturation.

	3.16	3.17	3.18	3.19	3.20	3.21
Mean	0.006468	0.008886	0.021360	0.021696	9.306352	9.293772
Variance	0.000346	0.000359	0.000216	0.000238	0.009677	0.012260
Standard deviation	0.018591	0.018945	0.014703	0.015415	0.098370	0.110723
Minimum	0.004321	0.006856	0.019419	0.019801	9.093342	9.067965
Maximum	0.190223	0.196437	0.166353	0.174300	9.626617	9.798135

Figure 3.22: Further evaluation of throughput measurements

Chapter 4

Outlook

I have shown that the Phantom protocol design is implementable. I have also encountered and discussed problems with the design and my implementation and proposed further improvements. I have pointed out where more research is needed. The prototype is nowhere near being ready for productive use, for various reasons discussed in this thesis, but it has come one step further through my work.

Magnus and I are currently looking at good licensing options for the prototype implementation and will release it under a license we see fit soon. Hopefully a community interested in the Phantom protocol will form and find my work helpful in bringing Phantom to the real world.

The prototype implementation will probably not be the basis for the final implementation of Phantom, but I hope it will help in making Phantom the new de facto anonymization standard used worldwide.

List of Figures

Listings

Bibliography

[1] *Secure Hash Standard*. National Institute of Standards and Technology, 2002. Federal Information Processing Standard 180-2.

[2] Magnus Bråding. The Phantom Protocol. In *Proc. DEFCON 16*, 2008.

[3] Daniel Stutzbach and Reza Rejaie. Improving Lookup Performance over a Widely-Deployed DHT, 2006.

[4] Douceur, John R. The Sybil Attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer-Verlag, 2002.

[5] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[6] R. Hinden and B. Haberman. Unique Local IPv6 Unicast Addresses. RFC 4193 (Proposed Standard), October 2005.

[7] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, 2002.

[8] Kim Zetter. Rogue Nodes Turn Tor Anonymizer Into Eavesdropper's Paradise. *Wired*, 2007.

[9] Lenore Blum, Manuel Blum and Michael Shub. A simple unpredictable pseudo random number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.

[10] Steven J. Murdoch and George Danezis. Low-Cost Traffic Analysis of Tor. *Security and Privacy, IEEE Symposium on*, pages 183–195, 2005.

[11] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer-Verlag, 2002.

[12] Roger Dingledine, Nick Mathewson and Paul Syverson. Tor: The Second-Generation Onion Router. In *Proc. 13th USENIX Security Symposium*, 2004.