

Diploma Thesis

Building The Phantom Protocol Routing Path

Michael Prinzinger

October 20, 2009



Chair for Distributed Systems and Operating Systems

**Friedrich-Alexander-Universität
Erlangen-Nürnberg**



Friedrich-Alexander University of Erlangen-Nürnberg

I hereby assure to have made this project without the aid of others, only using the sources I have referenced in section 12. Furthermore I assure that this project has never been submitted in this or similar form to any other examination authority, nor been accepted as such in an academic record. Every statements, which have been directly taken from references, are indicated as such.

Erlangen, September the 14th 2009

[Place, Date]

[Michael Prinzing]

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 5 |
| 2 | Protocol Design | 6 |
| 2.1 | Routing Path | 7 |
| 2.1.1 | First Round | 7 |
| 2.1.2 | Second Round | 10 |
| 2.2 | Routing Tunnels | 11 |
| 2.2.1 | AP-Address Resolution | 11 |
| 2.2.2 | Secure Establishment of Routing Tunnels | 12 |
| 2.2.3 | Secure Connections over Routing Tunnels | 16 |
| 2.3 | Phantom Network Database | 17 |
| 2.3.1 | Distributed Hash Table | 17 |
| 2.3.2 | Phantom Network Database | 17 |
| 2.3.3 | Phantom Network Database API | 17 |
| 3 | Implementation | 18 |
| 3.1 | Libraries | 18 |
| 3.1.1 | OpenSSL | 18 |
| 3.1.2 | Google Protobuf | 18 |
| 3.1.3 | Google Test Primer | 18 |
| 3.1.4 | GLog (GoogleLog) | 18 |
| 3.1.5 | YAML-cpp | 18 |
| 3.1.6 | GC (Garbage Collector) | 19 |
| 3.2 | Overview | 20 |
| 3.2.1 | Routing Path Sequence Diagramm | 20 |
| 3.2.2 | Inheritance Diagram | 21 |
| 3.2.3 | Composite Diagram | 21 |
| 3.2.4 | Phantom Routing Node Protoype | 22 |
| 3.2.5 | Phantom Anonymized Node Prototype | 23 |
| 3.3 | Environment | 24 |
| 3.3.1 | Logging | 24 |
| 3.3.2 | Config File | 24 |
| 3.3.3 | Serialization | 25 |
| 3.3.4 | Unit Testing | 26 |
| 3.3.5 | Garbage Collection | 27 |
| 3.4 | Components | 27 |
| 3.4.1 | Encryption / Decryption | 27 |
| 3.4.2 | Signature / Signature Verification | 33 |
| 3.4.3 | Hash Codes | 34 |
| 3.4.4 | Sockets | 35 |
| 3.4.5 | SSL Streams | 37 |
| 3.5 | Prototype | 40 |
| 3.5.1 | Phantom Network Database | 40 |
| 3.5.2 | Routing Tunnels | 40 |
| 3.5.3 | Routing Path | 41 |
| 4 | Manual | 56 |
| 4.1 | Installing Phantom | 56 |
| 4.2 | Folder Structure | 56 |
| 4.3 | Binaries | 56 |
| 4.3.1 | Anonymized Node | 56 |
| 4.3.2 | Routing Node | 56 |
| 4.4 | Test Scenarios | 57 |
| 4.4.1 | Local Test | 57 |
| 4.4.2 | Remote Test | 57 |

| | | |
|-----------|---|-----------|
| 4.4.3 | Unit Test | 58 |
| 5 | Testing the Prototype | 59 |
| 5.1 | Run Time | 59 |
| 5.1.1 | Remote Test | 59 |
| 5.1.2 | Local Test | 59 |
| 5.2 | What doesn't work yet | 59 |
| 6 | Analysis of Strenghts and Weaknesses of the Phantom Protocol | 60 |
| 6.1 | Design Goals | 60 |
| 6.2 | Weaknesses | 62 |
| 6.3 | Strengths | 63 |
| 7 | Comparison of Phantom with other anonymization approaches | 64 |
| 7.1 | TOR | 64 |
| 7.1.1 | Introduction | 64 |
| 7.1.2 | Comparision | 66 |
| 7.1.3 | Attack Scenarios | 67 |
| 7.2 | JAP | 69 |
| 7.2.1 | Introduction | 69 |
| 7.2.2 | Comparison | 70 |
| 7.2.3 | Attack Scenarios | 71 |
| 7.3 | I2P | 72 |
| 7.3.1 | Introduction | 73 |
| 7.3.2 | Comparison | 75 |
| 7.3.3 | Attacks | 76 |
| 7.4 | Synthesis | 77 |
| 8 | Outlook | 78 |
| 8.1 | Implemenatation | 78 |
| 8.2 | Further Ideas | 79 |
| 9 | Socio-Poltical Aspects | 80 |
| 10 | Juristical Aspects | 80 |
| 11 | Glossary | 81 |
| 12 | References | 81 |
| 12.1 | Libraries | 81 |
| 12.2 | Lecture Notes | 82 |
| 12.3 | Books | 82 |
| 12.4 | Artikel | 82 |
| 12.5 | Papers | 82 |
| 12.6 | Academic Thesis | 83 |
| 12.7 | Internet | 83 |
| 12.8 | Images | 84 |
| 12.9 | Standards | 84 |

1 Introduction

The Phantom Protocol

This thesis deals with the Phantom Anonymity Protocol presented by Magnus Bråding on DEFCON 16 in Las Vegas 2008.

The Phantom Protocol is an internet protocol to set up anonymous and encrypted connections over IP-networks. As the name suggests the protocol's main purpose is to provide anonymity, both to the one requesting a resource, and to the one providing a resource. Also by making heavy use of encryption it protects the content of what is requested and sent from being read, except to the requester and resource provider themselves. An important difference to established solutions to providing anonymity is that the phantom protocol is targeting the general population, and thus using the protocol should require almost no technical knowledge.

Design Goals

The main design goals of the protocol are:

1. Complete decentralization.
2. Maximum resistance against all kinds of DoS attacks.
3. Theoretically secure anonymization.
4. Theoretically secure end-to-end transport encryption.
5. Complete (virtual) isolation from the "normal" Internet.
6. Maximum protection against identification of protocol usage through traffic analysis.
7. Capability of handling larger data volumes, with acceptable throughput.
8. Generic and well-abstracted design, compatible with all new and existing network enabled software.

This Draft

This draft consists of three parts.

1. [Practical Part] The first part deals with designing the protocol to incorporate the concepts and procedures presented in the white paper???. After that we will try to actually implement a part of the protocol: the setting up of the routing part.
2. [Theoretical Part] The second part deals with analyzing the protocol regarding the design goals stated above. Furthermore the concepts of the protocol are compared to alternative solutions like the TOR network.
3. [Socio-Political Part] The third part deals with the socio-political importance of anonymization in our current society. It tries to discover the chances and risks of introducing internet anonymity to the citizens of a democracy.

[I. Practical Part]

2 Protocol Design

The design of the protocol is taken from the white paper “*Generic, Decentralized, Unstoppable Anonymity: The Phantom Protocol*” (see ??).

The protocol can be roughly divided into three parts:

1. Building the Routing Path (2.1)
2. Building Routing Tunnels between Routing Paths (2.2)
3. The Phantom Network Database (2.3)

The following figure illustrates the structure, once both sides have established their routing paths and a routing tunnel was made to connect them.

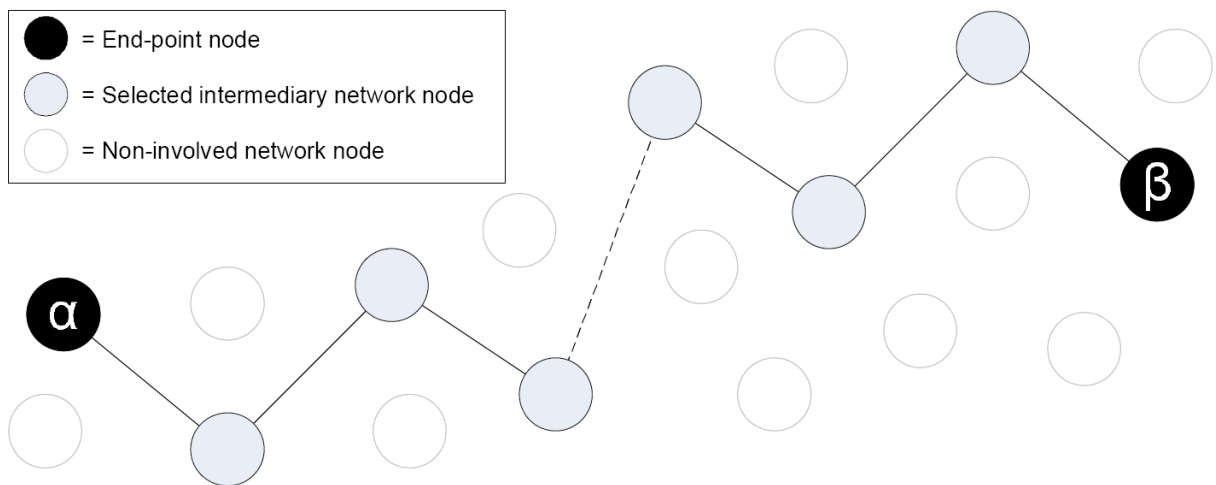


Figure 1: Phantom Protocol Overview

Here α stands for an anonymized node requesting some resources from an also anonymized node β providing these resources. To get the resources α establishes a routing path, selecting a couple of nodes from the **Phantom Network Database** and then builds up the **Routing Path** shown in the figure above. The last node of this Routing Path is called **EXIT-node** and will resolve connection requests from α to an **AP-Address** (Anonymous Protocol Address). β does exactly the same, however once β 's Routing Path was established the last node, here called **ENTRY-node**, registers the newly build path with an AP-Address plus port, Certificate, Public Key and its own IP-Address in the Phantom Network Database. Then the ENTRY-node opens a socket on the registered port and listens for incoming connection requests. α 's EXIT-node is now able, at the request of α , to resolve an AP-Address in the Phantom Network Database getting the IP-Address of β 's ENTRY-node, together with certificate to verify β 's identity and a public key to encrypt messages to β with.

Now once β 's ENTRY-node receives the connection request from α 's EXIT-node, on both sides separate connections between terminal node (EXIT/ENTRY) and anonymized nodes (α/β) is established in two rounds symmetrical for both paths, where stream encryption keys are chosen to be used only for this individual connection. After both sides finished these two rounds a real connection is established and data can be exchanged in a secure and anonymous way. This is called a **Routing Tunnel**.

In the following sections we will look at these parts in detail and explain its significance.

- a number of Y-nodes equal to the total number of X-nodes minus one (although always at least one), are located adjacent to each other in the other end of the sequence.
- one end of the sequence is chosen at random to be the beginning of the sequence.

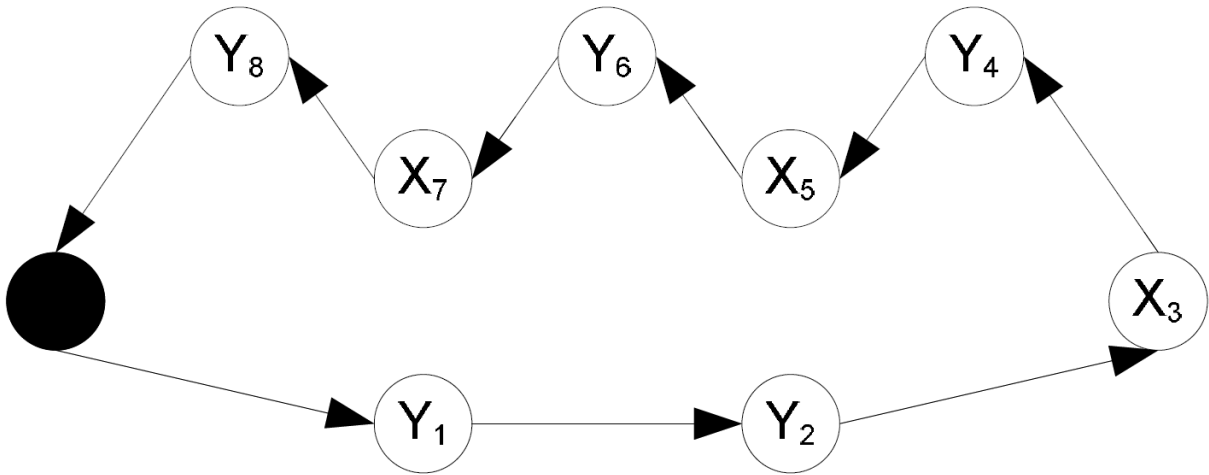


Figure 4: routing path in the first round

The anonymized node then generates a public/private key pair exclusively for the routing path establishment (routing path construction certificate and construction key) and prepares a special unique **Setup Package** for every selected node. This Setup Package contains the following information.

First Round Setup Package

- routing path construction certificate
- secure cryptographic hash of the entire (encrypted) setup package array
- secure cryptographic hash of the decrypted setup package
- dummy package creation information (128-bit seed, size, index and flags)
- 128-bit seeds for stream encryption key generation and the number of keys to be generated.
- node type flag

previous node

- previous node IP address
- previous node 128-bit connection ID (random)
- previous node communication certificate (SSL stream)

next node

- next node IP address
- next node 128-bit connection ID (random)
- next node communication certificate (SSL stream)

These setup packages are then encrypted first asymmetrically with the designated routing node’s path building certificate and then symmetrically with the 128bit ID of its incoming connection and finally signed with the construction key. Then these packages are placed in an array in a randomly order and sent to the first routing node of the previously established sequence of nodes.

| node type | description |
|-----------|--|
| α | the anonymized node establishing a routing path for requesting resources |
| β | the (anonymized) node (establishing a routing path for providing) resources ¹ |
| X | an intermediate routing node of the routing path |
| Y | an temporary node for the routing path construction |
| EXIT | a special exit routing node resolving the receiver’s AP-Address |
| ENTRY | a special entry routing node resolving connection requests to β ’s AP Address |

Table 1: Node Types

Routing Node

Each routing node then iterates through the array looking for the one setup package, it can decrypt using first the connection ID and secondly its own private key. Once it successfully decrypted one of the setup packages, every routing node

- authenticates the previous node by matching both the IP-address given in the setup package with the actual IP address of the connected node, as well as the connection IDs.
- verifies data integrity by checking the hash of the decrypted setup package, as well as the hash of the complete encrypted array.
- verifies the signature of the setup package with the construction certificate found in the setup package
- reads and stores all the remaining contents of the setup package: node type, information about previous and next node, etc.
- replaces the setup package in the array with a dummy package created with the seeds and flags found in the setup package.
- establishes a connection to the next node
- verifies the next nodes SSL communication certificate
- sends on the modified setup package together with the connection ID

The first round ends with the Y-node at the opposing end of the sequence sending the array ‘back’ to the anonymized node.

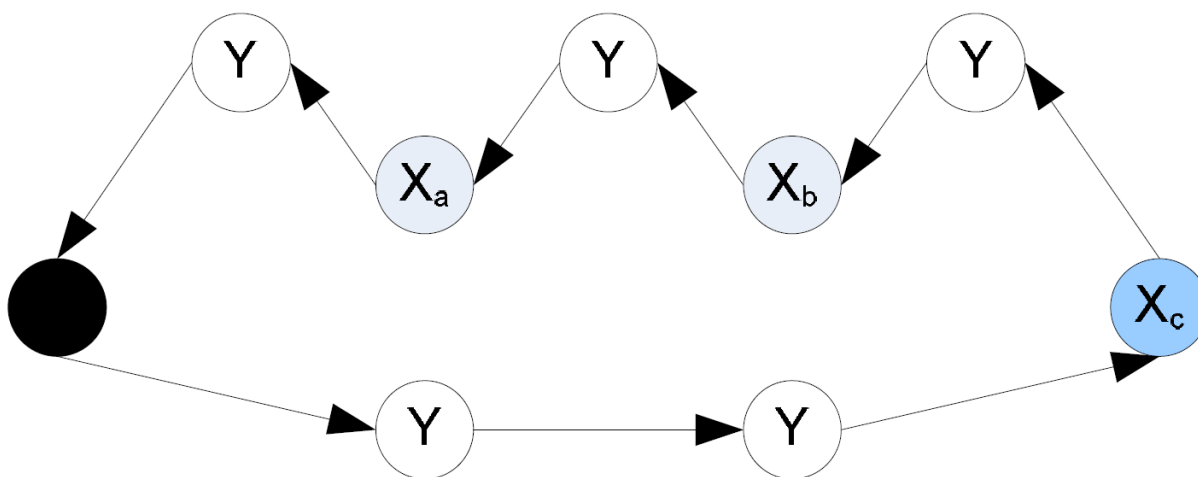


Figure 5: routing path creation, after the first round was completed (Xc is terminal node)

2.1.2 Second Round

The anonymized node creates a new array of Setup Packages with some additions

Second Round Setup Package

- a first round success flag
- updated dummy package creation info

all X-nodes

- updated information on the next node / previous node (connection ID, IP/Port, public key, SSL certificate), now pointing to the next/previous X-node

ENTRY-nodes

- AP address this path should be registered under
- a ready-made routing table entry, signed with the routing key of the AP address owner

The anonymized node then encrypts, signs and arranges these setup packages again in a randomized order in an array and sends it to the first routing node in the sequence.

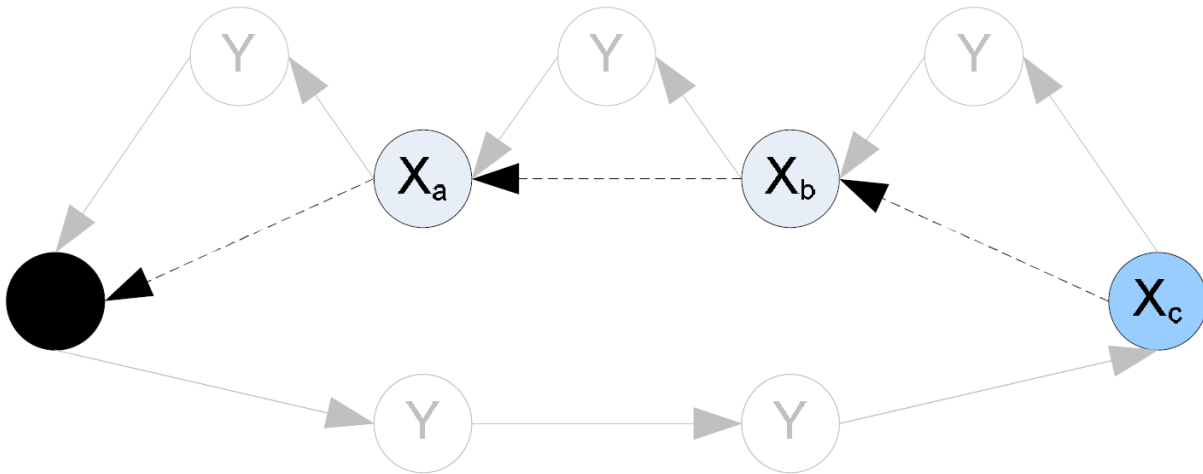


Figure 6: routing path after the second round

Routing Node

Again each routing node iterates through the array, until it can find a decryptable setup package. The integrity of the package is verified like in the first round (hash and signature). Furthermore it also confirms the success flag. Then it uses the seeds and flags found in the setup package to create new dummy packages to replace its own package in the array.

Y-Nodes If the node is a Y-Node it forwards the array to the designated next node, disconnects all its connections, deletes all stored data and resets its state to the exact state it had before the connection.

X-Nodes The X-node verifies that the updated previous node IP address and ID do not match the ones the node is currently connected to.

Then it waits for a new connection coming from the updated previous node IP address. Only after such a connection was successfully established (correct IP, ID and SSL certificate), it performs the usual modifications to the array and passes it on to the next node designated in the first round (the connection should still be active).

After that it verifies that the updated next node IP address and ID do not match the ones the node is currently connected to. After the verification the connection from the first round is terminated and a new connection to the updated IP address of the next node is attempted. Upon a successful connection the same array is passed on again on this connection.

At last the X-node finishes the setup by generating stream encryption keys using seeds and parameters from the first round setup package.

Terminating X-Node (Both) Since terminal nodes are at the end of the routing path, there will be no new connection like for the X-nodes above. Thus this step is skipped, and the terminal nodes start right away with the array modifications, sending it on the connection established in the first round. After closing this connection they attempt to connect to the updated previous node IP-Address and upon successful verification send the same array again on the new connection.

Terminating X-Node (ENTRY) The terminating node submits the pre-signed routing table entry into the network database.

The routing entry path has now been officially announced and other any use can use it to communicate with the anonymized node β .

The terminating ENTRY node is now waiting for arbitrary connections (see Routing Tunnels 2.2).

Terminating X-Node (EXIT) The terminating node waits for connection requests coming along the established routing path from the anonymized α node containing an AP-address to connect to.

In case of such a connection request it establishes a routing tunnel to the designated entry node of that AP-address (see Routing Tunnels 2.2).

The second array should also traverse the whole circle and end up at the anonymized node again. The first X-node of the resulting routing path (see X_a in the figure below) will establish a direct connection to the anonymized node (thinking it is just another X-node) and forward the array a second time like described above. With the second round array reaching the anonymized node twice, the second round is completed and the routing path is established.

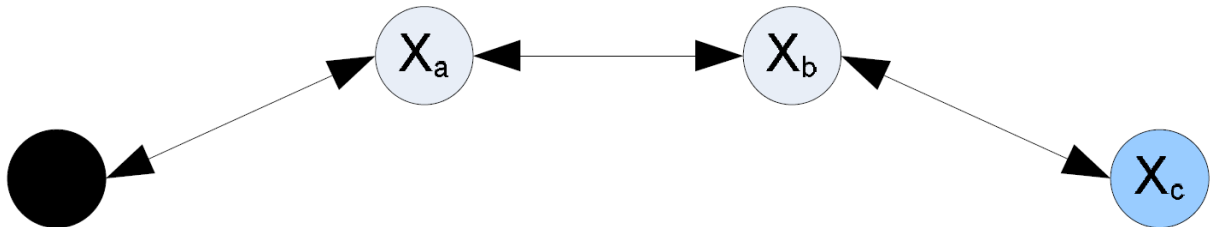


Figure 7: final established routing path

2.2 Routing Tunnels

When both sides of the communication have established routing paths, the corresponding EXIT and ENTRY nodes need to establish a Routing Tunnel between them. This second part of the design will explain, how such tunnels are created and maintained.

2.2.1 AP-Address Resolution

If contact to a specific node needs to be established, it is done by the use of **AP-Addresses** (Anonymous Protocol Address). The responsibility of resolving such AP-Addresses lies with the EXIT-nodes. This EXIT-node is communicated the AP-Address to connect to from the anonymous α -node. It then queries the **Network Database** (see 2.3) for the IP-Address of the corresponding ENTRY-node, as well as public-key and certificate of the anonymized β -node the AP-Address belongs to.

In the special case that β does not need anonymization (e.g. with β being a public service, like a search engine), the AP-Address is resolved directly to the IP-Address of β with β being its own ENTRY-node. However this makes no difference for α , as there is no way to tell if the resolved IP-Address leads to the ENTRY-node of a Routing Path or β directly.

The special case that α decides not to be anonymized is analogical.

2.2.2 Secure Establishment of Routing Tunnels

Now that the EXIT-node knows the IP-Address of the ENTRY-node, the next step is to actually form a connection between both nodes. We call such a connection a **Routing Tunnel**.

It must be understood that the Routing Tunnel setup is different, depending on whether the Routing Path is **outbound** (having an EXIT-node) or **inbound** (having an ENTRY-node).

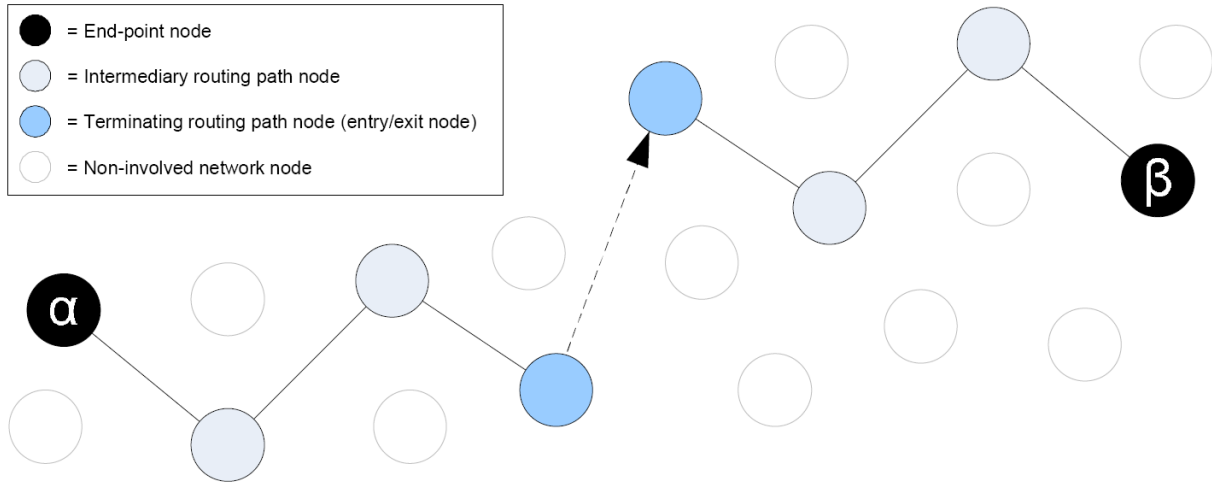


Figure 8: a routing tunnel

Outbound Routing Tunnel Setup Procedure

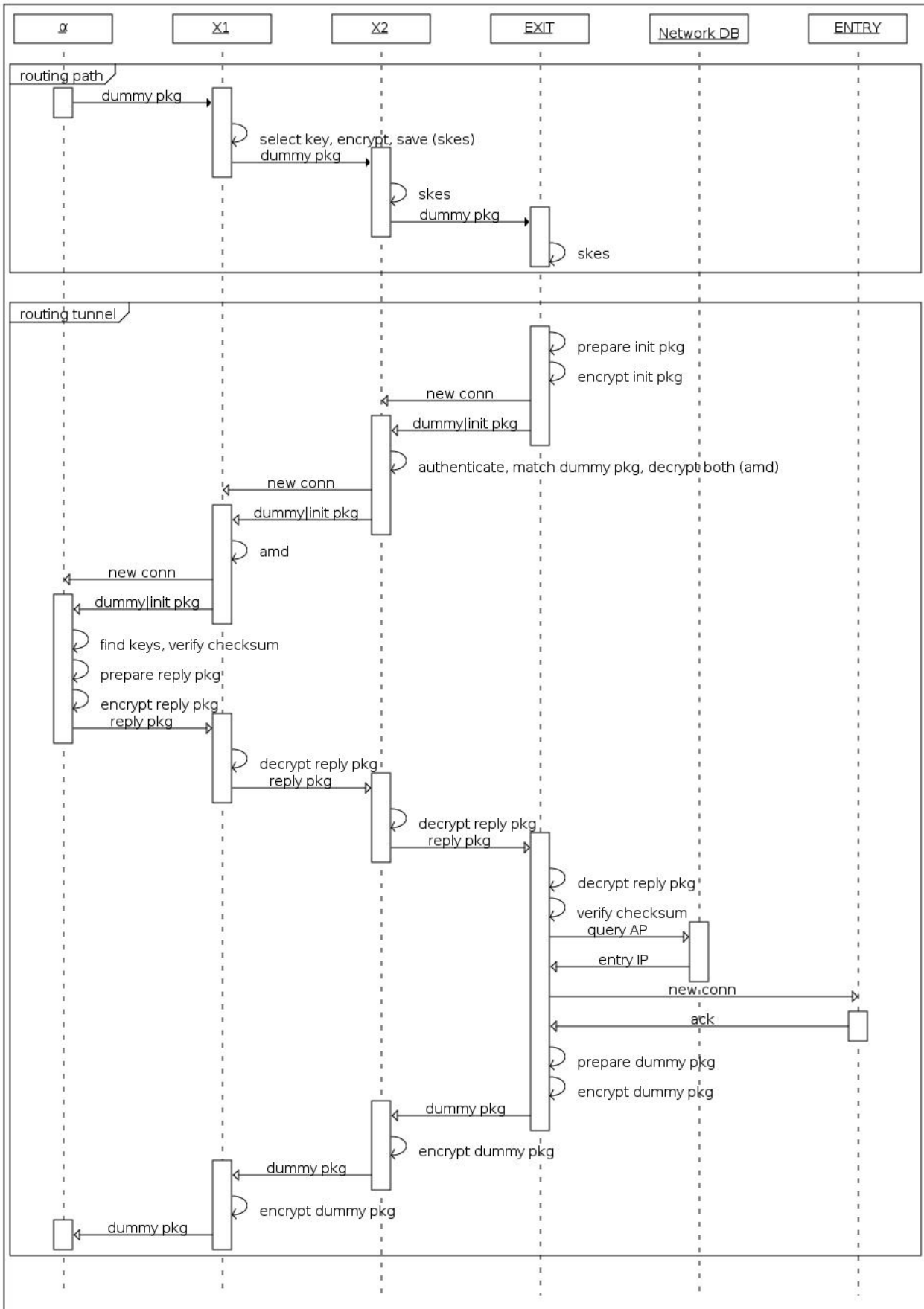


Figure 9: Outbound Routing Tunnel Sequence Diagram

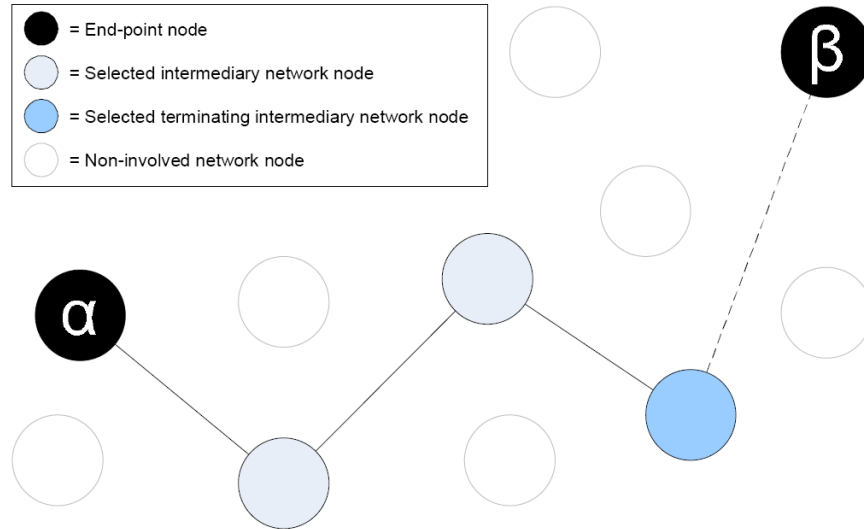


Figure 10: outbound routing tunnel

When α wants to establish a connection to a certain AP-Address, it creates a random dummy package having the size of one symmetric cypto block (e.g. 128 Bit) and sends it on the **Routing Path**. Each routing node then, randomly selects one prepared stream encryption key (see 2.1.1), encrypts the package it with it, stores package and key as a tuple and passes the package on to the next node. The EXIT-node, upon receiving and encrypting the dummy package, prepares a so called **tunnel initialization package** again having the size of one symmeric cipher block and containing: a crypto key initialization block and a checksum for the package contents. The EXIT-node then encrypts the tunnel initialization package with the previously chosen stream encryption key. Next it establishes a new connection to the previous node, designated in the second round setup package (see 2.1.2) authenticated by the connection id, and sends the received dummy package followed by the created routing tunnel initialization package to the previous node. Each routing node then authenticates the connection (IP, ID, certificate) and receives both packages and decrypts² them with the previously selected key and matches the decrypted dummy package with the stored one. If the two match, a connection to the previous routing node (designated in the second round setup package^{2.1.2}) is established, and both packages are sent on to the previous routing node. After this the stored dummy package can be deleted. With the anonymized α -node receiving the dummy package and the tunnel initialization package, the routing tunnel has been established. However α still needs to find the keys selected by the intermediate routing nodes. Since the seeds for these keys came from α it can find the correct keys by trial and error trying to “encrypt”³ the tunnel initialization package and checking the crypto key initialization block. When the keys are found, the checksum of the tunnel initialization package is checked.

Now having established a second parallel set of connections, the anonymized α -node prepares a tunnel initialization reply package, containaing: the desired AP-Address and port and a secure checksum. This package is then encrypted by the found stream keys in the appropriate order and sent on to the first routing node on the new connection. Each routing node then, decrypts the received reply package with its previously selected key and sends it on. The EXIT-node thus is able to decrypt the final layer, check the checksum, reads and resolves the AP-Address and connects to the corresponding ENTRY-node IP.

In case of an unsuccessful connection attempt to the resolved IP-Address of the ENTRY-node, the EXIT-node closes down all connection, causing a chain-reaction propagating all the way to the anonymized node, which by then knows the connection attempt failed.

In case of a successful connection attempt, the EXIT-node prepares a new dummy package filled with random data the same size as the last packages and sends it back on the new established connection. This package, like the last will be decrypted⁴ by every intermediate node and upon reaching the anonymized α -node, α knows the connection attempt was successful. With this the connection is complete and applications on higher layes can use it just like any TCP-connection.

²of course the tunnel initialization package is effectively encrypted, but it is important to use the decrypt function here

³now the tunnel initialization package is effectively decrypted, but it is important to use the encrypt function here

⁴effetively encrypted

Inbound Routing Tunnel Setup Procedure

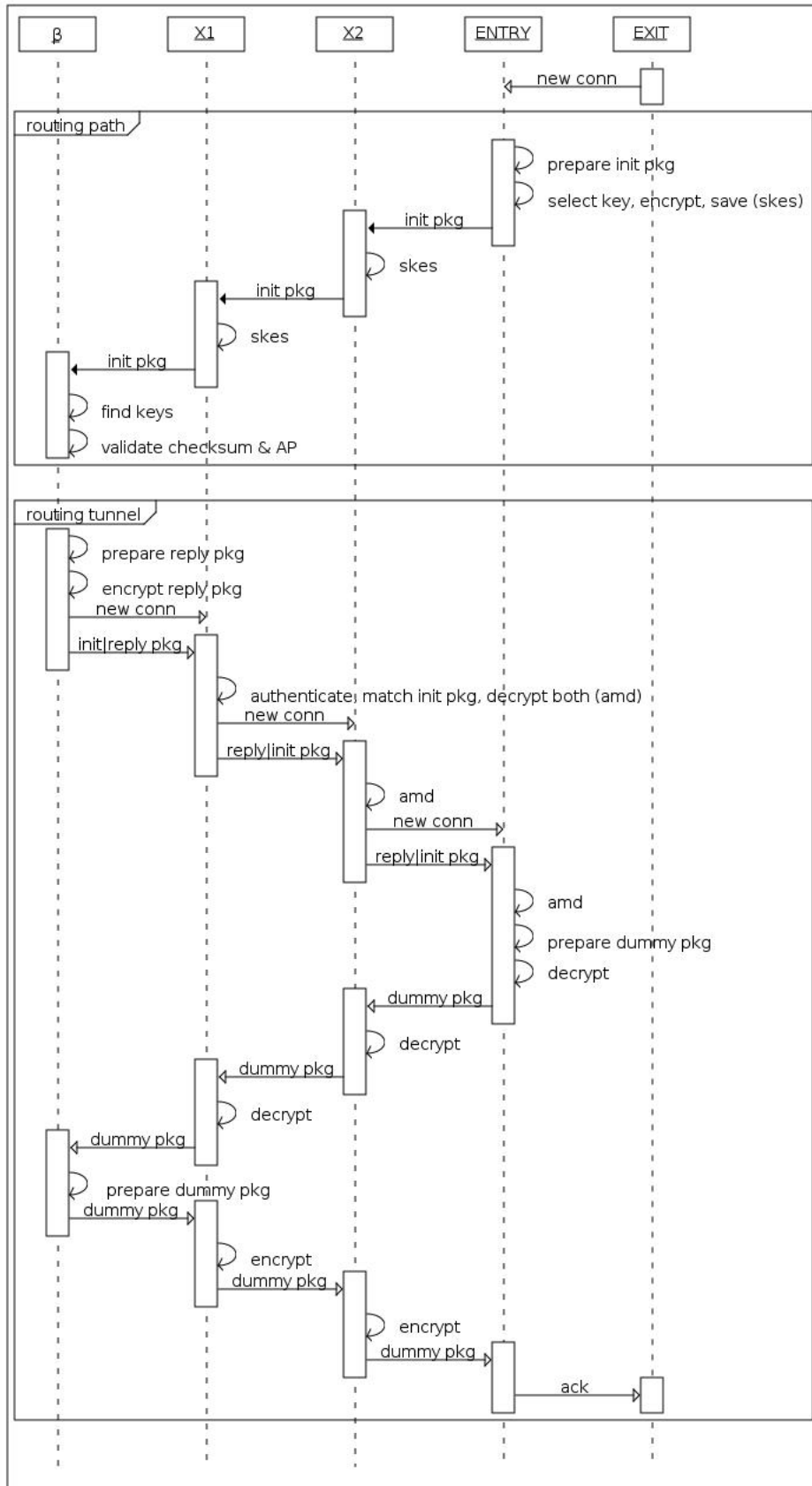


Figure 11: Inbound Routing Tunnel Sequence Diagram

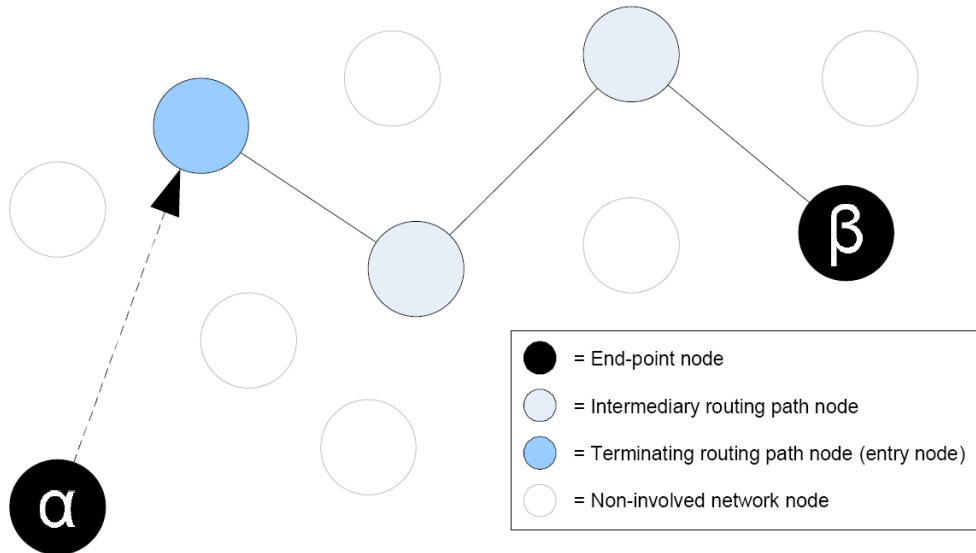


Figure 12: inbound routing tunnel

With the outbound scenario understood, we have to look on the other side of the tunnel, where an ENTRY-node awaits connection attempts and builds up a second routing tunnel for the opposite side. An important aspect is, that the setup procedure on this side is completely symmetrical to that of the outbound side, so that no external observer, nor any intermediate X-nodes can distinguish whether this is an inbound entry-path or an outbound exit-path.

Upon receiving a connection request from an EXIT-node, the ENTRY-node prepares the tunnel-initialization package containing: a crypto key initialization block, the IP-address of the connecting EXIT-node and the AP-address being targeted and sends it towards the anonymized β -node on the routing path established in (2.1). Just like in the above scenario the intermediate routing nodes randomly choose a pregenerated stream encryption key, stores package and key and encrypt the package before sending it on. The anonymized node β again is able to find out which keys were used and furthermore is able to read the package's contents. It verifies that the contained AP-address is indeed the one β registered for its entry-path. Then it prepares a tunnel initialization reply package containing a series of communication flags and a checksum and encrypts it with the keys it found out in the appropriate order. Finally it establishes a new connection to the next routing node and sends a copy of the received tunnel initialization package directly followed by the reply package. The intermediate routing nodes then proceed like in the previous scenario by, authenticating the connection (ID, IP, certificate), decrypting one layer of both packages, match the tunnel initialization package with the stored one and establish a new connection to the next routing node on which they pass on the two packages. The stored package can then be discarded.

With the ENTRY-node receiving and decrypting the reply package, matching the checksum and reading the flags, the inbound routing tunnel would be complete. However for the sake of symmetry another dummy package is created and sent back on the newly established path to the anonymized β -node. Upon receipt of this dummy package, β discards it and sends back yet another dummy package to the ENTRY-node on the newly established path, which in turn also discards it. Now the application layer is informed and applications can use the connection, just like any TCP-connection.

By this procedure the setup process though different in nature, looks identical and symmetrical to any outside observer, as well as any intermediate X-node.

2.2.3 Secure Connections over Routing Tunnels

With the routing tunnel being established communication over this tunnel is performed like this:

Any data that arrives on a node in the routing tunnel is decrypted or encrypted, depending on the direction (coming from the previous or next node), with the stream encryption key selected during the tunnel setup and sent on in the same direction. The anonymized node knows all the stream encryption keys and their respective order and can thus quickly encrypt and decrypt all data it receives or wants to send. The data passing the routing nodes is encrypted byte-wise to account for cases where not enough data is sent to fill an entire cipher block (e.g. 128

Bit). To avoid the same byte equals same encryption scenario, after each successful encryption, an internal state is updated.

2.3 Phantom Network Database

2.3.1 Distributed Hash Table

The basis for the **Phantom Network Database (PND)** is a distributed hash table (DHT). The protocol design includes the DHT design at an abstract level, so that the actual DHT implementation does not matter. A DHT is suitable for storing and broadcasting information for AP-Address resolution, because they are designed to handle constantly departing and joining DHT nodes and allow to quickly broadcast certain messages to all DHT nodes.

Every Phantom node simultaneously acts as DHT node in the Phantom Network Database. Every DHT node can store data, by submitting a key (sequence of bytes) followed by data (AP-Address and the corresponding IP-Address of the entry-path together with a certificate and a public-key). Every DHT node can then also retrieve information from the Network Database by submitting a query for afore mentioned keys.

2.3.2 Phantom Network Database

Building on to this DHT abstraction, the Phantom Network Database incorporates seven principles and abilities:

1. the database should be resilient to injection of false/unauthentic data (voting algorithms and signed data)
2. the database should be resilient to net splits
3. the database should support virtual tables, table fields and table records for data storage (only special formats allowed)
4. the database should be able to return random records in a secure fashion (limit one node's sphere of influence to one record)
5. the database should enforce a permission system (using signed requests and certificates)
6. the database should enforce expiry dates for table records
7. the database should provide a "comman channel" where authorized commands can quickly be sent to all DHT nodes (react to attacks and problems)

2.3.3 Phantom Network Database API

```
RegisterMyNodeInTheNetwork(own_ip_address, communication_certificate, path_building_certificate)
```

called each time a node goes online to participate in the network as DHT node

```
ReserveNewAPAddress(routing_certificate)
```

acquire an AP-address for a limited time

```
ExtendAPAddressLease(ap_address, signed_lease_request, routing_certificate)
```

extend lease for ap_address with valid signature created with the same certificate used when reserving ap_address

```
UpdateRoutingTableEntry(ap_address, signed_routing_entry, routing_certificate)
```

this is called to add, remove or update an entry-path for its AP-address

```
GetRandomNodeIPAddresses(noof_addresses)
```

returns noof_addresses IP-addresses of participating DHT nodes, which can then be used for building up a Routing Path (see 2.1)

```
GetEntryNodesForAPAddress(ap_address)
```

resolves ap_address to the IP-address of the corresponding entry-path, if registered

3 Implementation

3.1 Libraries

3.1.1 OpenSSL

OpenSSL is an open source implementation of the SSL and TLS protocols. The library implements basic cryptographic functions and provides various utility functions.

Within the Phantom Protocol we make heavy and various use of this library:

- asymmetric encryption (public/private keys) (RSA): *CryptoMachineRSA.h/cpp*
- signature and verification (certificates/certificate private keys) (RSA): *CryptoMachineRSA.h/cpp*
- symmetric encryption (AES): *CryptoMachineAES.h/cpp*
- hash functions (SHA256): *HashFunctions.cpp*
- OpenSSL streams and socket wrappers: *PhantomOpenSSLSocket.h/cpp* *PhantomOpenSSLServer.h/cpp* *PhantomOpenSSLClient.h/cpp*
- OpenSSL stream certificates and verification: *PhantomOpenSSLSocket.h/cpp*
- OpenSSL utility functions (e.g. error codes): *Utility.cpp*

OpenSSL provides an API here: <http://www.openssl.org/docs/>

but as many functions and structs are not covered or even mentioned there, it is recommended to use the open source code of the project itself as a reference: <http://www.openssl.org/source/>

3.1.2 Google Protobuf

Google Protobuf is an open source serialization framework, that allows class objects to be serialized, sent over network, received, parsed and restored at the other end.

Within the Phantom Protocol we utilize Protobuf to serialize and send **SetupPackages** to the routing nodes that we want to use for the **Routing Path** (see).

Each class, which's object we want to be serializable/parsable, has a *.proto* description file. Out of this file the C++ header and source file are created automatically using the **protoc** compiler coming with the library. In Phantom we then create our own class (e.g. SetupPackage), inheriting the auto-generated class (e.g. SetupPackageProto), providing an API to create, manipulate and serialize setup packages adapted to the project's purposes and needs.

We use the following serializable classes:

- *SetupPackage.h/cpp* < *SetupPackageProto.pb.h/cc* < *SetupPackageProto.proto*

Google Protobuf is documented here: <http://code.google.com/apis/protocolbuffers/docs/overview.html>

3.1.3 Google Test Primer

Google Test Primer is an open source unit test framework. Implementing this library simplifies, structures and organizes the creation and running of tests.

Within the Phantom Protocol we use it to write and run unit tests for most classes and key functionalities. In the base directory of the project, all unit tests can be found in the subfolder *test/*.

Google Test Primer is documented on this page: <http://code.google.com/p/googletest/wiki/GoogleTestPrimer>

3.1.4 GLog (GoogleLog)

3.1.5 YAML-cpp

YAML-cpp is a parser for documents written in YAML (yaml ain't markup language). YAML is more human-friendly / human-readable than other markup languages, such as XML.

Within the Phantom Protocol we utilize YAML for the config file, and YAML-cpp for parsing it (*.phantom.cfg*, *Config.h/cpp*).

Furthermore, until the **Phantom Network Database** (see 2.3) is implemented, we use a list of computers with corresponding IP, certificate and public key written in YAML as a substitute (*nodelists* directory).

YAML is documented here: <http://yaml.org/spec/1.2/>

YAML-cpp is documented here: <http://code.google.com/p/yaml-cpp/>

3.1.6 GC (Garbage Collector)

Boehm-Demers-Weiser conservative garbage collector is garbage collecting replacement for C malloc and C++ new.

For more information about gc, please refer to this site: http://www.hpl.hp.com/personal/Hans_Boehm/gc/

3.2 Overview

3.2.1 Routing Path Sequence Diagramm

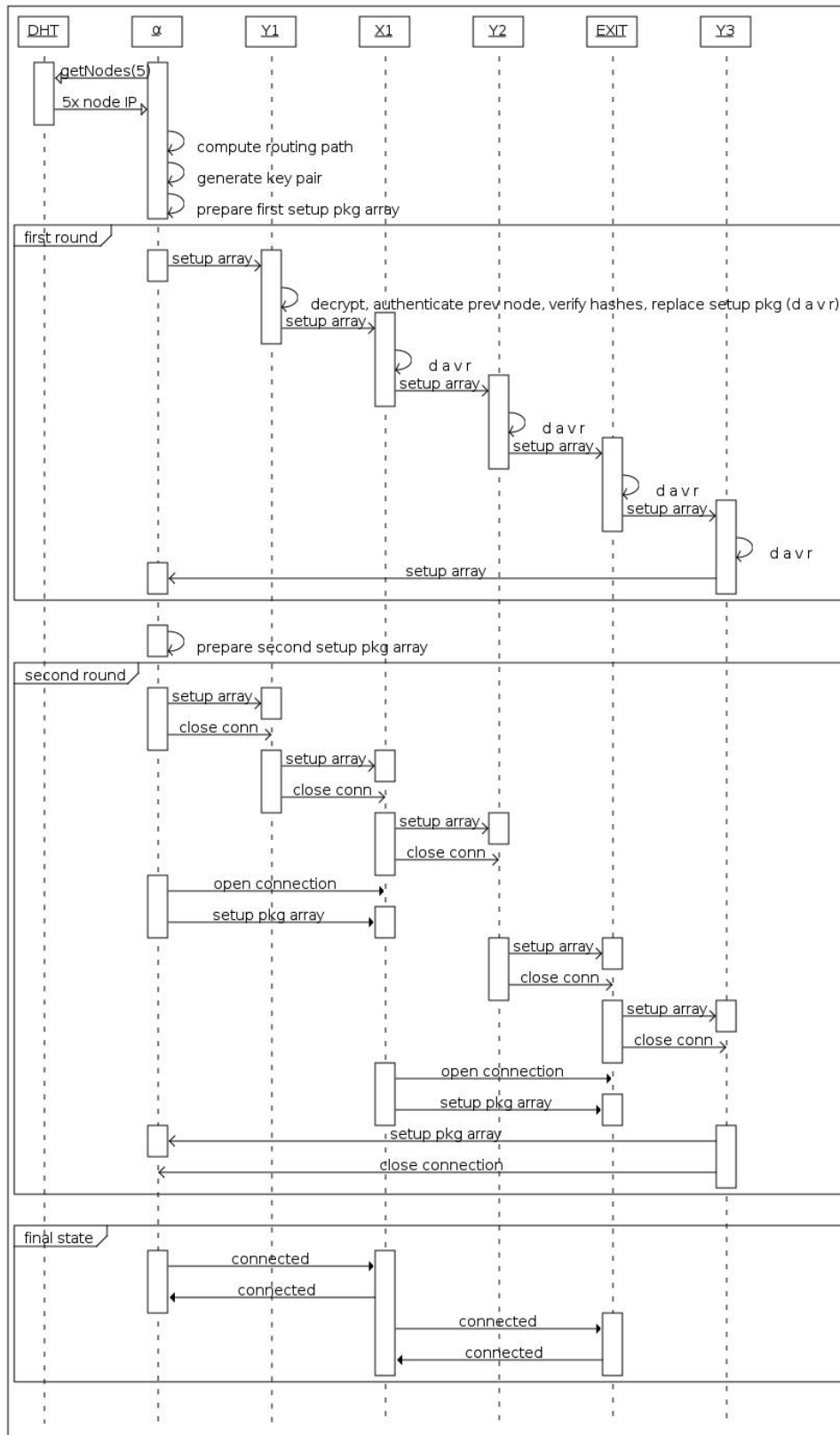


Figure 13: Sequence Diagram

3.2.2 Inheritance Diagram

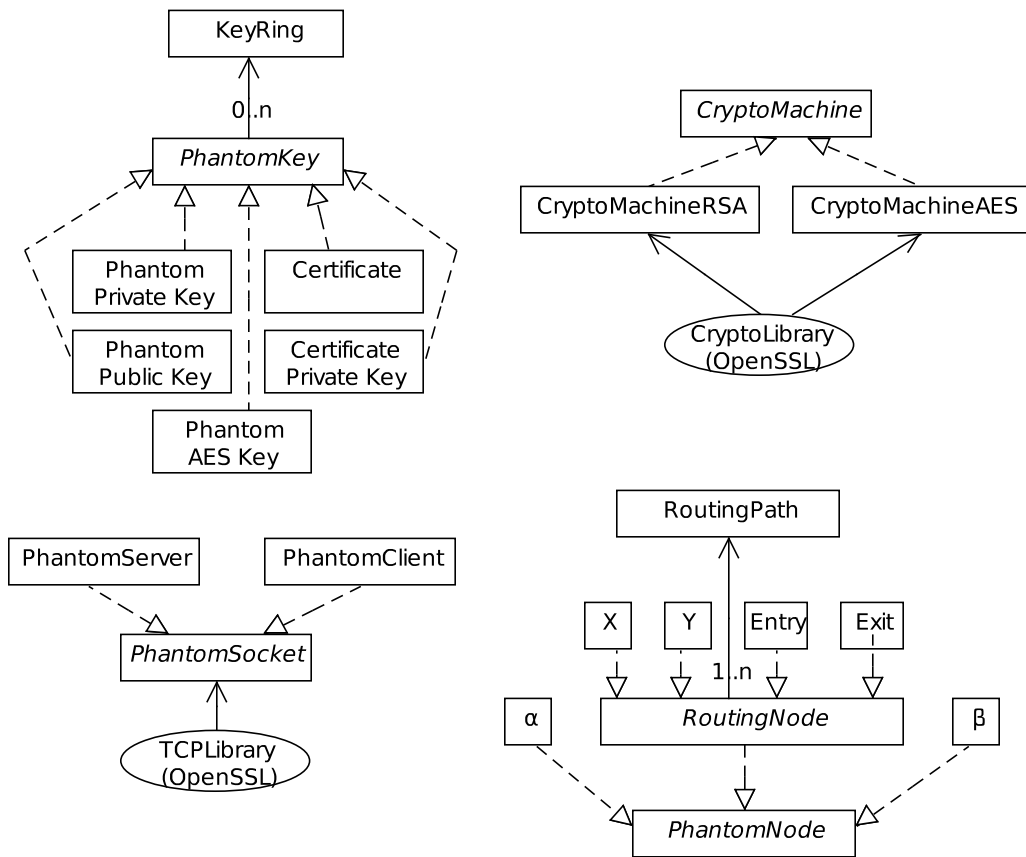


Figure 14: UML Inheritance Diagram

3.2.3 Composite Diagram

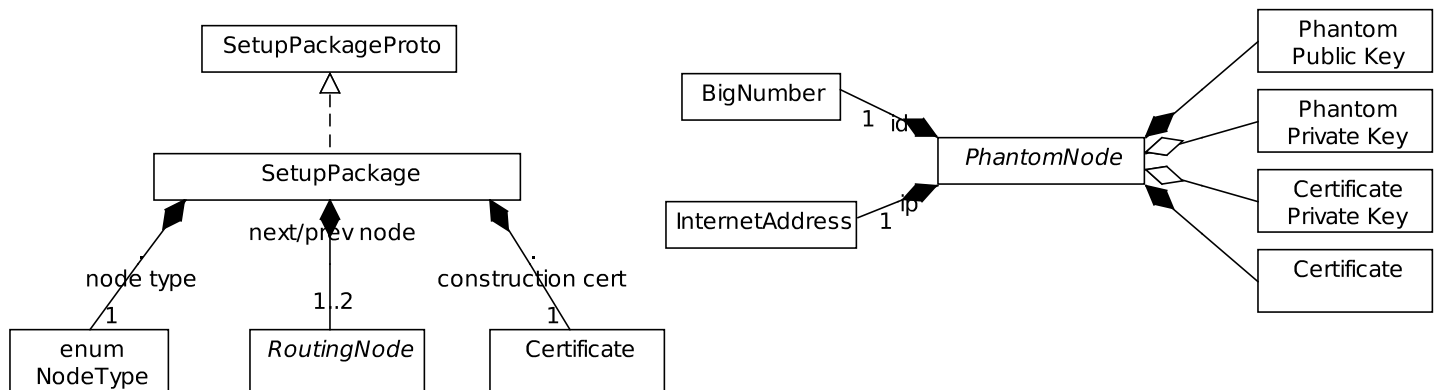
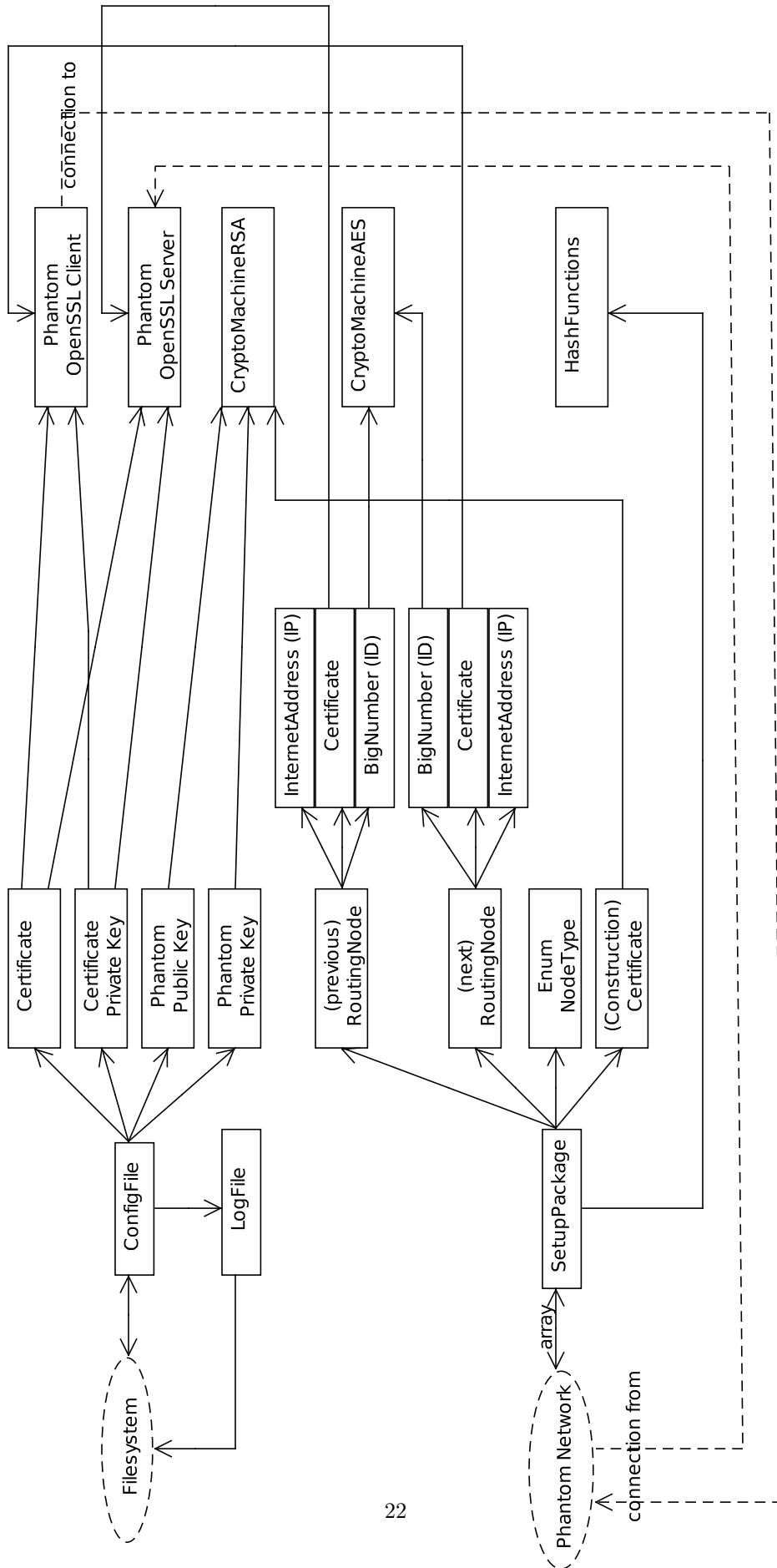
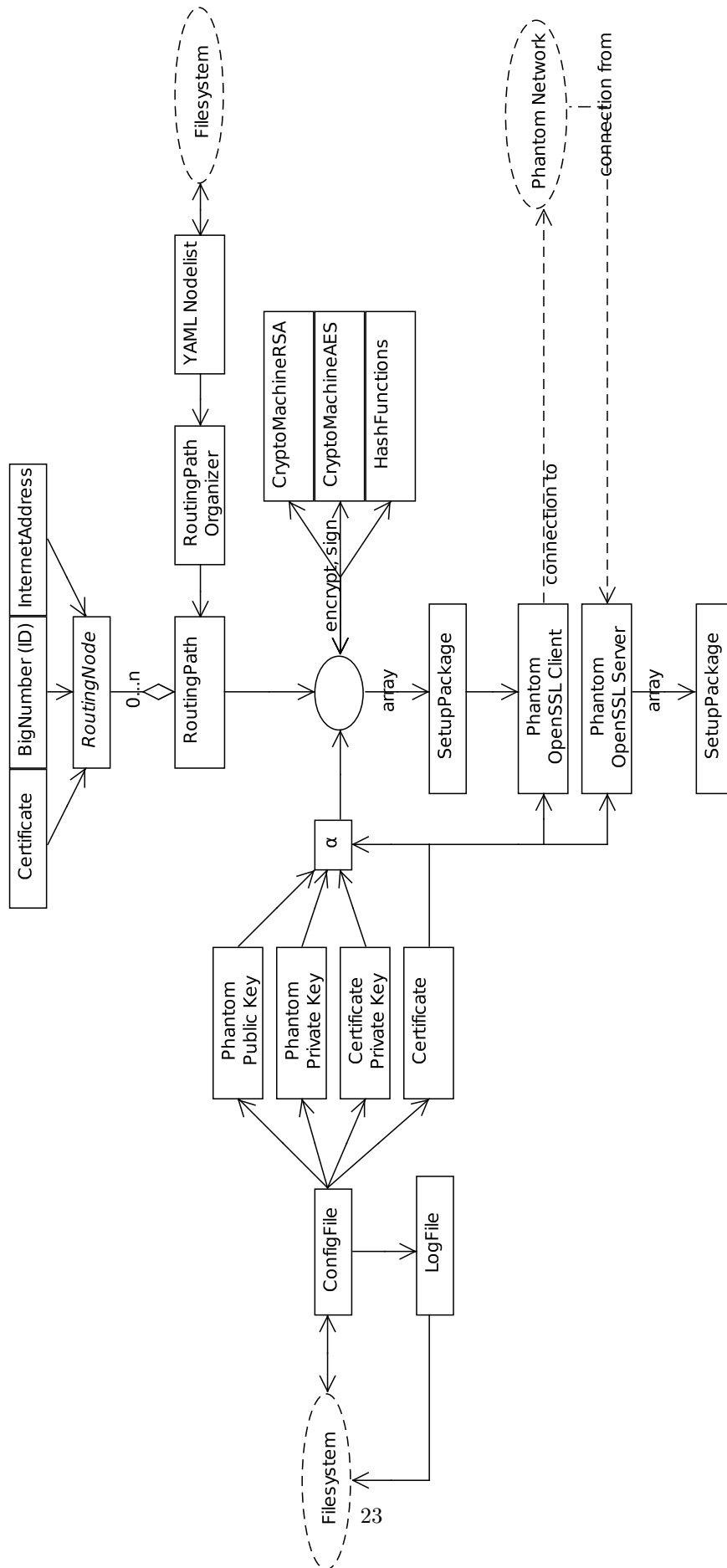


Figure 15: UML Composite Diagramm

3.2.4 Phantom Routing Node Prototype



3.2.5 Phantom Anonymized Node Prototype



3.3 Environment

3.3.1 Logging

3.3.2 Config File

In the Phantom Protocol, there are certain degrees of freedom for the user like the security level. To allow the user to choose these settings, the Phantom Protocol implements a **Config File** and a Config File Parser as an interface to the user. For now settings can be changed by editing the config file with a text editor. At a later point however a Graphical User Interface might be considered to simplify the interface.

Naming

The config file is named *.phantom.cfg* and is by default placed in the home folder (Unix) of the user executing Phantom.

Format

The config file is written in YAML. YAML is a human friendly markup language, which is easy to read and edit. For more details see 3.1.5.

Settings

version: version of the phantom protocol

security-level: determines the length of the routing path
possible settings are: OFF, LOW, MEDIUM, HIGH, PARANOID

upper-bandwidth-limit: limit for what amount of bandwidth phantom may use to route other users

basepath: absolute path to the phantom base directory

log-file: (relative) path to the logfile

certificate-location: (relative) path to the openssl certificate trust store

keys

public-key: path to personal public key

private-key: path to personal private key

certificates

certificate: path to personal certificate

private-key: path to personal certificate private key

port: preferred connection port

API

The config file can be accessed through the static class/object's `Config.h/cpp` methods. For each attribute there is a get method to access it, for example the basepath can be recovered by using

```
Config::instance()->getBasePath();
```

To use a config file not in the default location (Unix: *~/phantom.cfg*), instance can be called with a filename argument:

```
Config::instance(string filename)->getBasePath();
```

```
#phantom config file
version: 1.0 security-level: MEDIUM
upper-bandwidth-limit: 100 kbit/s
basepath: /home/cip/2003/simiprin/Diplomarbeit/Reposit
log-file: $basepath/log/phantom.log
certificate-location: $basepath/certs
keys:
  public-key: $basepath/keys/rsa_public_key.pem
  private-key: $basepath/keys/rsa_private_key.pem
certificates:
  certificate: $basepath/certs/certificate.crt
  private-key: $basepath/certs/certificate_private_key
port: 4002
```

3.3.3 Serialization

Being a network protocol sending data from one computer to another is a vital part of the Phantom Protocol. However many things must be considered before sending data from one socket to another: what about the computer's byte order, encoding, how to send objects of classes, etc. To not overcomplicate this aspect of the protocol a serialization library was chosen to be used for serializing and deserializing objects to be sent over the network.

For this project **Google ProtoBuffers** (see 3.1.2) were chosen. Using ProtoBuffers a serializable class is created by defining a *.proto* file for them and then running the *protoc* compiler to create source and header files out of this definition. Then a wrapper class adapted to the project can be created, that when inheriting from the proto files is then (de)serializable. For more information about Google Protobuffers, please refer to the library section on chapter 3.1.2.

SetupPackage

the essential objects to be transmitted between sockets building the **Routing Path** are **Setup Packages**. A Setup Package holds all information necessary to set up a routing node for a routing path. A list of components can be found in the protocol design (see 2.1.1). These components were implemented to be easily serialized, sent, received and parsed.

```
node_type enum
prev_node_id string
next_node_id string [optional]
prev_node_ip string (ip), port (int32)
next_node_ip string (ip), port (int32) [optional]
construction_certificate string
prev_node_certificate string
next_node_certificate string [optional]
hash_code bytes
```

This logic is then ended in a wrapper adapted for the use within the Phantom Protocol project.

All the information about the next/previous nodes are set/returned as **RoutingNode** with ID as **BigNumber**, IP as **InternetAddress** and certificate as **Certificate**. The construction certificate of the anonymized node is also dealt with as **Certificate**. The endum node type is converted to **PhantomNode::NodeType** and finally the hash code bytes are treated as **const char***.

Because the hash code of the setup package must be computed, before the hash code can be added to it, the constructor of a setup package then looks like this:

```
message SetupPackageProto {
  enum NodeType { ALPHA = 0; BETA = 1; X = 2; Y = 3; E
  required NodeType node_type = 1 [default = X];
  required string prev_node_id = 2;
  optional string next_node_id = 3;
  message InternetAddress {
    required string ip = 1;
    required int32 port = 2;
  }
  required InternetAddress prev_node_ip = 4;
  optional InternetAddress next_node_ip = 5;
  message Certificate {
    required string cert = 1;
  }
  required Certificate construction_certificate = 8;
```

```
SetupPackage(PhantomNode::NodeType nt, phantom::Certificate* const_cert, PhantomNode* prev_node, PhantomNode* next_node,
```

with `next_node` being optionally NULL for SetupPackages for terminating nodes. The hashcode can then be computed using:

```
const char* hash = hash::computeHashSHA256(setup);
SetupPackage::setHashCode(hash, hash::getHashSHA256Length());
```

with all being set, we can serialize the package using:

```
int byte_size = SetupPackage::getSize();
char* buffer = new char[byte_size];
SetupPackage::SerializeToArray(buffer, byte_size);
```

after sending it on one and receiving it on another socket, we can reconstruct it using:

```
int byte_size; //known since the buffer is sent as vector
char* buffer = new char[byte_size];
SetupPackage::ParseFromArray(buffer, byte_size());
```

finally we have rebuilt our SetupPackage on a remote computer and can read its contents with get methods. To verify the setup package's hash code, we must first read and then remove the hash code inside the package; then we can compute our own hash and compare the two of being equal:

```
const char* sent_hash = SetupPackage::getHashCode();
SetupPackage::removeHashCode();
const char* computed_hash = hash::computeHashSHA256(setup);
HASH_CHECK(sent_hash, computed_hash, hash::getHashSHA256Length());
```

HASH_CHECK will throw a SecurityException, if the two hashes doesn't match.

3.3.4 Unit Testing

Since the Phantom Protocol Project is intended to be developed beyond this thesis, and since it is meant to be used in a security critical area, a unit testing framework constantly testing its main components and functionalities becomes necessary. Again as establishing such a framework would be much unnecessary work, a premade library for this exact purpose has been chosen and included into the project: **Google Test Primer** (see also 3.1.3).

Location

All test cases including the test binary main class are located in the subfolder `test/` of the phantom base directory. Inside the folder there is located an own Makefile, that is automatically called together with the `src/Makefile` when executing `make` in the phantom base directory. The binary for executing all test cases, simply called `test`, can be found in the subfolder `bin/` of the phantom base directory.

Execution

Simply run the `bin/test` binary in a shell, and all test cases will consecutively be executed. The test framework will print results of the tests to the shell.

Writing Test Cases

For writing your own test cases, you can take the structure of preexisting test cases as an reference or look at the reference here: <http://code.google.com/p/googletest/wiki/GoogleTestPrimer>

Please place them in the `test/` subfolder of the main directory.

List of Test Cases

- BigIntegerTest
- CertificateVerificationTest
- ConfigTest
- CryptoMachineAESTest
- CryptoMachineRSATest
- HashTest
- KeyRingTest
- LogTest
- OpenSSLSocketTest
- RSA_AESTest
- RandomGeneratorTest
- SerializationTest
- SignVerifyTest
- UtilityTest
- YAMLLTest

3.3.5 Garbage Collection

To avoid memory leaks a garbage collector has been added to the environment. We use the Boehm-Demers-Weiser conservative garbage collector, which replaces the C++ new operator. It allows to allocate objects in standard fashion, but does not require to free the memory afterwards, as it will take care of it. C++ classes can inherit from `class gc` to make garbage collection available for objects instanciated from this class.

```
//gc
#include <gc/gc.h>
#include <gc/gc_cpp.h>
class PhantomNode : public gc {...}
```

All base classes include this libraries and inherit from `gc`, like described above.

3.4 Components

3.4.1 Encryption / Decryption

The Phantom Protocol requires both asymmetric encryption (public key, private key), as well as symmetric encryption. The choice of the cipher is up to the implementation.

For this first implementation RSA (1024 bit) was chosen for asymmetric and AES (128 bit) for symmetric encryption. At a later point other algorithms can be added and a choice can be offered to the user. For the illustration of the concept, however, one cryptographic algorithm is enough. The algorithms have been taken from a cryptographic library (*openssl/crypto*).

```
TEST_F(SignVerifyTest, SignAndVerifyTest) {
    string message = "The Pirate Song:\n\nRising the fla
    vector<char>* mvec = utility::string2vector(message)
    vector<char>* sig;
    CryptoMachineRSA* crypto = new CryptoMachineRSA();
    //sign
    CertificatePrivateKey* key = new CertificatePrivateKey
    crypto->loadCertificatePrivateKey(key);
    sig = crypto->sign(*mvec);
    //verify
    Certificate* cert = new Certificate(Config::instance
    crypto->loadCertificate(cert);
    ASSERT_TRUE(crypto->verify(*mvec, *sig));
    delete sig, mvec, crypto, cert, key;
}
```

Figure 20: SignAndVerify Unit Test

Cryptographic Base Class

Class *CryptoMachine.h* serves as an abstract base class for classes implementing symmetric or asymmetric encryption. It contains some virtual functions required by all cryptographic ciphers.

Asymmetric Encryption: RSA

As stated RSA was chosen as asymmetric cipher. It is implemented in the class *CryptoMachineRSA.h/cpp*, which inherits from the cryptographic base class *CryptoMachine.h*.

As asymmetric cipher both public keys and private keys can be loaded and saved.

```
void loadPublicKey(std::string public_key_file);
void loadPrivateKey(std::string private_key_file);
void loadPublicKey(PhantomPublicKey& public_key);
void loadPrivateKey(PhantomPrivateKey& private_key);
void savePublicKey(std::string public_key_file);
void savePrivateKey(std::string private_key_file);
```

Keys to be loaded can be stored in the filesystem or in *PhantomPublicKey.h/cpp* *PhantomPrivateKey.h/cpp* objects. New keys can be generated using

```
void generateKey(int key_length);
```

Messages to be encrypted or decrypted are always dealt as vector of bytes (`vector<char>`). This choice of data structure allows us to deal with any sequence of bytes and allows us to store the length of the sequence as well. The *Utility.cpp* class offer methods to convert from other data structures like `string` to `vector<char>`.

```
vector<char>* encrypt(vector<char>& message);
vector<char>* decrypt(vector<char>& message);
```

For encryption depending on the padding mode and the key length, messages may not exceed a certain length.

```
flenint RSA_public_encrypt(int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding);
int RSA_private_decrypt(int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding);
flen must be less than RSA_size(rsa) - 11 for the PKCS #1 v1.5 based padding modes,
    less than RSA_size(rsa) - 41 for RSA_PKCS1_OAEP_PADDING
    and exactly RSA_size(rsa) for RSA_NO_PADDIN
```

from the OpenSSL man page.

The solution to this is to split messages longer than this maximum length in chunks of the maximum allowed length, encrypt them individually and join them together.

Algorithm 1 RSA encrypt

```
#include <openssl/rsa.h>
std::vector<char>* CryptoMachineRSA::encrypt(std::vector<char>& message) {
//message length
int m_len = message.size();
//error code and return value (number of encrypted bytes)
int err, ret;
//return vector
std::vector<char>* rv;
//unencrypted byte array (input for RSA_encrypt)
unsigned char* from = utility::vector2unsigned_char(message);
//encrypted byte array (output from RSA_encrypt)
unsigned char* to = new unsigned char[RSA_size(this->rsa)]();
/*
 * NOTE: m_len must be less than RSA_size(rsa)-11 for PKCS#1 v1.5 padding,
 * less than RSA_size(rsa)-41 for RSA_PKCS1_OAEP_PADDING
 * and exactly RSA_size(rsa) for RSA_NO_PADDING.
 */
if(m_len > (this->getBlockSize()-11)) {
    rv = new std::vector<char>();
    //split too large vector in a list of vectors of appropriate size
    std::list<std::vector<char>*>* vlist = utility::split(message, this->getBlockSize()-11);
    //now iterate over all vectors
    std::list<std::vector<char>*>::iterator i;
    for(i=vlist->begin(); i != vlist->end(); ++i) {
        std::vector<char>* vt = (*i);
        std::vector<char>* evt;
        //encrypt vt
        evt = this->encrypt(*vt);
        //add it to return vector
        rv->insert(rv->end(), evt->begin(), evt->end());
        delete vt;
        delete evt;
    }
    delete vlist;
} else {
    ret = RSA_public_encrypt(m_len, from, to, this->rsa, RSA_PKCS1_PADDING);
    SSL_GUARD(ret == -1);
    rv = utility::char2vector((char*)to, ret);
}
delete[] from;
delete[] to;
return rv;
}
```

For decryption, we do the exact same thing: split the encrypted byte array in chunks of maximum length, decrypt them and join the together again.

Algorithm 2 RSA decrypt

```
#include <openssl/rsa.h>
std::vector<char>* CryptoMachineRSA::decrypt(std::vector<char>& message) {
//message length
int m_len = message.size();
//error code and return value (number of encrypted bytes)
int err, ret;
//return vector
std::vector<char>* rv = new std::vector<char>();
//unencrypted byte array (input for RSA_encrypt)
unsigned char* from = utility::vector2unsigned_char(message);
//encrypted byte array (output from RSA_encrypt)
unsigned char* to = new unsigned char[RSA_size(this->rsa)-11]();
/*
 * if vector size is bigger than RSA_size we have to split the whole vector
 * decrypt the resulting parts and then recombine them
 * (see also encrypt())
 */
if(m_len > this->getBlockSize()) {
    string d;
    //split large vector in a list of vectors of appropriate size
    std::list<std::vector<char>*>* vlist = utility::split(message, this->getBlockSize());
    //now iterate over these subvectors
    std::list<std::vector<char>*>::iterator i;
    for(i=vlist->begin(); i != vlist->end(); ++i) {
        std::vector<char>* vt = (*i);
        std::vector<char>* evt;
        //decrypt vt
        evt = this->decrypt(*vt);
        //add it to return vector
        rv->insert(rv->end(), evt->begin(), evt->end());
        delete vt;
        delete evt;
    }
    delete vlist;
} else {
    ret = RSA_private_decrypt(m_len, from, to, this->rsa, RSA_PKCS1_PADDING);
    SSL_GUARD(ret == -1);
    rv = utility::char2vector((char*)to, ret);
}
delete[] from;
delete[] to;
return rv;
}
```

Unit Test

There is a unit test testing asymmetric encryption with RSA in *test/CryptoMachineRSATest.cpp*.

Symmetric Encryption: AES

For this first implementation AES was chosen for symmetric cipher. In the construction of the **Routing Path** (see 2.1) symmetric encryption is used on top of asymmetric encryption. While the public key of the target node can be looked up in the **Phantom Network Database** (see 2.3), the symmetric key is generated during the set up process by the anonymized node. It serves a second function as connection id between two nodes and is transmitted

in clear text (below the outer SSL layer, see 3.4.5) and can then be used by the corresponding target node to decrypt the first layer of the message.

Since we only have one key with symmetric ciphers, which we generate at run time, the API for loading keys is as follows:

```
void loadKey(PhantomAESKey* key);
void loadKey(BigNumber* bn);
```

The first method loads the key stored in a *PhantomAESKey.h/cpp* object (standard key with additional check assuring correct key length). Since we use *BigNumber.h/cpp* objects to store afore mentioned connection ids, we can also directly load a connection id using it as symmetric key.

Generating keys is not necessary here, since we will only be using BigNumber ids.

Messages to be encrypted or decrypted are again given in byte array (`vector<char>`) format. The API for en/decryption is:

```
vector<char>* encrypt(vector<char>& message);
vector<char>* decrypt(vector<char>& message);
```

Since encryption and decryption in symmetric ciphers are symmetrical, the two methods above simply prepare the previously loaded key for de- respectively encryption, fill the message with padding bytes (use the number of padding bytes as value for padding bytes, see [PKCS5] 12.9) and call the subroutine `crypt(vector<char>& message, int mode)` with the corresponding mode of operation (AES_DECRYPT or AES_ENCRYPT). As block cipher mode for the first implementation **Cipher Block Chaining** (CBC, see 12.7) was chosen. CBC requires an initialization vector, which was added together with the key (the connection id) to messages.

Algorithm 3 AES encrypt

```
#include <openssl/aes.h>
vector<char>* CryptoMachineAES::encrypt(vector<char>& message) {
    //load key for encryption
    this->loadEncryptKey();

    // add Padding in case that we can not split vector in same sized chunks of AES block size
    int remainder = message.size() % AES_BLOCK_SIZE;
    int padding = 16 - remainder;
    if(remainder > 0) {
        while(remainder < 16) {
            message.push_back(int2char(padding));
            ++remainder;
        }
    }

    //encrypt
    return this->crypt(message, AES_ENCRYPT);
}
```

Algorithm 4 AES decrypt

```
#include <openssl/aes.h>
vector<char>* CryptoMachineAES::decrypt(vector<char>& message) {
    //load key for encryption
    this->loadDecryptKey();

    //decrypt
    vector<char>* decrypt = this->crypt(message, AES_DECRYPT);

    //remove padding
    int padding = char2int(decrypt->back());
    if(padding > 0 && padding < 16) {
        bool pad = true;
        // check that we really got a padding byte
        // and not a data byte by accident
        // by checking that all padding bytes have the same value
        char c = decrypt->back();
        int last = decrypt->size() - 1;
        for(int p = 0; p < padding; p++) {
            if ( (*decrypt)[last - p] != c) {
                pad = false;
                break;
            }
        }
        if(pad) {
            while(padding > 0) {
                decrypt->pop_back();
                --padding;
            }
        }
    }
    return decrypt;
}
```

Algorithm 5 AES crypt

```
#include <openssl/aes.h>
vector<char>* CryptoMachineAES::crypt(vector<char>& message, int mode) {
    unsigned char* inbuf = utility::vector2unsigned_char(message);
    unsigned char* outbuf = new unsigned char[message.size()]();
    //make a copy of init vector, since it will change during en/decryption
    unsigned char* iv = new unsigned char[AES_BLOCK_SIZE]();
    memcpy(iv, this->init_vector, AES_BLOCK_SIZE);
    AES_cbc_encrypt(inbuf, outbuf, message.size(), &this->aes_key, iv, mode);
    vector<char>* outvec = utility::unsigned_char2vector(outbuf, message.size());

    //clean up
    delete[] inbuf;
    delete[] outbuf;
    delete[] iv;

    return outvec;
}
```

Unit Test

There is a unit test testing symmetric encryption with AES in *test/CryptoMachineAESTest.cpp*.

There is also a test for consecutive application of RSA and AES to the same byte sequence found in *test/RSA_AESTest.cpp*.

3.4.2 Signature / Signature Verification

Signing messages and verifying the signature involves public and private keys and hash codes. Since we already got an implementation for RSA encryption/decryption, we can use it also for signing and verifying messages. However to emphasize the logical difference, as well as the difference in usage, we use different classes for the keys called **certificate** and **certificate private key** (*Certificate.h/cpp*, *CertificatePrivateKey.h/cpp*).

Thus it seemed a good design idea to put the API for loading certificates and certificate private keys, as well as for signing and verifying signatures in the already existing class implementing RSA: *CryptoMachineRSA.h/cpp*. For loading and saving certificates and certificate private keys, we have

```
void loadCertificate(Certificate* cert);
void loadCertificatePrivateKey(CertificatePrivateKey* key);
void loadCertificate(string filename);
void loadCertificatePrivateKey(string filename);
void saveCertificate(string filename);
void saveCertificatePrivateKey(string filename);
```

For generating a new certificate / certificate private key pair, the following API can be used

```
void generateCert(int key_length, int days);
```

where *days* denotes the validity of the certificate.

For signing we again treat messages as arbitrary sequence of bytes (`vector<char>`). What is signed is actually not the message itself, but its hash. For computing the hash of the message, we'll be using the class *HashFunctions.cpp* (see 3.4.3). The signature should be always sent together with the message. It's purpose is not to encrypt the message, but to make sure it was sent by the issuer of a certain certificate. That means for verification we need as parameters both the signature, as well as the message it was created for.

```
vector<char>* sign(vector<char>& message);
bool verify(vector<char>& message, vector<char>& signature);
```

Algorithm 6 RSA sign

```
#include <openssl/rsa.h>
vector<char>* CryptoMachineRSA::sign(vector<char>& message) {
    unsigned char* hash = hash::computeHashSHA256(&message);
    int sig_length = this->getSignatureLength();
    unsigned char* sig_ret = new unsigned char[sig_length]();
    unsigned int* sig_ret_length = new unsigned int[sizeof(int)]();

    SSL_GUARD (RSA_sign(NID_sha256, hash, hash::getHashSHA256Length(), sig_ret, sig_ret_length, this->r);
    vector<char>* signature = utility::unsigned_char2vector(sig_ret, *sig_ret_length);

    // clean up
    delete[] hash;
    delete[] sig_ret;
    delete[] sig_ret_length;
    return signature;
}
```

Algorithm 7 RSA verify

```
#include <openssl/rsa.h>
bool CryptoMachineRSA::verify(vector<char>& message, vector<char>& signature) {
    unsigned char* hash = hash::computeHashSHA256(&message);
    unsigned char* sig = utility::vector2unsigned_char(signature);
    bool result = (RSA_verify(NID_sha256, hash, hash::getHashSHA256Length(), sig, signature.size(), t

    //clean up
    delete[] hash;
    delete[] sig;
    return result;
}
```

Unit Test

There is a unit test testing signing and verification of signatures in *test/SignVerifyTest.cpp*.

3.4.3 Hash Codes

Hash Codes are required at three points in the protocol design:

- a hash code of the decrypted setup package
- a hash code of the entire encrypted setup packages array
- indirectly for signatures

Since hash codes are fairly easy computed, requiring no more than an algorithm, no extra class was created for hashes. Instead a collection of hash functions is provided in the source file *HashFunctions.cpp*. For the first implementation SHA256 was chosen as hash algorithm (in the *HashFunctions.cpp* file, there is also SHA1 as an alternative). SHA256 hash codes can be computed using the following API:

```
static unsigned char* computeHashSHA256(string text);
static unsigned char* computeHashSHA256(vector<char>* vec);
static unsigned char* computeHashSHA256(vector<vector<char> >* box);
static unsigned char* computeHashSHA256(SetupPackage* setup);
```

The methods have been adapted to the above listed requirements of the protocol design. Again the crypto library of OpenSSL was chosen to provide the actual implementations, for example:

Algorithm 8 SHA256 Hash Code

```
#include <openssl/sha.h>
static unsigned char* computeHashSHA256(vector<char>* vec) {
    unsigned char* hash_code = new unsigned char [SHA256_DIGEST_LENGTH]();
    char* vec_char = vector2char(*vec);
    SHA256(reinterpret_cast<const unsigned char*>(vec_char), vec->size(), hash_code);
    SSL_GUARD(hash_code == NULL);
    //clean up
    delete[] vec_char;
    return (hash_code);
};
```

Furthermore a method for hash comparison has been added to the source file:

```
static bool checkHash(unsigned char* hash1, unsigned char* hash2, int length);
```

Unit Test

There is a unit test for these hash functions, found in *test/HashTest.cpp*.

3.4.4 Sockets

The project contains two socket implementations: the Berkeley C Sockets, and OpenSSL Sockets

Berkeley Sockets

The project has a basic implementation of the Berkeley C Sockets realized in the classes

- *PhantomSocket.h/cpp*
- *PhantomServer.h/cpp*
- *PhantomClient.h/cpp*

With the later two both inheriting from the first one.

Since the project requires OpenSSL streams it is better to directly use the socket layer abstraction provided by OpenSSL. Thus another set of classes was written directly on top of the OpenSSL socket abstraction. The API for both type of sockets within phantom is basically identical, therefore we will only describe the OpenSSL sockets in detail. The main difference for the outside API is the remote socket file descriptors are returned in **BIO*** format instead of **int**.

Unit Test

A unit test illustrating the capabilities of **PhantomSocket** can be found in *test/BerkeleySocketTest.cpp*.

OpenSSL Sockets

OpenSSL brings its own socket layer abstraction, built on top of the Berkeley C Sockets. Since we will be requiring SSL streams at a later point it is beneficial to use the OpenSSL API to create and maintain sockets. The basic socket functionality is realized in three classes:

- *PhantomOpenSSLSocket.h/cpp*
- *PhantomOpenSSLServer.h/cpp*
- *PhantomOpenSSLClient.h/cpp*

Again the latter two implement the first one and add server/client specific logic. An OpenSSL socket is managed by five class variables:

bio local socket descriptor

abio remote socket descriptor stack (server only)

outbio active remote socket descriptor (server only)

ssl current ssl connection (for ssl streams, see 3.4.5)

ctx information about ssl connections (for ssl streams, see 3.4.5)

The API for accepting and connecting is:

```
void connect(std::string host, int port);
void connect(IPAddress* addr);
BIO* accept(int port);
BIO* accept2(int port);
```

OpenSSL accept is a bit tricky. The first time it is used on a socket, it actually creates the socket. Every call hereafter makes the socket listen on the given port. Therefore our API has two different `accept()` methods: `accept()` should be called when accepting for the first time, `accept2()` should be called thereafter (For details see the man page of `BIO_do_accept`).

For sending and receiving we distinguish what to send/receive by calling the corresponding method. Since we will be sending numbers (`int`), byte sequences (`vector<char>`) and arrays of byte sequences (`vector<vector<char>>`) we end up with three different methods for send and receive. We furthermore have different methods for these on the client and on the server (here we also need to specify the remote socket to send to/ receive from `acc_bio`).

```
void send_vec ( std::vector<char>& buf);
void send_vec ( BIO* acc_bio, std::vector<char>& buf);
void send_box ( std::vector<std::vector<char>>& box );
void send_box ( BIO* acc_bio, std::vector<std::vector<char>>& box );
void send      ( int number);
void send      ( BIO* acc_bio, int number);
std::vector<char>* receive_vec ( );
std::vector<char>* receive_vec ( BIO* acc_bio);
std::vector<std::vector<char>>* receive_box( );
std::vector<std::vector<char>>* receive_box( BIO* acc_bio );
int receive();
int receive(BIO* acc_bio);
```

`send_box()` and `receive_box()` actually make use of both of the other methods to send an array of bytes sequence and receive it as such on the remote socket. First the number of sequences is sent, then each sequence is send using `send_vec()`, which again uses `send()` to tell the remote socket the number of bytes and then consecutively sends the bytes.

Algorithm 9 sending data to remote socket

```
#include <openssl/bio.h>
#include <openssl/ssl.h>
void PhantomOpenSSLSocket::send(BIO* remote_bio, int number) {
    // convert to network byte order
    number = htonl(number);
    SSL_GUARD(BIO_write ( remote_bio, &number, sizeof(int)) <= 0);
}
void PhantomOpenSSLSocket::send_vec (BIO* remote_bio, std::vector<char>& buf) {
    //send size of vector
    this->send(remote_bio, buf.size());
    //send buffer itself
    char* wbuf = utility::vector2char(buf);
    SSL_GUARD(BIO_write ( remote_bio, wbuf, buf.size()) <= 0);
    delete[] wbuf;
}
void PhantomOpenSSLSocket::send_box (BIO* remote_bio, std::vector<std::vector<char>>& box) {
    //send size of box
    this->send(remote_bio, box.size());

    //send vectors inside box
    for(int i = 0; i < box.size(); i++)
        this->send_vec(remote_bio, box[i]);
}
```

Algorithm 10 receive data on socket

```
#include <openssl/bio.h>
#include <openssl/ssl.h>
int PhantomOpenSSLSocket::receive(BIO* remote_bio) {
    int* number = new int[sizeof(int)]();
    SSL_GUARD(BIO_read(remote_bio, number, sizeof(int)) <= 0)
    int retval = *number;
    delete[] number;
    return ntohl(retval);
}
std::vector<char>* PhantomOpenSSLSocket::receive_vec ( BIO* remote_bio ) {
    //receive size of buffer
    int read = this->receive(remote_bio);
    //receive buffer itself
    char* rbuf = new char[read]();
    std::vector<char>* victor;
    SSL_GUARD(BIO_read(remote_bio, rbuf, read) <= 0);
    victor = utility::char2vector(rbuf, read);
    delete[] rbuf;
    return victor;
}
std::vector<std::vector<char> >* PhantomOpenSSLSocket::receive_box ( BIO* remote_bio) {
    //first receive the size of our box
    int size = this->receive(remote_bio);
    std::vector<std::vector<char> >* box = new std::vector<std::vector<char> >();
    //now receive the size of each box and the contents
    for(int i = 0; i<size; i++) {
        vector<char>* tmp = this->receive_vec(remote_bio);
        box->push_back(*tmp);
        delete tmp;
    }
    return box;
}
```

Unit Test

There is a unit test testing connecting to and accepting from OpenSSL sockets found in *test/OpenSSLSocketTest.cpp*.

3.4.5 SSL Streams

On top of the just described socket layer we want to establish SSL streams. That means after the client socket has connected to the server socket, an SSL handshake will be made and certificates will be exchanged. Depending on the settings these certificates are then verified and eventually trusted. Thereafter an encrypted SSL stream is established where upon every exchanged message will be encrypted with the other socket's certificate. The logic of this has been implemented in the class *PhantomOpenSSLSocket.h/cpp*, since it is basically another way of establishing connections between sockets.

First of all the socket must be given a certificate private key and a certificate to use for these handshakes. When creating a new socket, the socket will be initialized with the certificate / certificate private key pair found in the config file. The constructors *PhantomOpenSSLServer()* / *PhantomOpenSSLClient()* call *init()*, which calls *init_keys(Certificate* cert, CertificatePrivateKey* key)* setting up a new server/client socket with the certificate key pair. SSL certificate / certificate private key information is stored in the class variable *ctx*. Information about the current SSL session (received certificate, etc.) is stored in the class variable *ssl*. Finally a mode of operation determines the behavior of the node concerning the trusting/verification of certificates (see table below).

Algorithm 11 initialize server socket

```
#include <openssl/bio.h>
#include <openssl/ssl.h>
void PhantomOpenSSLServer::init(Certificate* cert, CertificatePrivateKey* key, PhantomOpenSSLServer::)
    //super class init
    PhantomOpenSSLSocket::init();
    //init ctx
    this->ctx = SSL_CTX_new(SSLv23_server_method());
    SSL_GUARD(ctx == NULL);
    //init certificates and keys
    this->init_keys(cert, key);
    //flag SSL_MODE_AUTO_RETRY allows the server to request and negotiate a new handshake in the background
    SSL_CTX_set_mode(ctx, SSL_MODE_AUTO_RETRY);
    //set verify mode
    this->setVerificationMode(mode);
    SSL_CTX_set_verify_depth(this->ctx, 0);
    //initialize socket with the loaded ssl information
    this->bio = BIO_new_ssl(this->ctx, 0);
    SSL_GUARD(this->bio == NULL);
    //prepare new ssl session
    BIO_get_ssl(this->bio, &this->ssl);
}
```

| Mode | Meaning |
|---------------------------|--|
| SERVER_ONLY | Only the server certificate is verified, the client certificate is trusted |
| SERVER_CLIENT | Both certificates are verified |
| ACCEPT_FIRST_VERIFY_LATER | The server certificate is verified, the client one accepted |
| | but not verified during handshake, instead it can later be verified |
| | calling <code>verifyCertificate(BIO* conn, Certificate* crt2)</code> |

Table 2: verification modes for OpenSSL sockets

The code to initialize the client socket is identical, except for the line:

```
this->ctx = SSL_CTX_new(SSLv23_client_method());
```

According to the design, we will receive the peer nodes certificates in the first round **Setup Package** (see 2.1.1). Thus we need an API for loading and verifying certificates at a later point.

```
void loadCertificate(string filename);
void loadCertificate(Certificate* cert);
bool verifyCertificate();
bool verifyCertificate(BIO* conn);
bool verifyCertificate(BIO* conn, Certificate* crt2);
```

The first two methods load certificates into the socket's trust store (contains trusted certificates). Either of the first two `verifyCertificate()` methods (one for server sockets, one for client sockets) test the received peer certificate against the trust store, to check whether it can be trusted. However following the design (see 2.1) a routing node first accepts a connection from an unknown client, and then receives the client's certificate inside the encrypted setup package. This means we cannot proceed with the standard OpenSSL handshake procedure. Instead we created the mode `ACCEPT_FIRST_VERIFY_LATER` which simply accepts and uses the client certificate, but allows us to access it later, when we will have received the certificate through the setup package. After decrypting the setup package, we can then check the two certificates for being identical using: `verifyCertificate(BIO* conn, Certificate* crt2)`. This is still secure, since the client needs to be in possession of the certificate private key belonging to the certificate it offered during the handshake.

Algorithm 12

```
#include <openssl/bio.h>
#include <openssl/ssl.h>
bool PhantomOpenSSLSocket::verifyCertificate(BIO* conn) {
    SSL* s;
    BIO_get_ssl(conn, &s);

    //verify that we have a connection running
    SSL_GUARD(s == NULL);
    SSL_GUARD(s->session == NULL);
    //get certificate
    X509* x509 = SSL_get_peer_certificate(s);
    SSL_GUARD(x509 == NULL);
    //check certificate
    long certVerifyResults = SSL_get_verify_result(s);
    if(certVerifyResults != X509_V_OK)
        throw SSLError("Error! Certificate could not be verified.\n" + utility::SSLResolveVerifyR
    //free x509
    X509_free(x509);
    return true;
}
```

To verify the the current SSL session uses a valid certificate two checks are necessary: First we need to check that a certificate has been offered by the peer socket calling:

```
X509* x509 = SSL_get_peer_certificate(s);
SSL_GUARD(x509 == NULL);
```

Then we check whether the certificate exists in our trusted store (= part of the ctx struct):

```
long certVerifyResults = SSL_get_verify_result(s);
if(certVerifyResults != X509_V_OK)
    throw SSLError("Error! Certificate could not be verified.\n" + utility::SSLResolveVerifyResult
```

The first check is necessary since `SSL_get_verify_result()` will return `X509_V_OK` when no certificate has been offered.

Note that standard OpenSSL certificate verification involves so called Certificate Authorities (CA) signing certificates in a hierarchical manner. Now in order to verify a certificate this so called **trust chain** can be checked to a point where the signed CA is known to the user. In our case we don't need and don't want such central authorities. We don't need them, because the used certificates are transmitted to the routing nodes in a secure trustworthy manner and we don't want central authorities since Phantom is heavily based on distributed systems not having any central points. Thus we only use **self signed certificates** (trust chain level 0).

Furthermore the design requires a construction certificate generated and used only during the construction process of the routing path. Therefore a methods are provided for generating new certificates and telling the OpenSSL socket to use another certificate for the next session:

```
generateCertificate(int key_length, int days, Certificate* cert, CertificatePrivateKey* key);
void useCertificate(string filename);
void useCertificate(Certificate* cert);
void useCertificatePrivateKey(string filename);
void useCertificatePrivateKey(CertificatePrivateKey* key);
```

Unit Test

There is a unit test testing the establishing of SSL streams and the handling of certificates found in `test/CertificateVerificationTest`.

3.5 Prototype

For the implementation, this work focused on the first part of the design, dealing with the establishment of the **Routing Path**, which has essentially been implemented and illustrated with two binaries (3.5.3). To realize this part two binaries were written simulating the behaviour of an anonymized node building and controlling the routing path, and any number of routing nodes taking part in the path. The other aspects of the design: the **Phantom Network Database** (3.5.1) and **Routing Tunnels** (3.5.2) have been replaced with simplifications copying the result of the corresponding step. This chapter will explain the outline of the prototype coming with this draft and the simplifications of the left out design parts.

3.5.1 Phantom Network Database

Since this work focuses on the implementation of the construction of the routing path, the **Phantom Network Database** (see section 2.3) has not been implemented yet. Yet in order to create a prototype for the construction of the routing path a way to get a number of IP-Addresses, that can be used as routing nodes is required. Thus as replacement of the PND a simple list of possible routing nodes can be used. Since we already implemented a YAML-parser for the config file (see section 3.1.5), the node list is also written in the YAML format.

The structure of the list is **name**, **ip:port**, **public-key**, **certificate** for each node, e.g.

```
- name: faui00a
  ip: 131.188.30.60:2001
  public-key: $basepath/keys/public_key_faui00a.pem
  certificate: $basepath/certs/faui00a.crt
```

Two sample nodelists have been provided in the folder *nodelists/* in the phantom base directory:

- faulist.yaml
- locallist.yaml

The first list *nodelists/faulist.yaml* lists 11 IP-Addresses from the computer science cip-pool at the University of Erlangen and is to be used as remote test.

The second list *nodelists/locallist.yaml* lists 11 times the localhost address 127.0.0.1 with differing ports and can be used as a local test on one machine.

For each node listed in those files a own public-key/private-key pair in the folder *keys/* as well as a certificate/private key pair in the folder *certs/* was created. Further more each listed node has its own config file located in the folder *configs/*.

For the localhost test this config file needs to be specified using the command line option `--config (-c)` when starting a binary (see manual, section 4).

For the remote test the config file can be specified in the same way, or alternatively can be placed in the users home folder on the corresponding machine.

3.5.2 Routing Tunnels

Again since the part of the protocol dealing with **Routing Tunnels** (see section 2.2) is not focus of this work, it has been left out for the first prototype. In its stead, to simulate actual data traffic, the anonymized node simply exchanges data with the terminal node of the routing path, using onion encryption (see glossary on section 11) to do so.

The traffic to be exchanged can be specified with the `--data (-d)` command line option (see the manual, section 4). The data will be sent onion encrypted to the terminal node, which, once decrypted, will send add a data received message at the front of the received data and send it back to the anonymized node. In a real Routing Tunnel stream encryption keys would be used, but for this prototype this has been left out. Instead the connection ids used in the routing path construction are used.

In the *stories/* folder you find a couple of stories in different size and encodings to try out with `--data`.

- the_gifts_of_the_little_people.txt (5K)
- kobutorijiisan.txt (5K, japanese)
- the_laughing_man.txt (31K)
- little_brother.txt (648K)

3.5.3 Routing Path

Two binaries have been created implementing the construction of the routing path discussed in the protocol design in section 2.1.

The first binary named **anonymized_node** simulates an anonymized node that tries to establish a routing path for later communication.

The second binary named **routing_node** listens on a predetermined port for incoming connection requests to be used as a routing node in a routing path.

Anonymized Node This binary simulates the behaviour of an anonymized node, which wants to form an ENTRY- or an EXIT-path to take part in the Phantom Network. For this part of the protocol, it doesn't make a big difference, what function the path will play. The only difference is that the node type of the terminal node will be different, and in case the terminal node is an ENTRY-node the SetupPackage for this node will contain an additional AP-Address and RoutingTableEntry, which could later be used to register the path in the Phantom Network Database, but for now remain without effect.

You can specify the path type with the `--type (-t)` option. Valid parameters are **entry** and **exit**. Apart from the path type, config file, log file, nodelist file, and data to send can be specified. Details about how to use the prototype binaries can be found in chapter 4. In this section we will describe how we implemented the protocol design, presented in section 2.1.

[SETUP]

1. Reading commandline arguments

At the beginning, the commandline arguments are parsed. For a list of possible commandline arguments, see section 4.

2. Setting up anonymized node

The anonymized nodes manages the routing path, but for all other nodes inside this path, the anonymized node is just another routing node of the path. That means it will appear in certain node's setup package as previous or next node together with all the required routing information: connection id, IP-Address and SSL-certificate. Thus we create an **AlphaNode** with these properties. Additionally we will store the anonymized node's public and private key, and the certificate private key, as we will need them later. We find the location of these objects in the config file.

```
AlphaNode* alpha = new AlphaNode(BigNumber node_id, IPAddress ip, PhantomPublicKey public_key, Ph
```

Futhermore a **CryptoMachineRSA** and a **CryptoMachineAES** are initialized, and the **Construction Certificate** for the construction of the routing path is generated.

```
CryptoMachineAES* crypto_aes = new CryptoMachineAES();
crypto_aes->randomInitVector();
CryptoMachineRSA* crypto_rsa = new CryptoMachineRSA();
crypto_rsa->generateCert(1024, 356);
Certificate* construction_cert = crypto_rsa->getCertificate();
```

3. Parsing nodelist

The next step according to the design would be to fetch data about a number of routing nodes from the Phantom Network Database. Since this part is not yet implemented (see section 3.5.1), instead we will parse a yaml-coded list of routing nodes, which we will then put into a vector of routing nodes.

```
vector<PhantomNode*> nodes;
```

4. Creating routing path

Next we will create a routing path out of these nodes. For that a class **RoutingPathOrganizer** was created, taking a list of nodes and returning an ordered sequence of nodes with added node types (**X**, **Y**, **ENTRY**, **EXIT**) in two versions; one version including Y-nodes for the first round and a second version only including X-nodes and one ENTRY/EXIT-node in reverse order for the second round:

```
RoutingPathOrganizer(vector<PhantomNode*>& nodelist, PhantomNode::NodeType path_type);  
void buildPath(vector<PhantomNode*>& nodelist);  
void buildXPath();  
void makePath(PhantomNode::NodeType nt);  
RoutingPath* getRoutingPath();  
RoutingPath* getXNodeRoutingPath();
```

where `RoutingPath` is simply an ordered list of `RoutingNodes`.

Algorithm 13 Building a routing path

```
void RoutingPathOrganizer::buildPath(std::vector<PhantomNode*>& nodelist) {
    this->routing_path = new RoutingPath();
    // if we have n X-nodes, we need 2*n -1 Y-nodes, giving us 3*n -1 nodes in total
    int n = this->getNumberOfXNodes();
    //remember which nodes we already used  bool used[nodelist.size()];
    for(int i = 0; i < nodelist.size(); i++)
        used[i] = false;
    //put n-1 Y-nodes (at least 1) at the beginning (or the end) of the sequence
    bool sequence_begin = RandomGenerator::instance()->random_bool();
    bool xnode = false;
    int i = 0;
    if(sequence_begin) {
        while(i < n-1) {
            int pos = RandomGenerator::instance()->random(0, nodelist.size()-1);
            if(!used[pos]) {
                PhantomNode* node = nodelist[pos];
                node->setNodeType(PhantomNode::Y);
                this->routing_path->push_back((RoutingNode*)node);
                used[pos] = true;
                i++;
            }
        }
        //now that we have some Y-nodes at the beginning of the sequence, we continue with an X-node
        xnode = true;
    }
    //build the 'main path' alternating X-nodes and Y-nodes
    i = 0;
    while(i < 2*n) {
        int pos = RandomGenerator::instance()->random(0, nodelist.size()-1);
        //check that we didn't already add that node
        if(!used[pos]) {
            PhantomNode* node = nodelist[pos];
            if(xnode)
                node->setNodeType(PhantomNode::X);
            else
                node->setNodeType(PhantomNode::Y);
            xnode = !xnode;
            this->routing_path->push_back((RoutingNode*)node);
            used[pos] = true;
            ++i;
        }
    }
    //put n-1 Y-nodes (at least 1) at (the beginning or) the end of the sequence
    i = 0;
    if(!sequence_begin) {
        while(i < n-1) {
            int pos = RandomGenerator::instance()->random(0, nodelist.size()-1);
            if(!used[pos]) {
                PhantomNode* node = nodelist[pos];
                node->setNodeType(PhantomNode::Y);
                this->routing_path->push_back((RoutingNode*)node);
                used[pos] = true;
                i++;
            }
        }
    }
}
```

The start of the routing path is randomly chosen (`bool sequence_begin`) and according to that a number of Y-nodes equal to the total number of X-nodes - 1 is placed at the beginning or the end of the routing path. The rest of the path is filled with randomly chosen nodes from the nodelist, alternating giving them X or Y node type.

`buildXPath()` removes Y-nodes from the path and `makePath()` gives the terminal node the node type `nt` (`EXIT` or `ENTRY`).

5. Preparing first round setup packages

Now that we know the structure of the path, we can prepare setup packages for each participating routing node. Each setup package contains the following routing information:

- NodeType
- Construction Certificate
- RoutingNode `previous_node` (connection ID, IPAddress, Certificate)
- RoutingNode `next_node` (connection ID, IPAddress, Certificate)

Connection ID

Setting the connection IDs was a bit of a problem, considering the chosen layout for the routing information. Given two adjacent nodes X1 - X2. The next node connection ID of X1 must be the same as the previous node connection ID of X2. However given the outline X1 - X2 - X3, considering setup package for node X2: According to the layout described above X2 would be given two RoutingNode structs (previous and next node) containing all information X2 needs to know about X1 and X3. Then however in the setup package for node X1 next node connection ID would be X2's ID, while in X2's setup package the previous node ID would be X1's ID. Another problem with this layout is that the connection ID must be sent on top of the payload, meaning X2 would need to send X2's ID, which it had no information about. The solution was to slightly modify the connection IDs, when creating the setup packages after the following plan:

Outline: α - Y1 - X1 - Y2 - X2 - Y3 - α

| Node | prev_node_id | next_node_id |
|----------|--------------|--------------|
| α | Y3 | α |
| Y1 | α | Y1 |
| X1 | Y1 | X1 |
| Y2 | X1 | Y2 |
| X2 | Y2 | X2 |
| Y3 | X2 | Y3 |

the next node connection ID always corresponds to the ID stored in the current node struct, while the previous node connection ID actually corresponds to the ID stored in the previous node struct.

Table 3: plan for setting connection IDs

We furthermore need to make copies of the nodes in RoutingPath, otherwise we would overwrite the node's ID:

```
next_rn = new RoutingNode>(*next); //calls the copy constructor
next_rn->setID(current_rn->getID());
```

Setup Package Hash Code

There is one more thing we need to put into the setup package: a hash of the contents of the setup package. We compute this hash and add it to the setup package, before serializing and encrypting it. That means in order to verify this hash, the receiving node has to remove the hash from the package, before computing the hash on its own.

```

unsigned char* hash = hash::computeHashSHA256(setup);
setup->setHashCode(hash, hash::getHashSHA256Length());

```

Serialization

After finishing a setup package, we need to serialize it for transfer over the network. For that purpose we have included the library **Google Protobuf** (see section 3.1.2), which allows a simple serialization and deserialization of our setup package:

```

int byte_size = setup->getSize();
char* buffer = new char[byte_size];
setup->SerializeToArray(buffer, byte_size);
std::vector<char>* v = utility::char2vector(buffer, byte_size);

```

Asymmetric Encryption

Next we encrypt the setup package with the public key of the corresponding node:

```

crypto_rsa->loadPublicKey(*current_pukey);
vector<char>* asym = crypto_rsa->encrypt(*v);

```

Symmetric Encryption

At last we encrypt the setup package with the connection ID of the connection the package will be sent on:

```

crypto_aes->loadKey(prev_rn->getID());
vector<char>* sym = crypto_aes->encrypt(*asym);

```

Now the setup package is all set up to be transmitted over an unsecure network. We will add it to the array of setup packages, which we will send around the routing path in the next step:

```

vector<vector<char> >* packages;
packages->push_back(*sym);

```

Before we send the array of setup package, we scramble the order of the contained setup packages, as postulated in the design paper:

```

packages = utility::scramble(packages);

```

Finally we add the following extra information:

- signed hash of the entire setup package array (signed with the construction certificate)
- initialization vector for symmetric de/encryption
- connection ID of the actual next/first node in the routing path starting from α

```

unsigned char* original_hash = hash::computeHashSHA256(packages);
vector<char>* hash_vector = utility::unsigned_char2vector(original_hash, hash::getHashSHA256Length());
vector<char>* signature = crypto_rsa->sign(*hash_vector);
packages->push_back(*hash_vector);
packages->push_back(*signature);
vector<char>* init_vector = crypto_aes->getInitVector();
packages->push_back(*init_vector);
vector<char>* connection_id = utility::string2vector(alpha->getID()->toString());
packages->push_back(*connection_id);

```

6. Sending first round setup packages array

Having everything ready to send it to the first node of the routing path, we create a client socket with our own certificate and certificate private key, establish a connection to this node verifying its certificate (which we got from the PNDB, resp. the nodelist). The modes of operation for SSL sockets are explained in the table in section 3.4.5.

```
PhantomOpenSSLClient* client = new PhantomOpenSSLClient(cert, crt_key, PhantomOpenSSLClient::SERVE);
RoutingNode* first_node = rp->front();
client->loadCertificate(first_node->getCertificate());
client->connect(first_node->ip->getIPWithoutPort(), first_node->ip->getPort());
client->verifyCertificate();
client->send_box(*packages);
```

7. Waiting for first round setup packages array to come back

Now we open another server socket, sit back and wait. If everything works out the last node in the routing path takes us for another routing node, establishes a connection to us and passes on the setup packages array, just as any other routing node will.

```
PhantomOpenSSLServer* server = new PhantomOpenSSLServer(cert, crt_key, PhantomOpenSSLServer::SERVE);
RoutingNode* lastnode = rp->back();
server->loadCertificate(lastnode->getCertificate());
remote_socket = server->accept(alpha->getPort());
vector<vector<char> >* return_packages = server->receive_box(remote_socket);
```

Now we verify the last node's certificate and check for the correct connection ID and IP. Then we check that all the other information set by ourself is the same as we had set in step 5:

- initialization vector
- setup package array hash
- signature of hash

If everything is okay, we assume the first round was successful and continue with preparing the setup packages for the second round.

8. Preparing second round setup packages

The second round setup packages for all Y-nodes are exactly the same as in the first round added a first round success flag, added to all packages of this round:

```
setup->setFirstRoundSuccess();
```

The packages for all X-nodes will have updated information about the next and previous node, now pointing to the next/previous X-node. Also we will have new connection IDs for these:

```
rpo.generateNewXNodeIDs();
x_setup_package = new SetupPackage(x_current_rn->getNodeID(), construction_cert, x_prev_rn, x_n
```

The package for the terminal node (EXIT/ENTRY) will only have information about the previous node, the next node part will be empty. Also in case the terminal node is an ENTRY-node the setup package will contain the AP-Address of this routing path and a ready made RoutingTableEntry for registering with the PNDB.

```

x_setup_package = new SetupPackage(x_current_rn->getNodeType(), construction_cert, x_prev_rn, NULL);
x_setup_package->setAPAddress(ap_address);
x_setup_package->setRoutingTableEntry(rte);

```

Then like in the first round a setup package hash is added, the packages are serialized, encrypted asymmetrically, then symmetrically and finally added to the second round array.

Also like in the first round a signed hash of the entire array, an initialization vector and the connection ID are added on top making the second round array ready to be sent.

9. Sending second round setup packages array

The second round array is sent on the still existing client connection from the first round. This connection can be closed afterwards.

```

client->send_box(*packages);
client->close();

```

10. Waiting for second round setup packages array to come back

The second round array should traverse the routing path like in the first round. Thus we wait on the still existing server connection for the array to return and make the same checks as in step 7. Upon receiving the second round array, we can close the server connection too.

```

vector<vector<char> >* return_packages2 = server->receive_box(remote_socket);
... checks are omitted (see step 7)...
server->close(remote_socket);

```

11. Waiting for connection from next X-node

Now if everything works out the first X-node in the routing path will try to establish a connection to us, after receiving the second round array and after having received an incoming connection from its next X-node as well. We will open a new server socket and wait for this connection, checking certificate and IP.

```

RoutingNode* x_first_node = xrp->front();
server->loadCertificate(x_first_node->getCertificate());
BIO* x_socket = server->accept2(alpha->getPort());
server->verifyCertificate(x_socket);
connection_ia = server->getConnectionIPAddress(x_socket);
connection_ip = connection_ia->getIPWithoutPort();
SECURITY_GUARD(connection_ip != x_first_node->getIP()->getIPWithoutPort());

```

With the connection to the correct node established, the connected node will send the second round array a second time to us. All we need to do is check that the package is the same (except for the connection id, which should be α 's ID, since the first nodes previous connection ID = α 's ID, see table in step 5).

```

vector<vector<char> >* return_packages3 = server->receive_box(x_socket);
vector<char> connection_id3 = return_packages3->back();
BigNumber* id3 = new BigNumber(utility::vector2string(connection_id3));
SECURITY_GUARD(*id3 != *(alpha->getID()));
packages->pop_back();
return_packages3->pop_back();
SECURITY_GUARD(*packages != *return_packages3);

```

12. Onion encrypting data package

This concludes the establishment of the routing path. Now the path would stay ready for connection attempt to an AP-Address (EXIT-path) or for connection requests to this path (ENTRY-path). However since this part of the design (**Routing Tunnels**, see section 2.2) was not implemented yet, we use a simplification and simply send some dummy data, specified via command line options (see the manual, section 4). The target node of this data will be the terminal node of the path, which upon receiving the data will add a “data received message” and send the data back to the anonymized node. Instead of using the stream encryption keys for Routing Tunnels, we will be using the connection IDs as symmetric keys. In order to securely send the data we will **Onion Encrypt** (see glossary, section 11) it, so that only the terminal node will be able to read it.

```
std::vector<char>* data = utility::string2vector(story);
//for each routing node in reverse order//
    crypto_aes->loadKey(next_rn->getID());
    data = crypto_aes->encrypt(*data);
```

13. Sending data package

Now we send this onion encrypted data package to the existing server connection to the first node in the previously established routing path.

```
server->send_vec(x_socket, *data);
```

14. Waiting for data reply package

Finally we wait for the reply package and exit the test upon receiving it

```
vector<char>* reply = server->receive_vec(x_socket);
```

Routing Node A second binary was created to implement the behaviour of a participating routing node. When started it will create a server socket and listen for incoming connection requests. After that it will wait for setup packages and act according to the information found inside. The binary can be started with a couple of command line options like logfile and configfile. Details about how to use the prototype binaries can be found in chapter 4. In this section we will describe how we implemented the protocol design, presented in section 2.1.

[SETUP]

1. Reading commandline arguments

At the beginning, the commandline arguments are parsed. For a list of possible commandline arguments, see section 4.

2. Setting up server socket

First we need to listen for arbitrary connection attempts to our IP-Address. Since we cannot now who is connecting to us at this point, we use a special connection mode (**ACCEPT_FIRST_VERIFY_LATER**), that allows us to accept a connection with an unknown certificate. For a list of socket connection modes, see the table in section 3.4.5. We can verify this certificate however, after we will have received the setup package.

```
PhantomOpenSSLServer* parent_server = new PhantomOpenSSLServer(certificate, certificate_key, PhantomOpenSSLServer::ACCEPT_FIRST_VERIFY_LATER);
```

3. Waiting for incoming connections

Since we want this routing node to participate in more than one routing path, we work with threads. The parent will keep the server socket open and create a new worker thread to deal with new connections. However since a running worker thread, at a certain point in the routing path construction process has to accept a second connection from the next X-node, the parent needs to be able to

- (a) distinguish new connections from connections meant for existing worker threads
- (b) correlate the second type of connection with the corresponding worker thread
- (c) keep data structures safe from synchronization issues
- (d) wake sleeping worker threads when such an expected connection arrives

For the first and second objective a globally accessible `map<Certificate, pthread_cond_t*>` is used. For the third one a mutex for the map `pthread_mutex_t map_mutex` and for the fourth, we use condition variables `pthread_cond_t` for each thread. Now when a thread needs to wait for a connection from the previous X-node it will register a new condition variable together with the certificate of the expected connection to the map. The parent can then, when the certificate of an incoming connection is found in the map, wake the corresponding thread by calling its condition variable:

```
pthread_mutex_lock(&map_mutex);
map<Certificate, pthread_cond_t*>::iterator it = lookup_table.find(*cert);
if(it != lookup_table.end()) {
    pthread_mutex_lock(&socket_mutex);
    new_socket = remote_socket;
    pthread_cond_signal(it->second);
    pthread_mutex_unlock(&socket_mutex);
}
pthread_mutex_unlock(&map_mutex);
```

To make the new connection accessible to the running worker thread, a global variable `new_socket` is used. An additional mutex, `socket_mutex`, is used to avoid synchronization problems on this global variable.

Now for the other case, we will create a new worker thread, that will handle the connection:

```
pthread_mutex_unlock(&conn_mutex);
int ret = pthread_create(thread, NULL, handleConnection, remote_socket);
if(ret != 0)
    throw ThreadException("Error! could not create new thread");
ret = pthread_detach(thread[conn]);
if(ret != 0)
    throw ThreadException("Error! could not detach new thread");
++conn;
pthread_mutex_unlock(&conn_mutex);
```

The global counter `conn` is used to keep track of the number of active worker threads. The number of active connections can be limited by a maximum thread number or a maximum bandwidth usage for phantom.

4. Receiving first round setup packages array

Now we are inside a worker thread having received the connection from parent to start with. So the thread will associate the connection socket with a `PhantomOpenSSLServer` and try to receive the first round setup packages array on this new connection.

```
BIO* socket = reinterpret_cast<BIO*>(arg);
PhantomOpenSSLServer* server = new PhantomOpenSSLServer(PhantomOpenSSLServer::SERVER_CLIENT);
vector<vector<char> >* packages = server->receive_box(socket);
```

Upon receiving an array, it will first take out the connection ID, the initialization vector and the signed hash of the entire array.

```
vector<char> conn_id = packages->back();
connection_id = new BigInteger(utility::vector2string(conn_id));
crypto_aes->loadKey(connection_id);
```

```

packages->pop_back();
vector<char> init_vector = packages->back();
crypto_aes->setInitVector(&init_vector);
packages->pop_back();
signature = packages->back();
packages->pop_back();
sent_hash_vector = packages->back();
unsigned char* sent_hash = utility::vector2unsigned_char(sent_hash_vector);
packages->pop_back();

```

Connection ID and initialization vector is loaded into our `CryptoMachineAES`, which will serve us later to decrypt out setup package. Now that we have taken these four vectors away from the array, we can recompute the hash of the array and see if it matches the sent one:

```

unsigned char* computed_hash = hash::computeHashSHA256(packages);
HASH_CHECK(sent_hash, computed_hash, hash::getHashSHA256Length());

```

For checking the signature, we must first be able to access the construction certificate hidden in the still encrypted setup package. For now we put hash, signature and initialization vector back into the array, since we want to send it on later. We leave the connection ID for the next node out, since that information is also still hidden in the setup package.

```

packages->push_back(sent_hash_vector);
packages->push_back(signature);
packages->push_back(init_vector);

```

5. Decrypting first round setup package

When the thread was called into existence it has also set up a `CryptoMachineRSA` with public and private keys found in the config file.

```

CryptoMachineRSA* crypto_rsa = new CryptoMachineRSA(Config::instance()->getPublicKey(), Config::i

```

Now since the array's setup packages are not ordered, we have to iterate the entire array trying to encrypt every single array position until we find a vector, that we can decrypt when first applying the connection ID and then our private key:

```

for(int p=0; p<packages->size(); p++) {
    try {
        vector<char*> sym_dec = crypto_aes->decrypt((*packages)[p]);
        dec_setup = crypto_rsa->decrypt(*sym_dec);
        delete sym_dec;
        //no exception here means successful decryption
        break;
    } catch (SSLException se) {
        //retry with next package
    }
}

```

After successfully decrypting one of the setup packages we have the serialized version of a setup package meant for us. So we create a new `SetupPackage` and use **Google Protobuf's** ability to deserialize it:

```

char* buffer = utility::vector2char(*dec_setup);
SetupPackage* setup = new SetupPackage();
setup->ParseFromArray(buffer, dec_setup->size());

```

6. Reading first round setup package

The next step is to read the contents of the setup package about the previous and next node, our node type and the construction certificate.

```

PhantomNode::NodeType nt = setup->getNodeType();
Certificate* construction_cert = setup->getConstructionCertificate();
RoutingNode *next_node = (RoutingNode*)(setup->getNextNode());
RoutingNode *prev_node = (RoutingNode*)setup->getPrevNode();

```

Before we trust this information, we first check the hash code of the contents is okay. For that we need to read the hash, then remove it from the setup package, then compute it ourself and finally compare the two to be equal.

```

unsigned char* sent_hash = setup->getHashCode();
setup->removeHashCode();
unsigned char* computed_hash = hash::computeHashSHA256(setup);
HASH_CHECK(sent_hash, computed_hash, hash::getHashSHA256Length());

```

We can then use the construction certificate to check the signature from step 4. We can also use the previous node information to check the currently used certificate, the IP and connection ID.

```

// verify signature
crypto_rsa->loadCertificate(construction_cert);
SSL_GUARD ( crypto_rsa->verify(sent_hash_vector, signature) == 0 );
// verify ID
SECURITY_GUARD(*connection_id != *(prev_node->getID()));
// verify IP
IPAddress* connection_ia = server->getConnectionIPAddress(socket);
string connection_ip = connection_ia->getIPWithoutPort();
SECURITY_GUARD(connection_ip != prev_node->getIP()->getIPWithoutPort());
// verify client's certificate
server->verifyCertificate(socket, prev_node->getCertificate());

```

7. Passing on first round setup packages array

Everything seems to be in order, as far as we can tell. So we will try to establish a connection to the next node, as stated in the setup package, and pass the array on to this node. Before that we have to update the connection ID with the next node connection ID received in the setup package.

```

vector<char*> connection_id = utility::string2vector(next_node->getID()->toString());
packages->push_back(*connection_id);
PhantomOpenSSLClient* client = new PhantomOpenSSLClient(); //uses the certificate/key pair from t
client->loadCertificate(next_node->getCertificate());
client->connect(next_node->getIP()->getIPWithoutPort(), next_node->getIP()->getPort());

```

Since we have all information about the node, we just connected to, we can immediately check its certificate.

```

//verify certificate
client->verifyCertificate();

```

Certificate seems to be fine, we can pass on the array.

```
client->send_box(*packages);
```

This concludes the first round of the routing path construction.

8. Waiting for second round setup package array

With the first round finished, we simply stand back and wait for the second round setup package coming from the already established server socket connection.

```
packages = server->receive_box(socket);
```

Having received the second round array, we take connection ID, initialization vector and signed hash from the array, check them like in the first round and put them back replacing the connection ID with the ID of the next node connection.

```
SECURITY_GUARD(*connection_id != *connection_id2);
SECURITY_GUARD(init_vector != init_vector2);
HASH_CHECK(sent_hash, computed_hash, hash::getHashSHA256Length());
SECURITY_GUARD ( crypto_rsa->verify(sent_hash_vector, signature) == 0 );
packages->push_back(sent_hash_vector);
packages->push_back(signature);
packages->push_back(init_vector);
vector<char>* new_connection_id = utility::string2vector(next_node->getID()->toString());
packages->push_back(*new_connection_id);
```

9. Decrypting second round setup package

Just like in the first round, we will iterate the whole array trying to find the one setup package we can decrypt. Once we could decrypt a package, we will deserialize it into an `SetupPackage` object.

```
... code identical to step 5 ...
```

10. Reading second round setup package

Also like in the first round, having decrypted and deserialized the setup package, we check its hash and read the information inside. Additionally we check that the construction certificate did not change compared to the first round. Also the package should now contain a first round success flag. And finally in case of being an `ENTRY`-node, the setup package should contain an additional `APAddress` and `RoutingTableEntry`.

```
Certificate* construction_cert2 = setup->getConstructionCertificate();
SECURITY_GUARD(*construction_cert != *construction_cert2);
SECURITY_GUARD(!setup->firstRoundSuccess());
if(nt == PhantomNode::ENTRY) {
    ap = setup->getAPAddress();
    rte = setup->getRoutingTableEntry();
}
```

11. Passing on second round setup packages array

Now the next steps depend on which node type we have assigned. We have to distinguish three cases: `Y`, `X` and terminal node (`EXIT/ENTRY`):

(a) Y-Node

In case we are an Y-node we simply send the package on to the still existing client connection to the next node. Then we disconnect all active sockets and end the worker thread.

```
client->send_box(*packages);
client->close();
server->close();
pthread_exit(0);
```

(b) Terminal Node

In case we are a terminal node, we first check that the previous node information of the second round package does not equal the information of the first round, and that the next node information is empty. Then we also forward the array on the still existing connection, and close both active sockets.

```
SECURITY_GUARD(*prev_node->getID() == *prev_node2->getID());
SECURITY_GUARD(*prev_node->getIP() == *prev_node2->getIP());
SECURITY_GUARD(*prev_node->getCertificate() == *prev_node2->getCertificate());
client->send_box(*packages);
client->CLOSE();
server->close();
```

(c) X-Node

i. Waiting for new connection from next X-node

Finally in case of being an X-Node we check that both previous and next node information has been updated.

```
SECURITY_GUARD(*prev_node->getID() == *prev_node2->getID());
SECURITY_GUARD(*prev_node->getIP() == *prev_node2->getIP());
SECURITY_GUARD(*prev_node->getCertificate() == *prev_node2->getCertificate());
SECURITY_GUARD(*next_node->getID() == *next_node2->getID());
SECURITY_GUARD(*next_node->getIP() == *next_node2->getIP());
SECURITY_GUARD(*next_node->getCertificate() == *next_node2->getCertificate());
```

If everything is in order, before we pass on the package we close the existing server socket connection, and wait for receiving a new connection from the next X-Node. However since we are a thread and the connection will first reach parent on the designated port, we have to create a condition variable, register it together with the certificate of the expected incoming connection and wait dormant for it to arrive.

```
pthread_cond_t socket_cond;
pthread_cond_init(&socket_cond, NULL);
lookup_table.insert(std::pair<Certificate, pthread_cond_t*>(*next_node2->getCertificate()), &socket_cond);
pthread_mutex_unlock(&map_mutex);
//wait for connection from next X-Node
server->close();
pthread_mutex_lock(&socket_mutex);
while(new_socket == NULL)
    pthread_cond_wait(&socket_cond, &socket_mutex);
x_socket = new_socket;
new_socket = NULL;
pthread_mutex_unlock(&socket_mutex);
```

The reason this second connection should arrive on the same port as the first, is because many firewalls would block a second port arranged during the first round.

ii. Receiving second round setup packages array again from next X-node

Once the new connection arrived, the worker thread will be woken up by the condition variable. Then it will check certificate and IP and eventually receive the second round setup package again on this connection. We will remove the connection ID that should be different and check the rest for equality.

```

    connection_ia = server->getConnectionIPAddress(x_socket);
    connection_ip = connection_ia->getIPWithoutPort();
    SECURITY_GUARD(connection_ip != next_node2->getIP()->getIPWithoutPort());
    server->verifyCertificate(x_socket, next_node2->getCertificate());
    vector<vector<char> >* packages2 = server->receive_box(x_socket);
    packages2->pop_back();
    packages->pop_back();
    SECURITY_GUARD(*packages != *packages2);

```

iii. Passing on second round setup packages array

Now the X-node will also pass on the second round array package on the still existing client connection and close it afterwards. Now we can also remove the entry in the global map.

```

    client->send_box(*packages);
    client->close();
    pthread_mutex_lock(&map_mutex);
    lookup_table.erase(*(next_node2->getCertificate()));
    pthread_mutex_unlock(&map_mutex);

```

12. Establishing connection to previous X-node

Only X and terminal nodes will reach this point. As such nodes we now must establish the real routing path, by connecting to the previous X-node. Once the connection is established, we verify certificate and IP and then send the second round setup package a second time (with updated ID).

```

    client->loadCertificate(prev_node2->getCertificate());
    client->connect(prev_node2->ip->getIPWithoutPort(), prev_node2->getPort());
    client->verifyCertificate();
    SECURITY_GUARD(connection_ip != prev_node2->getIP()->getIPWithoutPort());
    //adding new connection id
    packages->pop_back();
    new_connection_id = utility::string2vector(prev_node2->getID()->toString());
    packages->push_back(*new_connection_id);
    client->send_box(*packages);

```

This concludes the construction of the routing path on this side. This node is now ready to route data forth and back, de/encrypting it on the way.

13. Receiving Data Package

Since we did not implement the protocol design part about **Routing Tunnels** (see section 2.2), we use the connection IDs instead of stream encryption keys to en/decrypt routed data. We will lay waiting on the established client socket for data being sent. We will then decrypt this data with the corresponding connection ID.

```

    vector<char>* enc_data = client->receive_vec();
    crypto_aes->loadKey(prev_node2->getID());
    vector<char>* dec_data = crypto_aes->decrypt(*enc_data);

```

14. X-Node: Passing on data package

After decrypting one layer from the package, we send it on to the next X-node.

```
server->send_vec(x_socket, *dec_data);
```

15. Terminal-Node: Send reply package

Since we don't have Routing Tunnels yet, we use a simplification where communication is between anonymized node and terminal node (see section 3.5.2). Thus once a terminal node has received a data package, this package should be decrypted completely. The terminal node will add a "data received" message to the data and send it back as reply package.

```
string story = utility::vector2string(*dec_data);  
string reply_str = "R E P L Y   P A C K A G E\n\nThanks for sending:\n\n" + story;  
vector<char>* reply = utility::string2vector(reply_str);  
client->send_vec(*reply);
```

16. X-Node: Receiving and sending reply package

X-Nodes will receive the returned reply package and send it on to the previous X-Node.

```
vector<char>* reply = server->receive_vec(x_socket);  
client->send_vec(*reply);
```

This concludes the prototype with the anonymized node successfully being returned the data it had sent.

4 Manual

In this section we will briefly explain how to setup the environment to test the prototype presented in the previous section.

4.1 Installing Phantom

To install phantom, please check out the latest sources from <https://www4.informatik.uni-erlangen.de:8088/i4sec/phantom/> and run `make` in the base folder. `make` will compile the sources in `src/` and the unit tests in `test/`. Furthermore the config files in `config/` will be adapted to your phantom base path and then be copied to your home folder (if not already existing).

4.2 Folder Structure

Every node needs its own certificate/key and public/private key pair a config file with paths to these files for each node is required. For details about config files, please refer to section 3.3.2.

- A couple of config files has been provided in the `configs/` folder,
- a couple of certificate/key pairs has been provided in the `certs/` folder and
- a couple of public/private keys can be found in the `keys/` folder.

The config file can be specified via the command line option `--config (-c)`. If this option is not used, phantom will assume to find a config file named `.phantom.cfg` in the current users home directory. If you add the computer name to the config file, the parser will take the file fitting the current hostname. For example if you computer's name is `fau00e`, you can place a config file named `.phantom_fau00e.cfg` in your home folder. This is useful for the remote test (see 4.4.2 below), where you will run the routing node binary on many different computers, which share the same home directory.

Apart from these files, which are essential for phantom to run,

- there is a folder `log/`, which is the default location for logfiles. To specify another location, you can use the `--log (-l)` command line options.

- there is a folder `nodelists/` which contains lists of routing nodes. This serves as a simplification for the not yet implemented Phantom Network Database (see also 3.5.1)

- the binaries of the prototype can be found in the folder `bin/`

- the folder `scripts/` contains a couple of scripts that help running the test scenarios (see 4.4 below)

- the folder `stories/` contains some stories that can be used with `--data (-d)` to be sent forth and back the routing path, once it was established.

The source code of the project can be found in the folder `src/` and all the unit tests are located in `test/`.

All the other folders contain the libraries used in the project: `gc`, `glog`, `googletest`, `openssl`, `protobuf`, `yaml-cpp`

4.3 Binaries

To run the prototype, you will need to run these two binaries. The routing node binary will need to be run once for every node in the routing path.

4.3.1 Anonymized Node

Phantom Protocol Command Line Options

| | | |
|--------------------------|-------------------|---|
| <code>--config</code> | <code>(-c)</code> | specify a custom configfile to load |
| <code>--log</code> | <code>(-l)</code> | specify a custom logfile to load |
| <code>--type</code> | <code>(-t)</code> | specify path type (exit or entry) |
| <code>--nodeliste</code> | <code>(-n)</code> | specify a list with phantom nodes (for testing) |
| <code>--data</code> | <code>(-d)</code> | specify a file which's content is to be sent on the routing path (for |
| <code>--help</code> | <code>(-h)</code> | print this help text and exit |

4.3.2 Routing Node

Phantom Protocol Command Line Options

| | | |
|-----------------------|-------------------|-------------------------------------|
| <code>--config</code> | <code>(-c)</code> | specify a custom configfile to load |
| <code>--log</code> | <code>(-l)</code> | specify a custom logfile to load |
| <code>--help</code> | <code>(-h)</code> | print this help text and exit |

4.4 Test Scenarios

4.4.1 Local Test

anonymized_node

To run the local test scenario, you need to specify the *nodelists/locallist.yaml* when starting the *anonymized_node* binary.

You can use the config file *configs/.phantom_local.cfg* or use the config file located in your home folder *.phantom.cfg*. You may choose some data to be sent forth and back at the end of the routing path establishment (see section 3.5.2). A couple of stories, that can be used for that purpose are provided in the *stories/* folder.

```
bin/anonymized_node --config configs/.phantom_local.cfg --nodelist nodelists/locallist.yaml --data stor
```

routing_node

For every routing node in the routing path, you need to start the *routing_node* binary in a separate shell and specify the config file belonging to this node. The routing nodes are named *localnode00a...00k* and the config files are named correspondingly *configs/phantom_localnode00a...00k*. So for every participating node, please run:

```
bin/routing_node --config configs/.phantom_localnode00X.cfg
```

script

You can find a script *local_test.sh* in the *scripts/* folder, which will use *screen* to start all of the localnodes found in *locallist.yaml* in a separate **screen** (see glossary, section 11) session. The script takes as only argument the path to the phantom base directory. Like this you do not have to start them all by yourself. To end the processes, you can use *killall*:

```
scripts/local_test.sh ~/phantom/  
killall routing_node
```

4.4.2 Remote Test

The remote test scenario uses computers in physically different positions connected by LAN. The file *nodelists/faulist.yaml* contains 11 computers of the computer science cip pool. So in order to run this test, you need an account for these computers.

anonymized_node

To run the remote test scenario, you need to specify the *nodelists/faulist.yaml* when starting the *anonymized_node* binary [default].

You can use the config file *configs/.phantom_remote.cfg* or use the config file located in your home folder *.phantom.cfg*.

You may choose some data to be sent forth and back at the end of the routing path establishment (see section 3.5.2). A couple of stories, that can be used for that purpose are provided in the *stories/* folder.

```
bin/anonymized_node --config configs/.phantom_faii.cfg --nodelist nodelists/faulist.yaml --data stor
```

routing_node

You should be able to simply run *bin/routing_node* on any of the computers listed in *faulist.yaml*. The corresponding config file should be located in your home directory. If not you can also specify the config file yourself by using the *--config (-c)* command line option. Config files for faui nodes are *configs/.phantom_faii00a...k*. So for every participating node, please run:

```
bin/routing_node --config configs/.phantom_faiinode00X.cfg
```

script

Again there exists a script that will make it easier to run this test. The script *scripts/remote_test.sh* will also start a detached screen session for every node, use *ssh* to switch to the corresponding computer and then run the *bin/routing_node* binary. To end all the started processes at once, you can use *scripts/end_remote_test.sh*.

```
scripts/remote_test.sh ~/phantom/  
scripts/end_remote_test.sh
```

4.4.3 Unit Test

There is also a binary that will run all the unit tests one after the other and output test results on the shell. The unit tests (see also section 3.3.4) test the base functionality of phantom's environment and components.

```
bin/test
```

[II. Theoretical Part]

5 Testing the Prototype

5.1 Run Time

Although for a thorough test, the Routing Path and Phantom Network Database implementations are missing, we can at least test what we have to get an idea of how long it takes to set up the routing path in ideal conditions, and how long it takes to send data between anonymized node and terminal node.

Both tests are evaluated by run time.

The remote test takes place inside the local area network of the university of erlangen.

The local test takes place on a SUN server.

5.1.1 Remote Test

Scenario 1: 5/2 nodes

| Type | IP | Test Runs | Routing Path Size | Average Time |
|---------------|--------|-----------|-------------------|--------------|
| Routing Path | remote | 200 | 5/2 nodes | 2.05s |
| Data RTT 1K | remote | 200 | 5/2 nodes | 0.08s |
| Data RTT 31K | remote | 200 | 5/2 nodes | 0.04s |
| Data RTT 648K | remote | 200 | 5/2 nodes | 0.6s |

The unintuitive difference between the round trip time required to send 1K compared to 31K, is due to the protocols usage of TCP. TCP connections have the property that small data requires longer because of the TCP security overhead (packages received messages). So that until a certain threshold average transfer speed increases with increasing file size.

Scenario 2: 8/3 nodes

| Type | IP | Test Runs | Routing Path Size | Average Time |
|---------------|--------|-----------|-------------------|--------------|
| Routing Path | remote | 200 | 8/3 nodes | 3.19s |
| Data RTT 1K | remote | 200 | 8/3 nodes | 0.12s |
| Data RTT 31K | remote | 200 | 8/3 nodes | |
| Data RTT 648K | remote | 200 | 8/3 nodes | |

5.1.2 Local Test

Scenario 1: 5/2 nodes

| Type | IP | Test Runs | Routing Path Size | Average Time |
|---------------|-------|-----------|-------------------|--------------|
| Routing Path | local | 200 | 5/2 nodes | |
| Data RTT 1K | local | 200 | 5/2 nodes | |
| Data RTT 31K | local | 200 | 5/2 nodes | |
| Data RTT 648K | local | 200 | 5/2 nodes | |

Scenario 2: 8/3 nodes

| Type | IP | Test Runs | Routing Path Size | Average Time |
|---------------|-------|-----------|-------------------|--------------|
| Routing Path | local | 200 | 8/3 nodes | 3.19s |
| Data RTT 1K | local | 200 | 8/3 nodes | 0.12s |
| Data RTT 31K | local | 200 | 8/3 nodes | |
| Data RTT 648K | local | 200 | 8/3 nodes | |

5.2 What doesn't work yet

- TCP hook
- dummy packages

6 Analysis of Strengths and Weaknesses of the Phantom Protocol

6.1 Design Goals

Some of the following thoughts are taken from the white paper “*Generic, Decentralized, Unstoppable Anonymity: The Phantom Protocol*”, chapter 10 (see [1]).

In the introduction (on page 5) we stated the six design goals of the phantom protocol design. In this section, we will review them in particular respect of the implementation (on page 18).

1. Complete decentralization.

Routing nodes for the Routing Path are randomly selected from the Phantom Network Database. There are no central or favorite routing nodes, in this selection process.

The Phantom Network Database itself is designed as a **Distributed Hash Table** (see Glossary, page 81) and thus decentral in its nature.

Implementation: The PNDB has not been implemented yet, instead a static list of routing nodes is used. The nodes to be used for the Routing Path are randomly selected from this list.

2. Maximum resistance against all kinds of DoS attacks.

The first design goal of decentralization makes DoS attacks harder, which most frequently are targeted against central points of a network. However it is still possible to target as many routing nodes as reachable from one point in the network. Further improvement against this kind of attacks could be the introduction of blacklists in the PNDB, which however would go along with extra overhead and introduce other kind of vulnerabilities.

Implementation: A DoS-Attack on the prototype would have to focus enough of the routing nodes on the list, to make it impossible for the anonymized_node binary to form a routing path. In the config file there is an entry: `upper-bandwidth-limit: 100kbits/s`. If this limit is exceeded the routing_node binary will refuse participation in any more routing paths.

3. Theoretically secure anonymization.

The protocol is designed in a way, that for an attacker to connect an AP-Address with the IP-Address of the anonymized node (i.e. to break the anonymization) would require him to own all the randomly selected nodes of the routing path and the knowledge thereof. Since the anonymized node behaves just like any other routing node, it is impossible to tell its position in the routing path, without knowing that all other nodes are definitely owned by the attacker. Also the use of (signed) hash codes and the double encryption of the setup packages makes it very hard for routing nodes to exchange extra information among the path, without uninvolved nodes or the anonymized nodes noticing. Since however two adjacent nodes always know each other’s IP-Address they can of course open a second covert channel and communicate data over it. However this can only take place between adjacent nodes, so in order to gain benefit from this, again all routing nodes within one routing path would have to be owned by the same attacker.

Implementation: The protocol design was implemented in the above described fashion. However one point has been left out. The protocol design states the creation and insertion of dummy packages inside the setup packages array. This would allow to put the hash of the entire array inside each setup package, instead of outside, like it was implemented in the prototype. The hash of each received array would naturally be different (dependent on the dummy packages, inserted by the previous node), but be secured not only by a signature (like in the prototype), but also by symmetric and asymmetric encryption.

4. Theoretically secure end-to-end transport encryption.

For the construction of the routing path we have tripple encryption of the setup data: outer SSL stream shell, symmetric encryption (connection ID), asymmetric encryption (public key). An outside eave’s dropper would have to break all three layers of encryption. A malicious routing node two layers, a malicious adjacent routing node (has knowledge about connection ID) would still need to break the public key and find out which setup package in the randomly ordered array fits to the connection ID (not easy to tell, as the result of decrypting with the connection ID is just another sequence of still encrypted bytes).

Implementation: The implemented prototype already uses all three layers of encryption: SSL streams (page 37), symmetric encryption (page 30) and asymmetric encryption (page 28).

For the Routing Tunnels we still have the outer layer of SSL streams. Symmetric encryption by connection ID is replaced by stream encryption keys, sent inside setup packages and thus known to the anonymized node. All data will however be onion encrypted (see glossary, page 81) and thus even more secure. Thus an outside eave's dropper would have to break the SSL layer and the onion encryption up to the point he's listening. Since it is really hard to tell the length, even more the current position, in the path, the eave's dropper wouldn't even know how many onion layers, he would have to crack. A malicious routing node would face the exact same problem, taken away the SSL layer.

Implementation: Routing Tunnels have not been implemented yet. Instead as a simplification data will be sent back and forth between anonymized node and terminal node of the routing path and onion encrypted with the connection IDs instead of stream encryption keys.

5. Complete (virtual) isolation from the "normal" Internet.

Within the phantom networks IP-Addresses as a mean of end-to-end communication have been replaced by AP-Addresses. Thus a node inside the phantom network can only communicate to other nodes inside the network. In the same fashion no outside node (which has no phantom client installed locally) can communicate with a node inside the network. This means a popular search site can only be reached from a phantom node, if the search site's hosting server itself has joined the network, and thus owns a registered AP-Address within the Phantom Network Database. For the first introduction of the phantom protocol gateway servers could be set up, which would allow access to the normal internet, until the protocol has established itself on the market. However this must be a temporary solution only, since it undermines many of the here stated design goals, including this and the decentralized approach to routing.

Implementation: For the Routing Path construction AP-Addresses are not involved.

6. Maximum protection against identification of protocol usage through traffic analysis.

All communication during the Routing Path construction and later through Routing Tunnels is made through SSL connections and streams. That makes protocol identification very hard for outside listeners, as they would first have to break the SSL layer. However there is so much SSL traffic going on, that makes it even more difficult to even target a potential phantom SSL stream. Furthermore the design paper suggests using the standard web server port (tcp/443) or some comparable port, to also avoid identification by usage of a certain port.

Advanced traffic analysis however (using several listening points, comparing time stamps etc.) could probably be successful given a lot of effort and resources. Still it will be very difficult and probably yield a lot of false positives to be practically or commercially feasible.

Implementation: In the prototype SSL streams and SSL sockets have been implemented and used. That means already here protocol identification attempts would have to deal with the outer SSL layer. The current prototype however does not yet use port 443.

7. Capability of handling larger data volumes, with acceptable throughput.

Due to the protocol design it is near to impossible to find out the length of a Routing Path. This however means that even if two nodes within the separated Phantom Network would communicate directly with each other, they are so to say their own routing paths, it would still be secure to a great extent (at least what reasonable doubt is concerned). Also the user can choose its own security preferences and thus influence the length of his Routing Path. A user with low security requirements and fast communication requirement of large data volumes can set the Routing Path to a very small length, while a endangered user (e.g. a reporter) could choose a higher security settings with slower round tript time.

Implementation: This is realized by an option in the config file, e.g. `security-level: MEDIUM` (page 24 for a list of possible settings). This value determines the length of the Routing Path, still including a random element. For example `MEDIUM` results in a Routing Path with 2 or 3 routing nodes.

8. Generic and well-abstracted design, compatible with all new and existing network enabled software.

The protocol simply replaces TCP/IP-routing in a secure and anonymous way. That means all existing network software using TCP/IP can use the Phantom Protocol, too. This can be realized for example by simple binary hooks. The advantage here is that with the phantom protocol being below the application level, applications itself do not to be changed in order to use the isolated phantom network.

Futhermore the three parts of the design: Routing Path, Routing Tunnels and Phantom Network Database are completely independent of each other and can be replaced without affecting the other parts, which allows for separate development and improvement.

Implementation: The construction of the Routing Path has been implemented in the prototype presented in this draft. The other parts could indeed be easily replaced by simplifications to illustrate the protocols functionality and capabilities. The prototype does not yet create binary hooks or does in any way tunnel application level internet traffic.

6.2 Weaknesses

Some of the following thoughts are taken from the white paper “*Generic, Decentralized, Unstoppable Anonymity: The Phantom Protocol*”, chapter 11 ([1]).

Total Control Scenario

As mentioned in the previous section, one risk is one attacker being able to control all of a routing path’s nodes and knowing so. In this scenario the attacker can synchronize the information of all individual routing nodes. In this case the attacker will be able to:

- know to which AP-Address the anonymized node is connecting to (EXIT-path)
- be able to correlate AP-Address of the ENTRY-path with the IP-Address of the anonymized node (ENTRY-path)

However the attack would still not be able to influence the transferred data, because of the end-to-end encryption.

It should be mentioned, that even if controlling all routing nodes of the routing path, it is really difficult to gain the knowledge that one does own all the routing nodes. This is because the nodes adjacent to the anonymized node cannot distinguish it from any other routing node in the path.

Total Correlation Scenario

Another attack scenario is of an attacker being able to monitor all traffic of an entire routing tunnel. The attacker would thus be able to correlate that the same amount of data is being sent forwards or backwards through the routing tunnel and consequently be able to locate the two end-points of the tunnel.

Counter measures to this include random delays when routing data or the opening of additional connections. However this would slow down the protocol, which would contradict design goal #7 (see design goals on page 5). Another counter measure that would make traffic analysis significantly harder, without slowing down the protocol too much, would be the inclusion and deletion of a random amount of junk data.

Covert Channels (inside)

Routing nodes within a routing path know the IP-Addresses of the nodes adjacent to them. Thus they could open up a second connection to the adjacent node and communicate out-of-protocol data. On such a channel they could send extra data, like: time stamps, package size, their own symmetric key, etc.

Still they would not be able to alter the data itself, due to end-to-end encryption. Also they could never be certain that both adjacent nodes take part in the covert operation. Even worse, since no routing node does now if one of its adjacent nodes is the anonymized node, trying to open a secondary covert connection channel to the adjacent node would end in the anonymized node noticing the covert operation, terminating the connection immediately (Not included in the prototype). Also the protocol design further limits this possibility by the first round of the routing path construction (see page 7), buffering later to be adjacent nodes by a temporary intermediate Y-node.

Covert Channels (outside)

A slightly different scenario is that of one or more routing node giving all its information on a covert channel to an outside observer, who can benefit by it gathering information about the sent data (prev-next node IPs, IDs (symmetric keys), certificates), plus additional data like time stamps and package size. The more such data an observer can gather the easier traffic analysis becomes.

Known Routing Nodes

Another problem is that even though protocol identification is made really difficult (see design goal #3 on page 5), the IP-Addresses of computers participating in the Phantom Network as routing nodes can be easily found out by anyone joining the Phantom Network querying the Phantom Network Database. In some countries even the participation of computers as routers of data could be prohibited, so this is a real problem. A solution would be to allow users anonymization without having to join the network as routing nodes. Also users in a totalitarian country could use phantom using only nodes outside the country as routing nodes (e.g. by using GeoIP). However the choice whether to join the network only as “client” (anonymization) or also as “server” (routing node) brings us to the next problem, the client dilemma.

The Client Dilemma

Like other peer to peer system Phantom lives and dies with people joining the network as peers, which take the system’s services, while giving the system something back. Like for example in file sharing networks you would not only download data, but also offer to upload data for others.

In Phantom the distinction is whether to only use Phantom for anonymization or to also offer to be a routing node for others. Since offering to be a routing node is a potential risk (any node within the network can query your IP-Address from the PNDB, see previous problem), many users will decide only to use the networks anonymization service, but not offer the same service to the other nodes. We call this the client dilemma.

Forcing users to also offer routing services is not feasible for totalitarian countries (see previous problem) and could severely hinder the acceptance of Phantom among the population. On the other hand, if nobody or very few nodes can be used for routing, the network will be slow and weak against DoS attacks.

Isolation from the normal Internet

The isolation from the normal IP based Internet is both a strength and a weakness. It means at first every normal web page will be unavailable to someone using Phantom. Of course the user will be able to switch Phantom on and off, even a script can be provided that will automatically switch to standard IP based internet routing (maybe prompting a warning, when doing so), but still this effectively limits the usability of Phantom.

If the Phantom Protocol gets more popular, more and more webserver and webservices will start to support it, but until then this is a severe problem. A startup solution might be gateways within the Phantom Network to the normal Internet, however this would undermine many of the design goals of Phantom, like the decentralization and resistance against DoS attacks.

6.3 Strengths

- user friendly (install and go)
- transparent

7 Comparison of Phantom with other anonymization approaches

There are several more approaches to anonymization of network traffic than Phantom. Though their purpose is too far extent identical, the way they try to keep the communicating parties anonymous can be very different. In this chapter we will look at three more ways to provide anonymization:

- a central controlled approach with TOR
- a approach mixing routed traffic with JAP
- a peer 2 peer approach with I2P

We will start with the most popular and most establish of these approaches: the TOR Network.

7.1 TOR

Tor is a network of virtual tunnels that allows people and groups to improve their privacy and security on the Internet. It also enables software developers to create new communication tools with built-in privacy features. Tor provides the foundation for a range of applications that allow organizations and individuals to share information over public networks without compromising their privacy.

Individuals use Tor to keep websites from tracking them and their family members, or to connect to news sites, instant messaging services, or the like when these are blocked by their local Internet providers. Tor's hidden services let users publish web sites and other services without needing to reveal the location of the site. Individuals also use Tor for socially sensitive communication: chat rooms and web forums for rape and abuse survivors, or people with illnesses.

Journalists use Tor to communicate more safely with whistleblowers and dissidents. Non-governmental organizations (NGOs) use Tor to allow their workers to connect to their home website while they're in a foreign country, without notifying everybody nearby that they're working with that organization.

Groups such as Indymedia recommend Tor for safeguarding their members' online privacy and security. Activist groups like the Electronic Frontier Foundation (EFF) recommend Tor as a mechanism for maintaining civil liberties online. Corporations use Tor as a safe way to conduct competitive analysis, and to protect sensitive procurement patterns from eavesdroppers. They also use it to replace traditional VPNs, which reveal the exact amount and timing of communication. Which locations have employees working late? Which locations have employees consulting job-hunting websites? Which research divisions are communicating with the company's patent lawyers?

A branch of the U.S. Navy uses Tor for open source intelligence gathering, and one of its teams used Tor while deployed in the Middle East recently. Law enforcement uses Tor for visiting or surveilling web sites without leaving government IP addresses in their web logs, and for security during sting operations.

The variety of people who use Tor is actually part of what makes it so secure. Tor hides you among the other users on the network, so the more populous and diverse the user base for Tor is, the more your anonymity will be protected. [?]

7.1.1 Introduction

The acronym TOR stands for The Onion Router, and to understand TOR, we first have to understand Onion Routing.

Onion Routing

Modern warfare requires fast anonymous encrypted data transfer between any two points. Otherwise either the data could be read or the locations of the troops could be revealed. There were solution for anonymous encrypted data transfer, usually called remailer protocol (see Glossary on page 81), but they are slow and usually took minutes to several hours for a transfer to complete. Thus the US Navy (Office of Naval Research) financed a program to develop some faster technology to meet these ends. This program was called Onion Routing. In the year 1996 David M. Goldschlag, Michael G. Reed and Paul F. Syverson presented Onion Routing for the first time to the public. Four years later at a conference called "Workshop on Design Issues in Anonymity and Unobservability" Paul Syverson and Roger Dingledine met and decided to develop the second generation of Onion Routing. Their project was later called TOR and was financed by DARPA⁵. [Kubieziel2007]

⁵Defense Advanced Research Projects Agency

The principle of Onion Routing is that a data-package will be imposed with several layers of encryption (onion layers). The number of layers corresponds to the number of hops. Each routing server will then remove one layer of encryption with a pre arranged key.

TOR

The goal of TOR is to anonymize applications with low latency replies. To establish connections a proxy using the SOCKS protocol (see Glossary on page 81) is used. This proxy can establish a connections into the TOR network through the firewall of the anonymized computer. The TOR network went online in the year 2003 and its developers decided to publish the code under a free licence (MIT-licence) opening doors for broad community development. When the financing of DARPA run out 2004, the Electronic Frontier Foundation (EFF) continued the financing for one year. After that the project became independent and financed by donations. In the year 2007 estimated TOR user numbers were around 200.000 with traffic of more than 100 MB per second. [Kubieziel2007]

How TOR works

The anonymized computer wants to connect to a webpage. The browser on the anonymized computer is set to use the SOCKS-proxy (TOR client) mentioned before. This proxy chooses a sequence of Onion Routers and connects with the first in the sequence. Then they will do a Diffie-Hellman Key Exchange (see Glossary, page 81) to exchange a symmetric key. Then the anonymized computer connects to the second Onion Router using the first one, also negotiating a symmetric key for this router. This continues until a key for the last router in the sequence was negotiated. All packages, like the HTTP request for the webpage are then onion encrypted (see Glossary, page 81) with the symmetric keys of the chosen onion routers. With every router decrypting one layer of encryption, the last router, also called exit node, ends up with the unencrypted package. It can now perform the HTTP request on standard fashion. The reply is encrypted using the symmetric key and then passed on to the previous Onion Router. The previous router receives the package, encrypts it with his symmetric key and passes it on until it reaches the anonymized computer, which, being in possession of all symmetric keys, can decrypt the reply and display it on the browser.

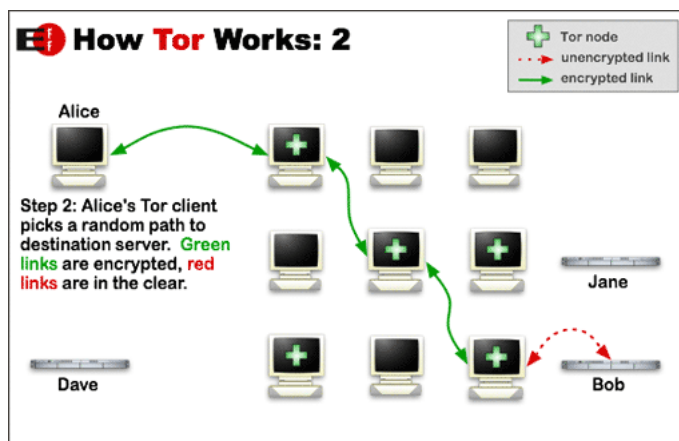


Figure 21: How TOR works (from [?])

Every OnionProxy always establishes more than one connections sequences and decides for every new connection for one of these sequences. Also after 10 minutes an existing sequence is closed and a new one established, removing all connection information of the previous sequence.

It is important that public key cryptography in TOR is only used to negotiate symmetric session keys. This results in a tremendous performance benefit as public key cryptography is very time intensive. [Kubieziel2007][?]

Hidden Services

There is also a service to anonymize a server offering resources called **hidden services**. The anonymized server will create a long lasting key pair and open a connection sequence to a couple of TOR servers called **introduction point**. It will pass on public key, key length and time stamp to these servers. The key is used to sign messages (called **service descriptor**). If an introduction point accepts such a service descriptor a permanent connection to

it is established. When an anonymized client wants to access the service, it can look up the descriptor in a central directory listing one of the introduction servers. The introduction server is then used to negotiate another TOR node as **rendezvous point**. Both client and server establish an anonymous connection to the rendezvous point and use it to do a Diffie-Helman-Key-Exchange to exchange a symmetric session key. Now the two computers can communicate anonymously and with end-to-end encryption. To resolve anonymous services TOR uses a DNS top level domain called onion, which official DNS server can understand. To avoid treacherous names checksums are used instead of plaintext names. [Kubieziel2007][?]

7.1.2 Comparison

Compatibility

Phantom replaces and simulates the TCP protocol. Thus all applications being able to use TCP can also use Phantom. No proxy is and no special settings for the applications are required. Like explained above TOR bases its functionality on the usage of a SOCKS proxy. [1]

Usability

Phantom is even more user friendly than TOR, since no special settings need to be made in network applications and no proxy need to be set up. A simple installaion of the binaries will suffice.

Throughput

Phantom security is bases on reasonable doubt. With no certain knowledge of where the anonymized node is located, short or even direct connections are possible maintaining a reasonable level of security. With shorter connections Phantom is able to provide high throughput. If however high throughput is no criterium for the user, longer paths can be chosen for even more security. [1]

Data Size Limit

TOR limits the size routed data may have. At the current point Phantom has no such limit. [1]

End-To-End Encryption

Phantom provides secure end-to-end encryption securley preventing any data sniffing at any point of the routing. TOR does provide encryption between the anonymized node and the exit node (also called out proxy), but not between exit node and target server. An attack at this unencrypted part of the routing is called **out-proxy-sniffing**. Such attacks are not possible with the Phantom Network.[?]

Isolation From The Internet

The Phantom Network is an isolated part of the normal IP based Internet. That also means webservices which do not support the Phantom Protocol are not reachable from within the Phantom Network. TOR is designed to make the normal Internet accessible in an anonymized way.

However nobody prevents Phantom participants from opening Gateway nodes, which will establish connections to the normal Internet just like TOR's exit nodes.[1]

Transparency

Since Phantom replaces the computer's network communication logic, it is impossible for any application to establish their own connection not using Phantom sockets. TOR met this problems with e.g. flash applications connecting outside of TOR's proxy logic. However this does not protect from malicious applications actually revealing the computers's IP address on the orderly established conection. This is a difficult problem. Even when setting up a watchdog parsing the outgoing network traffic for some programm sending the computer's IP address, it is always possible to trick such a parser by using a non obvious way to convey the IP address (i.e. encryption).

Blockade

Instead of traffic analysis, a government can simply decide to block any anonymization service. In the case of TOR this means block the central directory servers, which list all the TOR nodes. In 2007 there were five such servers, making it fairly easy to block them. With Phantom using a distributed hash table instead of central directory servers, blocking access to the Phantom Network Database is made significantly harder.

Furthermore the government could also access the directory servers and also block all the listed TOR nodes instead. This can also not be prevented in Phantom, but with every Phantom user in theory being also a Phantom node, this again is made significantly harder, since the IP addresses of currently available Phantom nodes will change multiple times every second making it again significantly difficult to block all of them. Also with dynamically assigned IP-addresses the number of falsely blocked services will reach a level, where the damage will exceed the gain.

The TOR developers also work on this problem and are developing small hidden **bridge nodes**, which like with Phantom, consist of standard DSL users helping users in TOR blocking countries to access the network.[Kubieziel2007]

Trap Doors for Users

Neither TOR nor Phantom can prevent users from certain mistakes the user has to take responsibility for. For example no anonymization service is any use, if the user states her or his real name and address at some public accessible point.

Even more dangerous is the habit of users blindly accepting unverified certificates from web services. This opens doors for man-in-the-middle attacks. Instead of talking to the desired service, the user is really talking to a in between computer pretending to be that service. The user will notice this by getting a warning of invalid certificate, but if she or he ignores that warning, there is not much the software can do.

A third common trap door are cookies. Normal internet users have set their browser to accept all kind of cookies without asking. This again open doors for attackers using this cookies to store data on the users computer that will help to identify her or him.[?]

7.1.3 Attack Scenarios

For every anonymization approach exist various different attacks. This draft does not attempt to list and discuss them all, instead a selection of interesting attacks against the approach was made, and then compared to how successful they would be with Phantom.

DNS Leak Attack

DNS resolution requests are usually performed by many applications without using a tor proxy. TOR now offers a special DNS top level domain called onion, which should prevent such leaks. DNS leaks should be impossible for Phantom, because at the current design DNS is not used. It is likely to be included in a later design to provide accessing webservice by DNS name, but even then the DNS request will be tunneled by Phantom's network logic and probably be resolved by the Phantom Network Database, instead of standard central DNS servers.[?]

Fingerprint Attack

The fingerprint attack was described by Andrew Hintz. It is based on the fact that most webservices transfer exactly the same amount of data to someone requesting their service. An attacker can now build a database of webservice fingerprints mapping data size (and form) to certain webservices. Now the attacker can sniff at certain routing points and could end up with the knowledge which computer accessed which websites judging only from the size and form of the transferred data.

In Phantom this attack is weakened by the fact that the position of the anonymized computer is unknown, thus the attacker can only conclude which webservice's data is routed, but not to where.[Kubieziel2007]

DoS Attack

As stated in section 6.1 on page 60 Phantom is very resilient against DoS attacks (see Glossary, page on page 81), because of its distributed nature.

TOR however, using central directory servers, is vulnerable against this kind of attack. Also the TOR nodes are also limited, as the setup of a TOR server is significantly harder than joining the TOR network as a client. Thus

even apart from the obvious vulnerability of the directory servers, the relative amount of TOR nodes and their static nature makes them also target for DoS attacks.

Central Resource Attack

In this scenario the attacker gets access to one of TOR's directory servers and changes the listed TOR nodes to nodes the attacker controls. Therefore the attacker knows that someone getting their list of TOR nodes from his directory server will only use undermined TOR nodes. To prevent this TOR's developers only set up a few directory servers and given control over them only to a few selected trusted people. But even if the people can be trusted, an attacker could still gain access in another way.

Again Phantom does not use central directory server to list Phantom nodes, but the distributed Phantom Network Database. However this means any attacker can set up a PNDB node and simply offer his own part of the database, which lists only undermined Phantom nodes. To minimize the risk an anonymized node always gets lists of Phantom nodes from several PNDB nodes and then uses trust algorithms, like **EigenTrust** (see Glossray, page on page 81), to select the most trustworthy set of nodes. Thus again in order to convey false information into the PNDB an attacker would have to control a significant amount of the database itself, which is very difficult.[Kubieziel2007]

Traffic Analysis Attack

Phantom is more resistant against traffic analysis and traffic identification, because it uses SSL connection between any to routing nodes in the network. This does not prevent traffic analysis entirely, but makes it significantly harder, since the risk of false positives raises. TOR on the other hand is very vulnerable to traffic analysis. Steven J. Murdoch and George Danezis from University of Cambridge presented an article⁶ at the 2005 IEEE Symposium on Security and Privacy on traffic-analysis techniques that allow adversaries with only a partial view of the network to infer which nodes are being used to relay the anonymous streams. These techniques greatly reduce the anonymity provided by Tor.[?]

Hidden Services Attack

Overlier and Syverson found presented an attack on hidden services, where the attacker was onion router and onion proxy at the same time. The attacker would try to connect to the hidden service for as long as was needed, until he himself was part of the route to the rendezvous point. If he ends up being adjacent to the rendezvous point, he disconnects the connection and starts anew. Otherwise he measures answer times to estimate his position in the routing sequence. If the answer time is short it assumes its adjacent node to be the hidden service provider.

Because of such kinds of attack Phantom has made the process of service/server encryption absolute symmetric to the process of client encryption. An attacker could never tell whether he is part of the service he wants to connect to or not. Nor could he securely estimate his position or the position of the anonymized node in the path.

Concerning TOR counter measures include the introduction of **entry guards**, which will take the role of being the first onion router in the path of the hidden service for a while, making attacks significantly harder. Also the rising number of TOR nodes reduces the chance of an attacker's onion router being used in such a path significantly. [Kubieziel2007, OverlierSyverson2006]

Geological Attack

Steven J. Murdoch presented a completely new and surprising attack in 2006 and improved it together with Sebastian Zander in 2008, which is based on geological conditions to locate a hidden service provider. A computer's clock reacts both to inside and outside influences by tiny shifts in time. This means continuously requesting large files from a hidden service will heat up the temperature inside the providing computer, slightly changing the system clock. Now stopping these requests will result in a cooling down of the providing computers temperature, also affecting the clock. Combining this with knowledge of current geological conditions around the world and their influences on the system clock behaviour allows to geographically locate the providing computer.

Phantom is in the same fashion also vulnerable to this kind of attack. [Murdoch2006, Murdoch2008]

⁶"Low-Cost Traffic Analysis of Tor" (PDF). 2006-01-19. <http://www.cl.cam.ac.uk/users/sjm217/papers/oakland05torta.pdf>. Retrieved 2007-05-21.

7.2 JAP

Java Anon Proxy, also known as Java Anonymous Proxy, JAP Anon Proxy, JAP or JonDonym, is a proxy system designed to allow browsing the Web with revocable pseudonymity. It is based in Germany and was originally developed as part of an ongoing project of the Technische Universität Dresden, the Universität Regensburg and Privacy Commissioner of Schleswig-Holstein. Written in the Java programming language.

Cross-platform, free, it sends requests through a cascade and mixes the data streams of multiple users in order to further obfuscate the data to outsiders.

JonDonym is available for all platforms that support Java.[JAP/Wiki]

7.2.1 Introduction

The project started with the acronym An.On, Anonymity Online, as a cooperation project between the data privacy center of Schleswig-Holstein and the university of Dresden in 2000/2001. Later the university of Regensburg also joined the project. The software to use the service was provided to the public under the acronym JAP, Java An.On Proxy or Java Anonymous Proxy. Jap is based on the works of professor Andreas Pfitzmann, who proposed the concept of Mix networks in 1990. His concept was based on so called **Mix Cascades**, a sequence of network routers (called **Mixes**), who would take encrypted messages from many different users, mix resp. mux them into one encrypted stream and pass them on, on a predetermined path. Those Mixes should be, as far as possible, operated by different organizations and in different countries.

In 2007 the public financing for the project has run out, and it was taken over by the JonDo's limited cooperation and commercialized under the label **JonDo**. The company takes the role as a mediator between network users and Mix hosts, taking money from the users, from which Mix hosts are paid.

Since the code is open source, parallel to the paid anonymization service, a free one was established distributing the software under the old name JAP, where mixes are gratuitously operated by organizations like the Chaos Computer Club, FoeBud or the Pirate Party. [Pfitzmann1991, Kubieziel2007, JAP/Wiki]

How JAP works

JAP's approach to anonymity consists of the JAP client program, mixing cascades and an InfoService. The client program acts as a HTTP(s)/FTP proxy to outgoing internet traffic (note JAP does not use a SOCKS-proxy like TOR). The client program first connects to the so called **InfoService**, which checks the user's JAP version is not outdated. If it is an automatic update is offered. The user then chooses a mixing cascade, being shown the current number of users using this cascade and an estimated ratio of anonymity. The client program registers with the first mix in this cascade, and all is set up for network traffic.

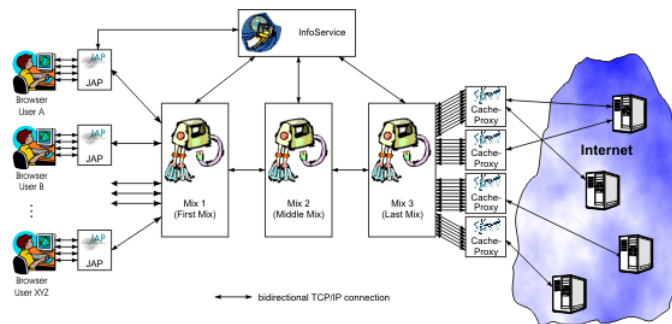


Figure 22: How JAP works (from [JAP])

When a user attempts to connect to a website, JAP receives this connection request and creates a symmetric key for it. Then a package is created containing this key, the header data, the connection request and some random data. The first 128 bytes of the message are encrypted with the public key of the last mix in the sequence, the rest is encrypted with the generated symmetric key. Now a new symmetric key for the last but one mix is generated and the procedure is repeated until all mixes in the sequence are processed. Before the package is sent to the first mix in the sequence, a random channel ID is generated and saved together with the generated symmetric keys.

The first mix will decrypt the first 128 bytes of the message with its private key, thus learning the symmetric key to decrypt the rest of the message. It will now generate a new channel ID for the net mix and save both channel IDs and the symmetric key locally. This procedure is repeated by every mix until the package reaches the end of the mixing cascade.

The last mix can completely decrypt the message and learns the original connection request, which he sends to the target website through a cache proxy. It will connect to the website, receive a reply and then generate two messages: the reply from the website and random data together with the information that the connection will be terminated. The mix encrypts both messages with the symmetric key building another mix package. This package is sent backwards through the mixing cascade to the original sender with each intermediate mix checking channel ID and choosing the corresponding symmetric key and encrypting the routed package with it. The last mix in the cascade passes the encrypted package on to the original sender. JAP will then terminate the connection to the browser.[JAP, Kubieziel2007]

Mixing Cascades

All mixes within one cascade will not immediately pass on packages, but instead wait for an arbitrary delay and then pass on all saved packages in a random order to the next mix in the cascade. This makes it harder to keep track of certain packages. Mixing cascades are fixed sequences of mixes. The user can choose to use a different cascade using the JAP software, but if he doesn't, he will remain with the same cascade. This fixed and managed routes are a main difference to other anonymization approaches like TOR, I2P or Phantom.

Furthermore mixes in one cascade are steadily sending random garbage messages to each other to avoid any eave's dropper from correlating more information from higher traffic. With these dummy message the traffic stays almost constant. However this is also a big hindrance for setting up a mix. This procedure requires very high data throughput with at least 100 MB per second. [Kubieziel2007]

7.2.2 Comparison

Security

JAP's security lies in its design of an anonymization service structure, that prevents any single party of gaining too much influence over the service. The JohDo company does not host mixes itself, but only manages mixes of other parties and organizes them into mixing cascades. With this, it is assured that no mixing cascades contains more than one mix of the same proprietor, which makes it significantly harder for an attacker to control more than one mix. On the otherhand the design allows for an attacker controlling one mix inside a mixing cascade without harm to the anonymity.

This approach is entirely different to Phantom's, which drives its security out of its distributed uncontrolled nature. Thus in the Phantom Network, *it is possible* for an attacker to control more than one Phantom Node, but as described on page on page 62, the design accounts for that, and only an attacker controlling the entire network and having the knowledge of doing so, can effectively break Phantom's security.[JAP/Wiki]

Compatibility

As previously stated, Phantom's new approach is to replace the network logic of the computer entirely, instead of using a proxy server. This makes Phantom basically compatible to all network software using TCP. JAP on the other hand uses a standard proxy server, which only understands HTTP(s) and FTP protocols.[JAP/Wiki]

Usability

The website of the An.On project states:

This installation option integrates the JonDoFox profile and a locally installed Firefox, so that JonDoFox operates seamlessly with your system. Any previous Firefox configuration you may have created remains untouched by the installation. After installing, you may choose between both your profile and the JonDoFox configuration at each start of Firefox.

So it is fairly easy to set up, however it requires a compatible browser, like firefox and will only work for internet traffic using the HTTP(s) and FTP protocols.

Phantom is just as easy to set up, but is independent to the application level.

Isolation

Same as TOR, JAP is created to offer anonymous access to the standard IP-based Internet, while Phantom creates its own isolated darknet.

Access to the normal internet always brings the weakspot of missing end-to-end encryption, since the last part of the path between exit mix and webservice will be unencrypted (except for https).

Entry/Exit Point

JAP's fixed structure of mixing cascades with predetermined entry and exit points brings another weakspot. While the traffic between mixes seems quite secure, the entry and exit points are weakspots. The missing end-to-end encryption was already discussed in the previous point. Here I want to pinpoint the fact that with fixed entry/exit addresses, incoming and outgoing traffic can be monitored. This will reveal the IP-Addresses of computers participating in the JAP Network, which can by itself be a problem in restrictive countries.

While Phantom has a similar problem with IP-addresses of possible routing nodes listed in the Phantom Network Database, the participation in the network both as client and server does sufficiently hide its own IP address.

Restrictive Governments

With JAP's managed structure and JohnDo as company behind the service, the JAP Network becomes vulnerable to governmental restrictions. Wikipedia tells the story of the German government forcing JAP developers to include a recording function, to record users visiting certain webpages. This function was forced upon them by a temporary mandate. The recording function was included, but could later be deactivated, when the JAP providers went to the court.

In January 2009 the EU wide enforcement of telecommunications data retention has come into force. This forces all mixes located in Europe to: store IP-Address, channel ID, date and time of all incoming connection (first mix), incoming and outgoing channel ID with date and time of the corresponding connection (intermediate mixes) and incoming channel ID, date and time, target port number with date and time of the request (last mix). This allows EU governments to effectively break JAP's anonymization service for mixing cascades, whose mixes are all located in the EU, by being able to access and correlate these stored information.

This shows a big vulnerability of managed networks against restrictive governments. The task of forcing individuals of a distributed network to record routed traffic is way more difficult, than in the case of JAP.[JAP/Wiki]

Speed

A clear advantage of JohnDo's paid anonymization service, is that standard DSL speed can be guaranteed, even while staying anonymous. The free version of JAP usually only provides speed rates, less than standard ISDN.

Phantom's speed rates will not match JohnDo's. However possible direct connections in the Phantom Network will probably allow for fairly good average speed rates. And the anonymization service is free of charge.[JAP/Wiki]

Traffic Analysis

In the design of the Phantom protocol, traffic analysis counter measures including dummy packages being sent on the network were relinquished in the sake of a higher data throughput. Although no scientific research has been made to the nature of how dummy packages could hinder traffic analysis, it can be assumed that it has an impact on the difficulty of following certain streams. JohnDo's approach to anonymity uses dummy packages between mixing cascades to make traffic analysis significantly harder.

7.2.3 Attack Scenarios

As an important difference to section 7.1.3 on page 67, no scientific attacks were actually performed on JAP. So the following attack scenarios are all of theoretical nature.

Fingerprint Attack

The finger print attack illustrated in the previous chapter about TOR (see page 67), could just as well be applied here.

In JAP it seems even easier, as the entry and exit mixes of mixing cascades are known.

Replay Attack

Performing a replay attack, an attacker resends eave's dropped packages back into a routing sequence, trying to gain knowledge about sender and receiver of the message. JAP has a built in security measure against such kind of attacks, that uses checksums to prevent messages from being replayed.

In the Phantom design no such counter measure is found, but it is doubtful that such an attack could be successful in Phantom. The strict way of opening a routing path and maintaining routing tunnels makes it hard to replay any eave's dropped message. The only scenario where this attack could possibly be applied would be to locate an anonymized server, acting as a fake anonymized client sending the same distinct request over and over to the server. If however theory or praxis would proof such an attack as potentially successful, countermeasure could include the anonymized server binary including similar checksums like JAP and changing routing paths, or even AP address in case of an detected replay attack. [Kubieziel2007]

Central Resource Attack

If an attacker could manage to take over the single central InfoService server, he could trick JAP users downloading a malicious software update, which could in any imaginable way reveal private information to the attacker. However since the code is open source, this attack would hopefully be noticed quickly.

A second way to use an undermined InfoService server, would be to only offer mixture cascades completely under the control of the attacker to certain users. This is less likely to be noticed than a malicious software update.

A malicious Phantom software would also hopefully noticed quickly, because of its open source nature. An attack trying to trick a user into only using undermined nodes for his routing path is significantly harder, because of the distributed nature of the Phantom Network Database. 7.1.3 on page 68

Total Control Attack

No anonymization service so far is able to deal with this attack scenario. If an attacker controls the entire anonymization network and knows thereof, the anonymization is effectively broken.

While in such an scenario Phantom can at least contain the secrecy of the transferred information (due to end-to-end encryption), JAP would also suffer from revealing the sent information to the attacker.

7.3 I2P

I2P is an anonymizing network, offering a simple layer that identity-sensitive applications can use to securely communicate. All data is wrapped with several layers of encryption, and the network is both distributed and dynamic, with no trusted parties.

Many applications are available that interface with I2P, including mail, peer-peer, IRC chat, and others.

The I2P project was formed in 2003 to support the efforts of those trying to build a more free society by offering them an uncensorable, anonymous, and secure communication system. I2P is a development effort producing a low latency, fully distributed, autonomous, scalable, anonymous, resilient, and secure network. The goal is to operate successfully in hostile environments. even when an organization with substantial financial or political resources attacks it. All aspects of the network are open source and available without cost, as this should both assure the people using it that the software does what it claims, as well as enable others to contribute and improve upon it to defeat aggressive attempts to stifle free speech.

Anonymity is not a boolean - we are not trying to make something "perfectly anonymous", but instead are working at making attacks more and more expensive to mount. I2P is a low latency mix network, and there are limits to the anonymity offered by such a system, but the applications on top of I2P, such as Syndie, I2P mail, and I2PSnark extend it to offer both additional functionality and protection.

I2P is still a work in progress. It should not be relied upon for "guaranteed" anonymity at this time, due to the relatively small size of the network and the lack of extensive academic review. It is not immune to attacks from those with unlimited resources, and may never be, due to the inherent limitations of low-latency mix networks.[I2P]

7.3.1 Introduction

While both TOR and JAP focus on anonymizing client requests by using a proxy and a central approach to organizing and distributing routes, I2P follows the distributed approach of Phantom also aiming at anonymizing the entire internet communication creating an isolated subnet, just like Phantom.

The name I2P derives from the original project name **Invisible Internet Project**. It is an open source community project started as a proposed modification of freenet (see Glossary, page 81), to allow for alternate transports, but then grew independent in 2003. The project, mainly written in Java with some third party applications written in Python, aims at creating an anonymous network supporting all kinds of traditional communication, including for example Usenet, E-Mail, IRC, HTTP, Telnet, Jabber, . . .

How It Works

I2P is a message based network, which can be seen as an anonymous IP layer, where messages are addressed to cryptographic keys. Packages can therefore be significantly larger than standard IP-packages. The I2P network is made up of a set of nodes (used as routers) and a number of virtual inbound and outbound paths (called tunnels). Each node is identified by a long lived cryptographic key, called **RouterIdentity**. Routers communicate with in traditional manner using TCP and UDP protocols. Clients are addresses by a cryptographic key, that enables them to send and receive messages. Clients can then register with any router and temporarily reserve certain tunnels, that can be used to send and receive messages.

Instead of a central directory server (TOR, JAP), I2P uses a distributed hash table implementation (Kademila) for organizing routers and tunnels.

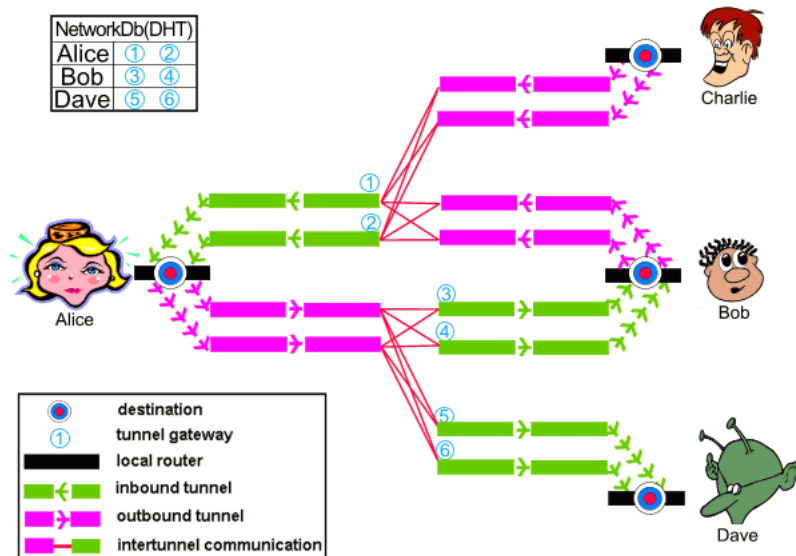


Figure 23: The I2P Network

In the above, Alice, Bob, Charlie, and Dave are all running routers with a single Destination on their local router. They each have a pair of 2-hop inbound tunnels per destination (labeled 1,2,3,4,5 and 6), and a small subset of each of those router's outbound tunnel pool is shown with 2-hop outbound tunnels. For simplicity, Charlie's inbound tunnels and Dave's outbound tunnels are not shown, nor are the rest of each router's outbound tunnel pool (typically stocked with 5-10 tunnels at a time). When Alice and Bob talk to each other, Alice sends a message out one of her (pink) outbound tunnels targeting one of Bob's (green) inbound tunnels (tunnel 3 or 4). She knows to send to those tunnels on the correct router by querying the network database, which is constantly updated as new leases are authorized and old ones expire.

If Bob wants to reply to Alice, he simply goes through the same process - send a message out one of his outbound tunnels targeting one of Alice's inbound tunnels (tunnel 1 or 2). To make things easier, most messages sent between Alice and Bob are garlic wrapped (see Glossary, page 81), bundling the sender's own current lease information so that the recipient can reply immediately without having to look in the network database for the current data.[I2P]

Encryption

A large variety of encryption is used including 2048bit ElGamal, 256bit AES (CBC mode with PKCS#5 padding), 1024bit DSA signatures, SHA256 hashes, 2048bit Diffie-Hellman negotiated connections with station to station authentication, and ElGamal / AES+SessionTag.

While originally all content sent over the I2P network was encrypted through four layers and had end-to-end encryption, more recent versions remain with three layers of encryption with general end-to-end encryption being abolished, while end-to-end garlic encryption remains between both parties' routers. Look at the figure below for illustration. No data is encrypted between Alice's application and the first router a in the outbound tunnel. From a to Bob's first router h of his inbound tunnel all data is encrypted in the fashion depicted in the figure. Having reached h , the data is sent unencrypted to Bob's application. However when a is placed on Alice's computer and h on Bob's, this is no security risk and only says that the communication between application and client/server is not encrypted.

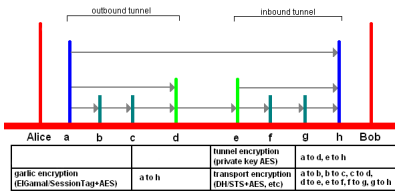


Figure 24: I2P End-To-End Encryption

7.3.2 Comparison

I2P’s approach to anonymity is very similar to Phantom, including a distributed hash table, outbound and inbound paths and tunnels between them. Instead of AP-Addresses, I2P uses the Network Database to correlate cryptographic keys with entry nodes of inbound paths. There are some more differences, let’s look at them in detail.

Usability

I2P’s approach allows for anonymization of all traditional internet communication, which allows for a broader usability than TOR or JAP, which mainly focus on websites (HTTP(s) and FTP). However currently for every such means of communication a proxy server with individual port or even a special tunnel must be configured. This largely limits I2P’s user base, as many users seem overcharged with the task of setting up I2P for their needs.

Phantom’s approach replaces the computers network communication entirely and all subsequent communication will use the same port (SSL/443). No additional settings for applications will be needed, which will simplify Phantom’s usability to a great extent. For I2P developers say that the current version is still beta, and in development. The first stable version will be more user friendly. [I2P/Netzwelt]

Compatibility

As explained in the previous point, I2P is more compatible to current network application, but still requires special versions or configuration thereof. If Phantom’s approach succeeds, all TCP/IP applications could be used without modification.

Speed

I2P aims for a low-latency anonymous network, while Phantom aims for a high throughput anonymous network. Furthermore Phantom’s design let’s you choose both the number of nodes of your routing path, and maybe even their location. This can result in a outbound/inbound path of nodes only within your region or in even in zero length routing path, which both results in speed benefits, almost like normal internet usage (except for the end-to-end encryption). Later versions of I2P will also include the option of 0-Hop out/inbound paths. [I2P, 1]

Isolation From The Internet

In this point both approaches are equal. Unlike TOR and JAP they try to create an isolated part of the internet, where both client and server are equally anonymized and protected. I2P thus has its own websites, reachable only from inside the I2P network called **eepsites**. The same counts for file sharing, instant messaging and a distributed data store.

Encryption

The encryption is also very similar between both approaches, as both use end-to-end encryption and a kind of stream encryption.

However while Phantom goes for the “traditional” Onion Encryption, I2P uses the “modern” garlic encryption. While in Onion Encryption one message at a time is enwrapped with several layers of encryption, in Garlic Routing messages are first combined (like garlic cloves), before being wrapped in layers of encryption and routed along the network. This makes traffic analysis much harder, as it is usually focused on tracing individual messages.

7.3.3 Attacks

Traffic Analysis Attack

Since scientific reviews of TOR have proven the network vulnerable to traffic analysis, newer approaches like I2P and Phantom have taken that risk into account. I2P's answer to traffic analysis is Garlic Encryption, making the tracing of individual messages much harder. On the other side Phantom's answer is the outer SSL-shell, which hides all normal traffic within SSL-streams.

Since Phantom is still in an early development stage, it is interesting in how far it would be feasible to use Garlic Routing, instead of Onion Routing. However since Garlic Routing takes requires more time, this would oppose design goal #7, while supporting design goal #6 (see design goals on page 5).[1, I2P]

Central Resource Attack

Neither I2P nor Phantom include any central point, vulnerable for this kind of attack. Both realize organization of routing paths by a distributed hash table called Network Database. Attacks on a DHT are significantly harder than on a central point.

Sybil Attack

Sybil describes a category of attacks where the adversary creates arbitrarily large numbers of colluding nodes and uses the increased numbers to help mounting other attacks. For instance, if an attacker is in a network where peers are selected randomly and they want an 80% chance to be one of those peers, they simply create five times the number of nodes that are in the network and roll the dice. When identity is free, sybil can be a very potent technique for a powerful adversary. The primary technique to address this is simply to make identity 'nonfree' - Tarzan (among others, see Glossary, page 81) uses the fact that IP addresses are limited, while IIP (see Glossary, page 81) used HashCash to 'charge' for creating a new identity.[I2P]

Neither Phantom nor I2P directly account for this kind of attack yet. However due to Phantom's design of reasonable doubt, an attacker does not win much, when controlling only a part of the network.

Predecessor Attack

In a predecessor attack, the location of the anonymized node is tried to be found by the use of statistics. An attacker would first gather traffic information to see what nodes are close to a destination, by keeping track of previous and next hops. After a certain threshold is reached, given a perfectly random sample, the attacker would see which node is statistically closer to the destination than the rest, locating the desired node with certain statistical probability.

I2P accounts for this attack by not using a perfectly random way to choose the nodes of an inbound/outbound path, but instead uses the peer selection algorithm (see Glossary, page 81). Furthermore both Phantom and I2P use paths of random length, which at least always provides some reasonable doubt.[I2P]

Partitioning Attack

A special attack for distributed systems like I2P and Phantom is the intentional invocation of net splits. Both approaches are vulnerable to this kind of attack.

The Phantom white paper suggests implementing a distributed hash table resilient to net splits.

The I2P developers further suggest to include certain net split detection mechanisms, as keeping track of trusted nodes and if a significant number of them suddenly becomes unavailable at the same time, temporarily disconnect from the network or at least warn the user of a severe security risk.[1, I2P]

Fingerprint Attack

End-to-end encryption helps a lot against this kind of attack, described earlier with TOR (page 67). However when using certain facts, like HTTP requests are usually much smaller than replies, this attack can still be powerful to distinguish outbound and inbound paths and even correlate them using the fingerprint sizes.

I2P Garlic Routing gives some more protection against this attack by delaying messages in a queue and then sending them mixed with other messages. Phantom's standard Onion Routing does not provide this extra security, and seems vulnerable against this kind of attack. [I2P]

7.4 Synthesis

In summary there are two distinct approaches to providing anonymity: the centralistic approach that provides access to the “normal” internet followed by TOR and JAP and the distributed approach that aims at creating a isolated subset of the internet, where end-to-end encryption can be ensured followed by I2P and Phantom. The first approach is following the classic development of mixes and onion routing, while the second approach takes recent developments in p2p technology into account.

Both approaches have their own flaws and merrits. A centralistic approach is always vulnerable to DoS attacks and less scalable. On the other hand routes can be better organized and controlled (e.g. mixing cascades in JAP containing mixes from different providers in different countries) and it holds the possibility of quick reaction to attacks. A distributed approach by its nature is less likely to be affected by DoS attacks and more scalable. However this approach has its own flaws, like for example net splits. Also the choice of routers is entirely up to selection algorithms and the user.

No modern approach is better than classic ones, because of being new and modern. However the newer approaches have been developed with the knowledge of all the weakspots and successful attacks of/against the classic approaches. Also the general development of the internet tends to distribution instead of centralization, both in the sense of resources as also in users taking an active part in the development of the net (this developments are often gathered under the buzzword Web2.0). I2P and Phantom are following this trend fo distributed resoruces and also provide a more active part to the user (e.g. selection of the length and location of routing nodes).

However, none of the two new approaches have met much scientific review yet, and only that and years of experience will proove them to be really more fit for the task, than their classic counterparts.

8 Outlook

The current state of the implementation of the project was discussed in the first part (6), then the current design was analyzed and compared to other approaches in the second part (59). Now this chapter will give a short overview of what needs to be done to complete implementing the protocol and list some ideas for improvement of the design.

8.1 Implementation

Routing Path

Together with this draft the first part of the design, the construction of **Routing Paths**, was implemented and illustrated with a prototype.

The second part **Routing Tunnels** and the third part the **Phantom Network Database** still need to be implemented. Implementing the other parts will also require some additions to the first part, which have been left out, since they were not necessary for the routing path, and could not be tested without the other parts. This includes stream encryption keys for the Routing Tunnels and proper routing table entries for the Phantom Network Database. Another aspect of **Routing Paths**, that has not been implemented with this prototype are the dummy packages mentioned in section 2.1 on page 7. Instead processed packages were simply left in the array in their encrypted state. Following the design here would require to precompute the state of the array after each routing node and give the node information of which and how many dummy packages to add to the array, in order to make it fit the correct state (hash) for the next node. This process has three advantages against the current practice of leaving packages inside the array:

- there is no need processed information needs to be passed on, and thus it is better discarded
- the number of packages will vary from node to node, making it hard to tell the number of total nodes in the routing path
- the hash of the entire encrypted array (minus the package for the current node) could be put inside the package for the current node

Despite these advantages this procedure seems to overcomplicate things more than necessary. Double encrypted setup packages should be considered secure, else there is no need for the encryption at all. So they can as well be left in the array. Further a changing array might also reveal more information to an attentive attacker than an array which is unmodified the entire process and simply passed along. The hash of the entire not changing array can obviously not be put inside the individual setup packages. So in this implementation it was put at the same level of the packages, but signed with the construction certificate, which can be found inside the setup package. So no loss in security there either.

Apart from these modifications the design has been closely followed in the implementation.

Routing Tunnel

The white paper ([1]) suggests the three parts of the protocol to be implemented as independent as possible. However it still makes sense to use a shared environment, so certain components of the established Routing Path prototype could be used: Logging, cryptography, serialization, . . .

The implementation of Routing Tunnels, would probably require a distinction between anonymized node, X-node, EXIT-node and ENTRY-node. Since the procedure is however symmetric big code chunks can possibly be used by all four parts. As mentioned in this section, the SetupPackages sent in the Routing Path Construction code would have need to be extended by providing stream encryption keys for the establishment of Routing Tunnels.

Phantom Network Database

It was suggested to choose a well established implementation of a distributed hash table for this part, and if necessary adapt it to the requirements listed in section 2.3.2 on page 17. A good idea for an existing implementation of a DHT seems Kademlia (see Glossary, page 81), which I2P is using for the same purpose. As mentioned in this section, the ready made RoutingTable conveyed in the SetupPackage, would have to be updated for the purposes of the implemented DHT.

8.2 Further Ideas

Network Logic

One of the most interesting problems remaining is how to logically replace the computer's network logic, in order to use the Phantom Protocol instead of TCP/IP for all outgoing network traffic. An idea stated in the mailing list was the Tun/Tap interface, which like all other solutions, has its own merits and flaws. This problem will not be solved easily, but if implemented properly will distinguish Phantom from any other approach to anonymization.

Network Protocol

In this first stage of development it is tried to replace the TCP protocol with Phantom on the application layer. However later versions should at least also include UDP for a good integration of Phantom into the internet protocol family. Of course UDP can be tunneled by TCP, but that somewhat destroys the purpose of UDP.

Even later some interesting ideas in the mailing list included the attempt to try to make Phantom entirely independent from lower network layers. Which would allow motivated talented volunteers to implement Phantom over radio, or even steganography. At least in theory.

DNS

One of the design goals of Phantom is to make it easy enough to be used and accepted by the broad population. This will require some way of an easy addressing of websites or services, other than their AP-Address. The "normal" internet solved this dilemma by the introduction of DNS (Domain Name Service), which allows to resolve URLs (Uniform Resource Locator) to an IP-Address by the use of hierachically ordered layers of central DNS servers for several domains.

All other approaches discussed in this draft use DNS to some extent to resolve URLs, and all approaches suffer from a problem called **DNS Leakage**, clear text requests revealing information about the anonymized computer. Possible solutions to this would be to extent the PNDB with DNS resolution capabilities. There are projects working on a distributed heterarchic DNS resolution like **DistributedDNS** or **PowerDNS**, which could be used for this purpose. The advantage would be the DNS requests could be encrypted, anonymous (using the routing path) and implemented in the distributed design.

(Geo)IP Blacklists

Another possible extenstion would be the introduction of blacklists to the PNDB. This would enable to blacklist malicious nodes, which tried to attack or sabotage the network or behave strangely. Using GeoIP would further allow to exclude routing nodes in endangered countries, where the government has an active interest in preventing anonymization.

Exchange With Other Approaches

A step I deem very important in the development of Phantom is exchange with developers of the other anonymization approaches presented in this draft. While the TOR and JAP developers have years of experience with problems, attacks and hurdles and may have found solutions or ideas how to succumb them, which they could share, I2P developers develop a very similar approach and thus are bound to meet and have met the same problems. Thus exchange with any of these would be an invaluable help to the development of Phantom.

[III. Socio-Political Part]

9 Socio-Political Aspects

10 Juristical Aspects

11 Glossary

Distributed Hash Table

Diffie-Hellman Key Exchange

DoS Attack

EigenTrust

Freenet

Garlic Routing

IIP

Kademlia

Onion Encryption

Peer Selection Algorithm

Remailer Protocol

Screen

SOCKS

Tarzan

12 References

For digital references, a copy can be found in the ./references folder.

12.1 Libraries

OpenSSL

Cryptography and SSL/TLS Toolkit
<http://www.openssl.org/>

GoogleTestPrimer

framework for writing C++ unit tests
<http://code.google.com/p/googletest/>

GoogleProtoBuf

framework for (de)serialization and data interchange
<http://code.google.com/p/protobuf/>

GLog (GoogleLog)

application-level logging library
<http://code.google.com/p/google-glog/>

Yaml-cpp

yaml-cpp is a YAML parser and emitter in C++
<http://code.google.com/p/yaml-cpp/>

GC (Garbage Collector)

Boehm-Demers-Weiser conservative garbage collector for C++
http://www.hpl.hp.com/personal/Hans_Boehm/gc/

12.2 Lecture Notes

Uebung Verteilte Systeme, 4. Uebung (Lehrstuhl fuer Verteilte Systeme, University of Erlangen)

12.3 Books

Technical

References

- [CDK2005] George Coulouris, Jean Dollimore, Tim Kindberg, Distributed Systems (Concepts And Design), Addison-Wesely, Essex 1988, Fourth Edition 2005
- [Stevens1998] W. Richard Stevens, Unix Network Programming, Prentice Hall PTR, New Jersey 1998, First Edition

Socio-Political

References

- [Bäumler2003] Helmut Bäumler, Anonymität im Internet (Grundlagen, Methoden und Tools zur Realisierung eines Grundrechts), Vieweg Verlag, Braunschweig/Wiesbaden 2003, 1. Auflage
- [Brunst2009] Phillip W. Brunst, Anonymität im Internet (rechtliche und tatsächliche Rahmenbedingungen), Max-Planck-Institut, Freiburg 2009
- [Gaycken2008] Sandro Gaycken, Constanze Kurz, 1984.exe (Gesellschaftliche, politische und juristische Aspekte moderner Überwachungstechnologien), transcript Verlag, Bielefeld 2008
- [Kubieziel2007] Jens Kubieziel, Anonym im Netz (Techniken der digitalen Bewegungsfreiheit), Open Source Press München, 2007
- [Prantl2008] Heribert Prantl, Der Terrorist als Gesetzgeber (Wie man mit Angst Politik macht), Droemer Verlag, München, 2008
- [Schulzki-Haddouti2000] Christiane Schulzki-Haddouti, Vom Ende der Anonymität (Die Globalisierung der Überwachung), Verlag Heinz Heise, Hannover 2000, 1. Auflage
- [Schulzki-Haddouti2003] Christiane Schulzki-Haddouti, Bürgerrechte im Netz, Bundeszentrale für politische Bildung, Bonn 2003

12.4 Artikel

References

- [Roessler1998] Thomas Roessler, Anonymität im Internet, Datenschutz und Datensicherheit 22 / 1998, Seite 619 - 622
- [Rost2003] Martin Rost, Zur gesellschaftlichen Funktion von Anonymität (Anonymität im soziologischen Kontext), Datenschutz und Datensicherheit 27 / 2003, Seite 155 - 158

12.5 Papers

The Phantom Protocol

References

- [1] Magnus Bråding (2009): Generic, Decentralized, Unstoppable Anonymity: The Phantom Protocol (Version: 0.8)

TOR

References

- [OverlierSyverson2006] Lasse Øverlier, Paul Syverson, Locating Hidden Servers, IEEE Symposium on Security and Privacy, May 2006.
- [Murdoch2006] Steven J. Murdoch, Hot or Not: Revealing Hidden Services by their Clock Skew, CCS 2006
- [Murdoch2008] Steven J. Murdoch, Sebastian Zander, An Improved Clock-skew Measurement Technique for Revealing Hidden Services, 17th USENIX Security Symposium, San Jose 2008

JAP

References

- [Pfitzmann1991] Andreas Pfitzmann, Birgit Pfitzmann, Michael Waidner, ISDN-Mixes: Untraceable communication with very small bandwidth overhead, GI/ITG Conference, February 1991

12.6 Academic Thesis

12.7 Internet

Encryption

AES (Advanced Encryption Standard)

http://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=311664832 (Wikipedia)

Block Cipher Modes

http://en.wikipedia.org/w/index.php?title=Block_cipher_modes_of_operation&oldid=312409841 (Wikipedia)

Padding

<http://www.di-mgt.com.au/cryptopad.html> (DI Management Services, Sydney)

Hash Functions

http://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=310539152 (Wikipedia)

OpenSSL

[http://www.ibm.com/developerworks/views/linux/libraryview.jsp?topic_by=All+topics+and+related+products&sort_order=\(IBM\)](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?topic_by=All+topics+and+related+products&sort_order=(IBM))

TOR

References

[TOR[1] TOR <http://www.torproject.org/>

[TOR[2] TOR/Wiki http://en.wikipedia.org/w/index.php?title=Tor_%28anonymity_network%29&oldid=319544190

JAP

References

[JAP] http://anon.inf.tu-dresden.de/index_en.html

[JAP/Wiki] http://de.wikipedia.org/w/index.php?title=Java_Anon_Proxy&oldid=64293446

I2P

References

[I2P] <http://www.i2p2.de>

[I2P/Wiki] <http://de.wikipedia.org/w/index.php?title=I2P&oldid=65516007>
<http://en.wikipedia.org/w/index.php?title=I2P&oldid=319683363>

[I2P/Netzwelt] <http://www.netzwelt.de/news/79089-anonyme-netz-netz-i2p-neuer-version.html> (Das anonyme Netz im Netz: I2P in neuer Version)
http://www.netzwelt.de/news/75371_5-i2p-anonyme-netz-netz.html (I2P: Das anonyme Netz im Netz)

12.8 Images

Phantom White Paper

All images used in section 2 (Protocol Design), are taken from the Phantom White Paper (Generic, Decentralized, Unstoppable Anonymity: The Phantom Protocol, Version: 0.8, 2009), with permission of the author Magnus Bråding.

12.9 Standards

[PKCS5] PKCS #5, Password-Based Encryption Standard, RSA Laboratories, Version 2.0, March 1999.

List of Algorithms

| | | |
|----|---|----|
| 1 | RSA encrypt | 29 |
| 2 | RSA decrypt | 30 |
| 3 | AES encrypt | 31 |
| 4 | AES decrypt | 32 |
| 5 | AES crypt | 32 |
| 6 | RSA sign | 33 |
| 7 | RSA verify | 34 |
| 8 | SHA256 Hash Code | 34 |
| 9 | sending data to remote socket | 36 |
| 10 | receive data on socket | 37 |
| 11 | initialize server socket | 38 |
| 12 | | 39 |
| 13 | Building a routing path | 43 |

List of Figures

| | | |
|----|--|----|
| 1 | Phantom Protocol Overview | 6 |
| 2 | Illustration of a routing path | 7 |
| 3 | eight routing nodes picked from the Phantom Network Database | 7 |
| 4 | routing path in the first round | 8 |
| 5 | routing path creation, after the first round was completed (Xc is terminal node) | 9 |
| 6 | routing path after the second round | 10 |
| 7 | final established routing path | 11 |
| 8 | a routing tunnel | 12 |
| 9 | Outbound Routing Tunnel Sequence Diagram | 13 |
| 10 | outbound routing tunnel | 14 |
| 11 | Inbound Routing Tunnel Sequence Diagram | 15 |
| 12 | inbound routing tunnel | 16 |
| 13 | Sequence Diagram | 20 |
| 14 | UML Inheritance Diagram | 21 |
| 15 | UML Composite Diagramm | 21 |
| 16 | Phantom Routing Node Protoype | 22 |
| 17 | Phantom Routing Node Protoype | 23 |
| 18 | Sample Config File | 24 |
| 19 | Setup Package <i>.proto</i> File | 25 |
| 20 | SignAndVerify Unit Test | 27 |
| 21 | How TOR works (from [?]) | 65 |
| 22 | How JAP works (from [JAP]) | 69 |
| 23 | The I2P Network | 74 |
| 24 | I2P End-To-End Encryption | 75 |

List of Tables

| | | |
|---|--|----|
| 1 | Node Types | 9 |
| 2 | verification modes for OpenSSL sockets | 38 |
| 3 | plan for setting connection IDs | 44 |