MessageVortex

Transport Independent and Unlinking Messaging

Inauguraldissertation
zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel
von

Martin Gwerder (06-073-787)

von Glarus GL

October 22, 2020

Original document available on the edoc sever of the university of Basel edoc.unibas.ch.

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
Auf Antrag von

Prof. Dr. Christian F. Tschudin
Prof. Dr. Heiko Schuldt

Basel, der 18.2.2020 durch die Fakultätsversammlung

Prof. Dr. Martin Spiess

Abstract

In this paper, we introduce an unobservable message anonymization protocol, named MessageVortex. It bases on the zero-trust principle, a distributed peer-to-peer (P2P) architecture, and avoids central aspects such as fixed infrastructures within a global network. It scores over existing work by blending its traffic into suitable existing transport protocols, thus making it next to impossible to block it without significantly affecting regular users of the transport medium. No additional protocol-specific infrastructure is required in public networks and allows a sender to control all aspects of a message such as the degree of anonymity, timing, and redundancy of the message transport without disclosing any of these details to the routing or transporting nodes. Part of this work is an RFC document attached in Appendix A describing the protocol. It contains all the necessary information to build protocol nodes. The RFC draft is available through the official RFC channels. Additionally, the RFC document, additional documents, and a reference are available under https://messagevortex.net/.

Acknowledgments

I want to thank my wife Cornelia and my lovely three kids (Saphira, Florian, and Aurelius) for their patience and their support. Without them I could never have done this work.

I want to thank Prof. Dr. C. Tschudin and the University of Basel for the possibility of writing this work and for the challenges they opposed to me, allowing me to grow.

Dr. Andreas Hueni for his thoughts and challenging outside-the-normal-box thinking.

Prof. Dr. Carlos Nicolas of the University of Northwestern Switzerland for being such a valuable sparring partner allowing me to test my ideas.

I want to acknowledge all the individuals who have coded for the LaTeXproject for free. It is due to their efforts that we can generate professionally typeset PDFs (and far more) for free.

Contents

List of Corrections

Part I

Introduction

***

The most effective way to do it
is TO DO IT
Amelia Earhart

1 Preface

Almon Brown Strowger was the owner of a funeral parlor in St. Petersburg. He filed a patent on March 10th, 1891 for an "Automatic Telephone Exchange" [147]. This patent built the base for modern automated telephone systems. According to several sources, he was annoyed by the fact that the local telephone operator was married to another undertaker. She diverted potential customers of Mr. Strowger to her husband instead, which caused Almon B. Strowger to lose business. In 1922, this telephone dialing system, which is nowadays called pulse dialing, became the standard dialing technology for more than 70 years until tone dialing replaced it.

This dialing technology is the base for automatic messaging for voice and text messages (e.g., telex) up until today and is the foundation for current routed networks. These networks build the base for our communication-based Society these days and allow us to connect quickly with any person or company of our wish. We use these networks today as communication meaning for all purposes, and most of the people spend minimal thoughts on the possible consequences arising if someone puts hands on this communication.

This collected data may be used to judge our intentions and thus is not only confidential if we have something to hide. This problem has dramatically increased in the last years as big companies and countries started to collect all kinds of data and created the means to process them. It allows supposedly to judge peoples not only on what they are doing but as well, on what they did and what they might do. Numerous events past and present show that actors, some of which are state-sponsored, collected data on a broad base within the Internet. Whether this is a problem or not is a disputable fact. Undisputed is, however, that such data requires careful handling, and accusations should then base on solid facts. While people may classify personalized advertising as legit use, a general classification of citizens is broadly considered unacceptable[17, 109, 10, 59, 82].

To show that this may happen even in democracies, we might refer to events such as the "secret files scandal" (or "Fichenskandal") in Switzerland. In the years from 1900 to 1990 Swiss government collected 900'000 files in a secret archive (covering more than 10% of the natural and juristic entities within Switzerland at that time). The Swiss Federal Archives document this event in depth[82].

Whistleblower Edward Snowden leaked a vast amount of documents. These documents suggest that such attacks on privacy are commonly made on a global scale. The documents leaked in 2009 by him claim that there was a data collection starting in 2010. Since these documents are not publicly available, it is hard proving the claims based on these documents. However – A significant number of journalists from multiple countries screened these documents claiming that the information seems credible. According to these documents (verified by NRC), NSA infiltrated more than 50k computers with malware to collect classified or personal information. They furthermore infiltrated Telecom-Operators (mainly executed by British GCHQ) such as Belgacom to collect data and targeted high members of

governments even in associated states (such as the mobile phone number of Germany's president) [17, 109, 10, 2, 59]. A later published shortened list of "selectors" in Germany showed 68 telephone and fax numbers targeting economy, finance, and agricultural parts of the German government. A global survey done by the freedom house[53] claims a decrease in Internet freedom for the þ8 year in a row.

This list of events shows that big players are collecting and storing vast amounts of data for analysis or possible future use. The list of events also shows that the use of such data was at least partially questionable. This work analyses the possibility of using state-of-the-art technology to minimize the information footprint of a person on the Internet.

We leave a large information footprint in our daily communication. On a regular email, we disclose everything in an "postcard" to any entity on its way. Even when encrypting a message perfectly with today's technology (S/MIME[50] or PGP[41]), it still leaves at least the originating and the receiving entity disclosed, or we rely on the promises of a third party provider which offers a proprietary solution. Even in those cases, we leak pieces of information such as "message subject", "frequency of exchanged messages", "size of messages", or "client being used". A suitable anonymity protocol must cover more than the sent message itself. It includes, besides the message itself, all metadata, and all the traffic flows. Furthermore, a protocol to anonymize messages should not rely on the trust of infrastructure other than the infrastructure under control of the sending or receiving entity. Trust in any third party might be misleading in terms of security or privacy.

Furthermore, central infrastructure is bound to be of particular interest to anyone gathering data. Such control by an adversary would allow manipulating the system or the data or the data flow. So, avoiding a central infrastructure is a good thing when it comes to minimizing an information footprint available to a single entity.

Leaving no information trail when sending information from one person to another is hard to achieve. Most messaging systems disclose at least the peer partners when posting messages. Metadata such as starting and endpoints, frequency, or message size are leaked in all standard protocols even when encrypting messages.

Allowing an entity to collect data may affect senders and recipients of any information. The collection of vast amounts of data allows a potent adversary to build a profile of a person. Unlike in the past, the availability of information has risen to a never known extent with the Internet.

An entity in possession of such Profiles may use them for many purposes. These include service adoption, directed advertising, or classification of citizens. The examples given above show that the effects of this data is not limited to the Internet but reaches us effectively in the real world.

The main problem of this data is that it may be collected over a considerable amount of time and evaluated at any time. It even happened that standard prac-

tices at a time are differently judged upon at a later time. Persons may then be judged retrospectively upon these types of practice. This questionable type of judgment is visible in the tax avoidance discussion[5].

People must be able to control their data footprint. Not providing these means does effectively allow any country or a more prominent player to ban and control any number of persons within or outside the Internet.

We design in this work a new protocol. This protocol allows message transfer through existing communication channels. These messages are next to unobservable to any third party. This unobservability does not only cover the message itself but all metadata and flows associated with it. We called this protocol "*MessageVortex*" or just "*Vortex*". The protocol is capable of using a wide variety of transport protocols. It is even possible to switch protocols while the messages are in the transfer. This behavior allows media breaches (at least on a protocol level) and makes the analysis even harder.

The new protocol allows secure communication without the need to trust the underlying transport media. Furthermore, the usage of the protocol itself is possible without altering the immediate behavior of the transport layer. The transport layers' regular traffic does, therefore, increase the noise in which hidden information has to be searched.

The primary goal of the protocol is to enable freedom of speech, as defined in Article 19 of the International Covenant on Civil and Political Rights (ICCPR)[157].

> everyone shall have the right to hold opinions without interference

and

> Everyone shall have the right to freedom of expression; this right shall include freedom to seek, receive and impart information and ideas of all kinds, regardless of frontiers, either orally, in writing or print, in the form of art, or through any other media of his choice.

We imply that not all participants on the Internet share this value. As of September 1st, 2016 Countries such as China (signatory), Cuba (signatory), Qatar, Saudi Arabia, Singapore, United Arab Emirates, or Myanmar did not ratify the ICCPR. Other countries such as the United States or Russia did either put local laws in place superseding the ICCPR or made reservations rendering parts of it ineffective. We may, therefore, safely assume that freedom of speech is not given on the Internet, as at least countries explicitly supersede them.

Network packets may pass through any point of the world. A sender has no control over it. This lack of control is since every routing device decides on its own for the next hop. This decision may be based on static rules or influenced by third party nodes or circumstances (e.g., BGB, RIP, OSPF...). It is furthermore not possible to detect what way has a packet taken. The standard network diagnostic tool `tracert` respectively `traceroute` returns a potential list of hops. This

list is only correct under certain circumstances (e.g., a stable route for multiple packets or same routing decisions regardless of other properties than the source and destination address). Any Output of these tools may, therefore, not taken as a log of routing decisions. There is no possibility in standard IP routed networks to foresee a route for a packet, nor can it be measured, recorded, or predicted before, while, or after sending.

As an example of the problems analyzing a packet route, we may look at `traceroute`. According to the man page of traceroute, `traceroute` uses UDP, TCP, or ICMP packets with a short TTL and analyzes the IP of the peer sending a TIME_EXCEEDED (message of the ICMP protocol). This information is then collected and shown as a route. This route may be completely wrong. The man page describes some of the possible causes.

We cannot state that data packets we are sending are passing only through countries accepting the ICCPR to the full extent, nor can we craft packages following such a rule.

```
$traceroute www.ietf.org
traceroute to www.ietf.org.cdn.cloudflare-dnssec.net (104.20.0.85), 64 hops max
1 147.86.8.253 0.418ms 0.593ms 0.421ms
2 10.19.0.253 1.177ms 0.829ms 0.782ms
3 10.19.0.253 0.620ms 0.427ms 0.402ms
4 193.73.125.35 1.121ms 0.828ms 0.905ms
5 193.73.125.81 2.991ms 2.450ms 2.414ms
6 193.73.125.81 2.264ms 1.961ms 1.959ms
7 192.43.192.196 6.472ms 199.543ms 201.152ms
8 130.59.37.105 3.465ms 3.138ms 3.121ms
9 130.59.36.34 3.904ms 3.897ms 4.989ms
10 130.59.38.110 3.625ms 3.333ms 3.379ms
11 130.59.36.93 7.518ms 7.232ms 7.246ms
12 130.59.38.82 7.155ms 17.166ms 7.034ms
13 80.249.211.140 22.749ms 22.415ms 22.467ms
14 104.20.0.85 22.398ms 22.222ms 22.146ms
```

Figure 1.1: A traceroute to the host www.ietf.org

To enable freedom of speech, we need a mean of transport for messages which keep sender and recipient anonymous to an adversary.

2 Our Contribution

This thesis contributes to anonymisation with an asynchronous messaging protocol called *MessageVortex*.

The protocol employs a new type of programmable forwarders called “routing nodes” (nodes) with a novel way of message mixing, moving away from a strictly chunked and onionized system, to a system where routing operations allow to increase or decrease in size without differentiating between decoy traffic and message routing. We refer to the instructions required to process a node as “routing blocks”. These routing blocks have an onionized structure, only exposing the required information for the current node. Routing blocks may travel with a message or join at any common routing node with the message.

To non-traceable routing this, we introduce a novel type of routing operation called

“addRedundancy”. This operation is a Reed-Solomon-calculation with encryption and a new type of padding. This operation transposes the received information in a form bigger or smaller than the original message by adding or removing redundancy operations. The applied padding structures the message in such a way that any possible result of a decryption operation results in a plausible padding structure. With standard paddings, decoy operations on traffic would possibly be identifiable as the resulting padding structure may be invalid leaking information. After applying these operations, the routing node then sends this transposed information to subsequent peers without any knowledge of what parts of the sent messages are relevant for the successful message delivery. Therefore, applying such operations makes it impossible for any node to differentiate between decoy traffic and real message traffic. Furthermore, tagging beyond peering nodes is not possible, as building relations between messages of non-neighboring nodes is not possible.

An outside observer is unable to identify messages, as they do not use proprietary communication protocol but hide within other standard internet protocols. We blend these transport protocols without modifying the servers used for message transport. This property makes the protocol very robust as the prosecution of server administrators is not sensible if traffic is running over their infrastructures.

As the structure of routing blocks does not expose the encryption keys required to build routing blocks for a peering node, a malicious node may only discover other possible peer partners when routing traffic without gaining the capability of talking to them. Other properties, such as type of routed traffic, message size, message content, communication partners, or intensity of communication, remain hidden. External global observers are unable to differentiate between regular protocol traffic and Vortex traffic. Assuming an observer capable of identifying the steganographically hidden information, he may apply censorship but remains unable to trace messages according to externally attributes, even assuming that he has additional information from collaborating nodes within the message path.

Our protocol differentiates from other protocols by the fact that our way of mixing and routing messages does not rely on knowingly injected decoy traffic and that we are capable of piggybacking multiple other carrier protocols without modifying the required, already available infrastructure on the internet or requiring dedicated infrastructure. The carrier protocols may even be switched during routing, making it even harder to observe message traffic.

3 Scope and Aproach

The main topic of this thesis was defined as follows:

- Is it possible to have a messaging protocol used on the Internet, based on “state of the science” technologies offering a high level of unlikability (sender

and receiver anonymity) towards an adversary with a high budget and privileged access to Internet infrastructure?

Based on this central question, there are several sub-questions grouped around various topics:

1. What technologies and methods may be used to provide sender and receiver anonymity and unlinkability when sending messages against a potential adversary?
   This question covers the principal part of the work. We first elaborate on a list of criteria for the *MessageVortex*protocol. We then create a list of suitable technologies and methods. Based on these findings, we define a protocol combining these technologies and researches into a solution. This solution is implemented and analyzed for suitability based on the criteria specified previously.
   

2. How can entities utilizing *MessageVortex*be attacked, and what measures are available to circumvent such attacks?
   Within this question, we look at various attacks and test resistance of the protocol based on the definition of the protocol. We do this by first collecting well-known attacks (either generic or specific to a technology used in the protocol). We then elaborate if those attacks might be successful (and if so under what circumstances).
   

3. How can design mitigate attacks target anonymity of a sending or receiving entity within *MessageVortex*?
   Within this question, we define baselines to mitigate attacks by identifying guidelines for using the protocol. We analyze the effectiveness of the guidelines and elaborate on the general achievement level of the protocol by looking again at the criteria defined in SQ1.
   

4 Notation

4.1 Cryptography

The theory in this document is heavily based on symmetric encryption, asymmetric encryption, and hashing. To use a uniformed notation I use $E^{K_a}(M)$ (where a is

an index to distinguish multiple keys) resulting in $\mathbf{M}^{\mathbf{K_a}}$ as the encrypted message. If we are reflecting a tuple of information, we write it in boldface. To express the content of the tuple, we use angular brackets $\mathbf{L}\langle\mathbf{normalAddress}, \mathbf{vortexAddress}\rangle$. If we want Messages encrypted with multiple keys do list the used keys as a comma-separated list in superscript $E^{K_b}\left(E^{K_a}(M)\right) = M^{K_a,K_b}$.

For a symmetric encryption of a message $\mathbf{M}$ with a key K_a resulting in $\mathbf{M}^{\mathbf{K_a}}$ where a is an index to distinguish different keys. Decryption uses therefore $D^{K_a}(\mathbf{M}^{\mathbf{K_a}}) = \mathbf{M}$.

As notation for asymetric encryption we use $E^{K_a^1}(\mathbf{M})$ where as K_a^{-1} is the private key and K_a^1 is the public key of a key pair K_a^p. The asymmetric decryption is noted as $D^{K_a^{-1}}(\mathbf{M})$.

For hashing, we do use $H(\mathbf{M})$ if unsalted and H^{S_a} if using a salted hash with salt S_a. The generated hash is shown as H_M if unsalted and $H_M^{S_a}$ if salted.

If we want to express what details contained in a tuple we use the the notation $\mathbf{M}\langle\mathbf{t}, \mathbf{MURB}, \mathbf{serial}\rangle$ respectively if encrypted $\mathbf{M}^{\mathbf{K_a}}\langle\mathbf{t}, \mathbf{MURB}, \mathbf{serial}\rangle$.

$$
\begin{aligned}
\text{asymetric:} E^{K_a^{-1}}(\mathbf{M}) &= \mathbf{M}^{K_a^{-1}} \\
D^{K_a^1}\left(E^{K_a^{-1}}(\mathbf{M})\right) &= \mathbf{M} \\
D^{K_a^{-1}}\left(E^{K_a^1}(\mathbf{M})\right) &= \mathbf{M} \\
\text{symetric:} E^{K_a}(\mathbf{M}) &= \mathbf{M}^{K_a} \\
D^{K_a}\left(E^{K_a}(\mathbf{M})\right) &= \mathbf{M} \\
\text{hashing (unsalted):} H(\mathbf{M}) &= \mathbf{H}_M \\
\text{hashing (salted):} H^{S_a}(\mathbf{M}) &= \mathbf{H}_M^{S_a}
\end{aligned}
$$

In general, subscripts denote selectors to differentiate the values of the same type, and superscript denotes relevant parameters to operations expressed. The subscripted and superscripted pieces of information are omitted if not needed.

We refer to the components of a *VortexMessage*as follows:

$$
\begin{aligned}
\text{Prefix component:}\mathbf{PREFIX} &= D^{K_a^1}\left(\mathbf{P}^{K_a^{-1}}\right) = D(\mathbf{P}) \\
\text{Header component:}\mathbf{HEAD} &= D^{K_a^1}\left(\mathbf{H}^{K_a^{-1}}\right) = D(\mathbf{H}) \\
\text{Route component:}\mathbf{ROUTE} &= D^{K_a^1}\left(\mathbf{R}^{K_a^{-1}}\right) = D(\mathbf{R})
\end{aligned}
$$

In general, a decrypted Block is written as a capitalized multi-character boldface sequence. An encrypted Block is expressed as a capitalized, single character, boldface letter.

4.2 Code and commands

We write code blocks as a light grey block with line numbers:

```
1 public class Hello {
2 public static void main(String args[]) {
3 System.println("Hello. "+args[1]);
4 }
5 }
```

Commands entered at the command line are in a grey box with a top and bottom line. Whenever root rights are required, the command line is prefixed with a “#”. Commands not requiring specific rights are prefixed with a “$”. Lines without a trailing “$” or “#” are output lines of the previous command. If long lines are split to fit into the paper, a “↩” is inserted to indicate that a line break was inserted for readability.

```
# su -
# javac Hello.java
# exit
$java Hello
Hello.
$java Hello "This is a very long command-line that had to be broken to fit into the code box displayed  ↩
     on this page."
Hello. This is a very long command-line that had to be broken to fit into the code box displayed on  ↩
     this page.
```

4.3 Hyperlinking

The electronic version of this document is hyperlinked. References to the glossary or the literature may be clicked to find the respective entry. Chapter or table references are clickable too.

5 Document Structure and Systematics

Part II

Relevant Concepts and Technologies

***

Where does a snake's tail start?
My son Florian

6 Anonymity Research

In this section, we collect protocols research related to anonymity. We did not stick to anonymous message transfer. Instead, we took a broad focus in terms of technology and outlined in each protocol strengths and weaknesses identified, which may be relevant to this research.

6.1 Definition of Anonymity

As the definition for Anonymity we take the definition as specified in [114].

> Anonymity of a subject means that the subject is not identifiable within a set of subjects, the anonymity set.[1]

and

> Anonymity of a subject from an attacker's perspective means that the attacker cannot sufficiently identify the subject within a set of subjects, the anonymity set.[1]

We define the anonymity set as the set of all possible subjects within a supposed message. The anonymity of a subject towards an observing third party is a crucial factor as it relates directly to our adversary model.

6.2 k-Anonymity

k-anonymity is a term introduced in [3]. This work claims that entities are not responsible for an action if an observer is unable to match a specific action to less than k entities.

The Document distinguishes between *Sender k-anonymity*, where the sending entity can only be narrowed down to a set of k entities and *Receiver k-anonymity*.

The size of k is a crucial factor. One of the criteria is the legal requirements of the jurisdiction. Depending on the jurisdiction, it usually is not possible to prosecute someone if an action is not directly coupled to one person. Another criterion might be the decreasing of k over time. If a Vortex account is used, we have to assume that some vortex identities go out of commission over time. If k is chosen according to a legal requirement, it should be taken into account that k might be decreasing over time.

[1] footnotes omitted in quote

6.3 ℓ-Diversity

In [89] an extended model of k-anonymity is introduced. In this paper, the authors emphasize that it is possible to break a k-anonymity set if there is additional information available which may be merged into a data set so that a distinct entity can be filtered from the k-anonymity set. In other words, if an anonymity set is to tightly specified, additional background information might be sufficient to identify a specific entity in an anonymity set.

It might be arguable that a k-anonymity in which a member is not implicitly k-anonymous still is sufficient for k-anonymity in its sense. However, the point made in this work is right and is taken into account. Their approach is to introduce an amount of invisible diversity into k-anonymous sets, so that common background knowledge is no longer sufficient to isolate a single member.

6.4 t-Closeness

While ℓ-diversity protects the identity of an entity, it does not prevent information gain. A subject which is in a class has the same attributes. This is where t-closeness[108] comes into play. t-closeness is defined as follows:

> An equivalence class is said to have t-closeness if the distance between the distribution of a sensitive attribute in this class and the distribution of the attribute in the whole table is no more than a threshold. A table is said to have t-closeness if all equivalence classes have t-closeness.

6.5 Single Use Reply Blocks and Multi-Use Reply Blocks

Chaum first introduced the use of reply blocks in [24]. A routing block, in general, is a structure allowing to send a message to someone without knowing the targets' real address. Reply blocks may be differentiated into two classes "Single Use Reply Blocks" (SURBs) and "Multi-Use Reply Blocks" (MURBs). SURBs may be used once while MURBs may be used a limited number of times.

Within our research, we discovered that if a routing protocol is reproducible, the traffic of a MURB may be used to identify some of the properties of the message. Depending on the type of attack, the block has to be repeated very often. For this reason, we limited the number of replays to a low number. The concept is that we have, in our case a routing block, which might be used up to n times ($0 < n < 127$). It is easily representable in a byte integer (signed or unsigned) on any system. It is big enough to support human communication sensibly and is big enough to add

not too much overhead when rerequesting more MURBs. The number should not be too big because if a MURB is reused, the same pattern of traffic is generated, thus making the system susceptible to statistical attacks.

6.6 Censorship

As a definition for censorship we take

> Censorship: the cyclical suppression, banning, expurgation, or editing by an individual, institution, group or government that enforce or influence its decision against members of the public – of any written or pictorial materials which that individual, institution, group or government deems obscene and "utterly without redeeming social value," as determined by "contemporary community standards."

The definition is attributed to Chuck Stone Professor at the School of Journalism and Mass Communication, University of North Carolina. Please note that "Self Censorship" (not expressing something in fear of consequences) is a form of censorship too.

In our more technical we reduce the definition to

> Censorship: A systematic suppression, modification, or banning of data in a network by either removal, or modification of the data, or systematic influencing of entities involved in the processing (e.g., by creating, routing, storing, or reading) of this data.

This simplified definition narrows down the location to the Internet as it is the only relevant location for us. Furthermore, it limits the definition to the maximum reach within that system.

6.6.1 Censorship Resistant

A censorship-resistant system is a system that allows the entities of the system and the data itself to be unaffected from censorship. Please note that this does not deny the presence of censorship per se. It still exists outside the system. However, it has some consequences for the system itself.

- The system must be either undetectable or out of reach for an entity censoring.
  The possibility of identifying a protocol or data allows a censoring entity to suppress the use of the protocol itself.
- The entities involved in a system must be untraceable.
  Traceable entities would result in a mean of suppressing real-world entities participating in the system.

6.6.2 Parrot Circumvention

In [68] Houmansadr, Brubaker, and Shmatikov express that it is easy for a human to determine decoy traffic as the content is easily identifiable as generated content. While this is true, there is a possibility here to generate “human-like” data traffic to a certain extent. As an adversary may not assume that his messages are replied to, the problem does not boil down to a true Turing test. It remains on a “passive observer Turing test”, enabling the potential nodes to choose their messages.

In our design, this is the job covered by the blending layer. The blending layer generates these messages. These messages are context-less or remain in the context of previous conversations.

7 Related Cryptographic Theory and Algorithms

Whenever dealing with obfuscating data and maintaining the integrity of data, cryptography is the first tool in the hand of an implementer. A vast amount of research in this area does already exist. For this work, we focussed on algorithms either very well researched and implemented or research, which seem very valuable when putting this work into place.

In symmetric encryption in this paper always assumes that

$$D^{K_a}\left(E^{K_a}\left(\mathbf{M}\right)\right) = \mathbf{M} \tag{7.1}$$

For a key $K_b \neq K_a$ this means

$$D^{K_a}\left(E^{K_b}\left(\mathbf{M}\right)\right) \neq \mathbf{M} \tag{7.2}$$

$$D^{K_b}\left(E^{K_a}\left(\mathbf{M}\right)\right) \neq \mathbf{M} \tag{7.3}$$

The following candidates have been analyzed:

- AES
  NIST announced AES in 2001 as a result of a contest. The algorithm works with four operations (subBytes, ShiftRows, mixColumns, and addRoundKey). These operations are repeated depending on the key length 10 to 14 times.

  AES is up until now (2018) unbroken. It has been weakened in the analysis described in [151], which reduces the complexity by roughly one to two bits.

- Camellia
  The camellia algorithm is described in [93]. The key sizes are 128, 192, and 256. Camellia is a Feinstel cipher with 18 to 24 rounds depending on the key size. Up until today, no publication claims break this cipher.

For all asymmetric encryption algorithm in this paper, we may assume that...

$$D^{K_a^{-1}}\left(E^{K_a^1}(\mathbf{M})\right) = \mathbf{M} \tag{7.4}$$
$$D^{K_a^1}\left(E^{K_a^{-1}}(\mathbf{M})\right) = \mathbf{M} \tag{7.5}$$

It is important that

$$D^{K_a^{-1}}\left(E^{K_a^{-1}}(\mathbf{M})\right) \neq \mathbf{M} \tag{7.6}$$
$$D^{K_a^1}\left(E^{K_a^1}(\mathbf{M})\right) \neq \mathbf{M} \tag{7.7}$$

And for any other Keypair $K_a^p \neq K_b^p$

$$D^{K_b^{-1}}\left(E^{K_a^1}(\mathbf{M})\right) \neq \mathbf{M} \tag{7.8}$$
$$D^{K_b^1}\left(E^{K_a^1}(\mathbf{M})\right) \neq \mathbf{M} \tag{7.9}$$
$$D^{K_b^{-1}}\left(E^{K_a^{-1}}(\mathbf{M})\right) \neq \mathbf{M} \tag{7.10}$$
$$D^{K_b^1}\left(E^{K_a^{-1}}(\mathbf{M})\right) \neq \mathbf{M} \tag{7.11}$$

The number of crypto algorithms was higher than the steganography options. When looking for well-researched algorithms basing on different mathematical problems and having well-defined outlines, numbers dropped dramatically again.

- RSA
  In 1978 the authors Rivest, Shamir, and Adleman published with [124] a paper which did revolutionize cryptography for years. In their paper, the authors described an encryption method later to be called RSA, which required a key pair (K_a) referenced as public (K_a^1) and private keys (K_a^{-1}). The novelty of this system was that anything encrypted with the public key was only decryptable with the private key and vice versa.

  RSA is up until the day of writing this paper not publicly know to be broken (unless a too small key size is used). However – Shor described in 1997 an algorithm which should enable quantum computers to break RSA far faster than done with traditional computers. In the section 7.2 we do elaborate these effects further.

- ECC
  The elliptic curves were independently suggested by [98] and [74] in 1986. Elliptic curve cryptography started to be widely deployed in the public space in 2006. Since then, it seems to compete very well with the well established RSA algorithm. While being similarly well researched ECC, has the advantage of far shorter key sizes for the same grade of security.

- McEliece
  McEliece was first implemented and then removed again. The key size to

gain equivalent security to RSA1024 was $\approx 1MB$. This was impractical and thus discarded again. This was done, although there is up until now no known quantum capable algorithm reducing the key size of McEliece.

- NTRU
  In [65] Hoffstein, Pipher, and Silverman described the NTRU algorithm. The inclusion of this algorithm was disputed as it is patented in the united states as US7031468. It was included because the company Security Innovation holding the patent, released the NTRU algorithm on March [1] 2018 into the public domain according to a blog entry on the company website. While NTRU is not as well researched as RSA, it has been around for more than 20 years without being significantly affected by known attacks.

- ElGammal
  We rejected ElGamal as a cryptosystem to include. It bases on the same mathematical problems for cryptoanalysis as RSA (discreet logarithms) but is not as common as RSA.

Homomorphic encryption, as introduced in [47], was from the beginning a strong candidate to be used within our work. Unfortunately, we did not find a way to apply the core $addRedundancy$ operation in homomorphic encryption. Transforming the original data to the GF space in an efficient way to apply matrices was not doable and thus rejected.

7.1 Deniable Encryption and Steganography

Deniable encryption and steganography have been considered out-of-bounds for this work. The main reason is that the presence of encryption (which is not deniable in both cases) may be sufficient for a censor to block a message. Adding a layer to make sure that encryption or steganography is deniable, does not add valuable properties to our system as the sheer presence of encryption might be sufficient for censorship.

7.2 Key Sizes

The question of key sizes is hard to answer as it depends on the current and future possibilities of an adversary, which is again depending on not foreseeable research. We tried to collect a couple of recommendations.

Encrypt II (http://www.ecrypt.eu.org/) recommends currently for a “foreseeable future” 256 Bits for symmetric encryption and for asymmetric encryption based on factoring modulus 15424 Bits. Elliptic Curve Cryptography and Hashing should be sufficient if used with at least 512 Bits. If the focus is reduced to the next ≈ 20 years, then the key size recommendations are reduced to 128 Bit for symmetric

encryption, 3248 Bits for factoring modulus operations, and 256 Bits for elliptic curves and hashing.

According to the equations proposed by Lenstra in [81] an asymmetric key size of 2644 Bits respectively symmetric key length of 95 Bits, or 190 Bits for elliptic curves and hashing should be sufficient for security up to the year 2048.

According to [29] (superseding well known and often used [44]) data classified up to “top secret” should be signed with RSA 3072+ or ECDSA P-384. For symmetric encryption, they recommend AES 256 Bits, for Hashing at least SHA-384 and for Elliptic curves a 384 Bit sized key.

As it might seem not a wise idea to consider the recommendation of a potential state-sponsored adversary and the Formulas proposed by Lenstra do not explicitly take quantum computers into account, we follow the advice of ENCRYPT II.

Furthermore, taking all recommendations together, it seems that all involved parties assume the most trust in elliptic curves rather than asymmetric encryption based on factoring modulus.

7.3 Cipher Mode

The cipher mode defines how multiple blocks encrypted with the same key are handled. Main characteristics of cipher modes to us are:

- Parallelisable
  Can multiple parts of a plaintext be encrypted simultaneously? This feature is important for multi CPU and multi-core systems as they can handle parallelizable more efficiently by distributing them on multiple CPUs.
- Random access in decryption
  Random access on decryption allows efficient partial encryption of a ciphertext.
- Initialisation vector
  An initialization vector has downsides and advantages. On the downsides is the fact that an initialization vector must be shared with the message or before distributing it. It is essential to understand that the initialization vector itself usually is not treated as a secret. It is not part of the key.
- Authentication
  Authentication guarantees that the deciphered plaintext has been unmodified since encryption. It does not make a statement over the identity of the party encrypting the text. Such an identifying authentication is referred to as signcryption.

We evaluated the most common cipher modes for suitability. For *MessageVortex*, we focussed on modes that have the properties parallelizable, random access,

and do not do authentication. The main focus, besides the characteristics mentioned above, was on the question of whether there is an open implementation available in java, which is reasonably tested.

- ECB (Electronic Code Book)
  ECB is the most basic mode. Each block of the cleartext is encrypted on its own. This results in a big flaw: blocks containing the same data will always transform to the same ciphertext. This property makes it possible to see some structures of the plain text when looking at the ciphertext. This solution allows the parallelization of encryption, decryption, and random access while decrypting. Due to these flaws, we rejected this mode.

- CBC (Cypher Block Chaining)
  CBC extends the encryption by xor'ing an initialization vector into the first block before encrypting. For all subsequent blocks, the ciphertext result of the preceding block is taken as xor input. This solution does not allow parallelization of encryption, but decryption may be paralleled, and random access is possible. As another downside, CBC requires a shared initialization vector. As with most IV bound modes, an IV/key pair should not be used twice, which has implications for our protocol.

- PCBC (Propagation Cypher Block Chaining)
  CBC extends the encryption by xor'ing, not the ciphertext but a xor result of ciphertext and plaintext. This modification denies parallel decryption and random access compared to CBC.

- EAX
  EAX has been broken in 2012[99] and is therefore rejected for our use.

- CFB (Cypher Feedback) CFB is specified in [39] and works precisely as CBC with the difference that the plain text is xor'ed and the initialization vector, or the preceding cipher result is encrypted. CFB does not support parallel encryption as the ciphertext input from the preceding operation is required for an encryption round. CFB does, however, allow parallel decryption and random access.

- OFB
  [39] specifies OFB and works exactly as CFB except for the fact that not the ciphertext result is taken as feedback but the result of the encryption before xor'ing the plain text. This denies parallel encryption and decryption, as well as random access.

- OCB (Offset Codebook Mode)
  This mode was first proposed in [125] and later specified in [126]. OCB is specifically designed for AES128, AES192, and AES256. It supports authentication tag lengths of 128, 96, or 64 bits for each specified encryption algorithm. OCB hashes the plaintext of a message with a specialized function $H_{OCB}(\mathbf{M})$. OCB is fully parallelizable due to its internal structure. All blocks except the first and the last can be encrypted or decrypted in parallel.

- CTR
  CTR is specified in [87] and is a mixture between OFB and CBC. A nonce concatenated with a counter incrementing on every block is encrypted and then xor'ed with the plain text. This mode allows parallel decryption and encryption, as well as random access. Reusing IV/Key-pairs using CTR is a problem as we might derive the xor'ed product of two messages. This problem only applies where messages are not uniformly random such as in an already encrypted block.

- CCM
  Counter with CBC-MAC (CCM) is specified in [161]. It allows to pad and authenticate encrypted and unencrypted data. It furthermore requires a nonce for its operation. The size of the nonce is dependent on the number of octets in the length field. In the first 16 bytes of the message, the nonce and the message size is stored. For the encryption itself, CTR is used. It shares the same properties as CTR.

  It allows parallel decryption and encryption as well as random access.

- GCM (Galois Counter Mode)
  GCM has been defined in [95], and is related to CTR but has some major differences. The nonce is not used (just the counter starting with value 1). To authenticate the encryption, an authentication token $auth$ is hashed with H_{GFmult} and then xor'ed with the first cipher block. All subsequent cipher blocks are xor'ed with the previous result and then hashed again with H_{GFmult}. After the last block the output o is processed as follows: $H_{GFmult}(o \bigoplus (len(A)||len(B))) \bigoplus E^{K^0}(counter_0)$. As a result, GCM is not parallelizable and does not support random access.

  The mode has been analyzed security-wise in 2004 and showed no weaknesses in the analyzed fields [96].

  GCM supports parallel Encryption and decryption. Random access is possible. However, authentication of encryption is not parallelizable. The authentication makes it unsuitable for our purposes. Alternatively, we could use a fixed authentication string.

- XTS (XEX-based tweaked-codebook mode with ciphertext stealing)
  This mode is standardized in IEEE 1619-2007 (soon to be superseded). A rough overview of XTS may be found at [91]. It was developed initially for Disks offering random access and authentication at the same time.

- CMC (CBC-mask-CBC) and EME (ECB-mask-ECB)
  In [61] Halevi and Rogaway introduces a cipher mode which is extremely costly as it requires two encryptions. CMC is not parallelizable due to the underlying CBC mode, but EME is.

- LRW
  LRW is a tweakable narrow-block cipher mode described in [154]. This mode shares the same properties as EBC but without the weakness of the same

clear text block resulting in the same ciphertext. Similarly to XEX, it requires a tweak instead of an IV.

7.4 Summary of Cipher Modes

Mode \ Criteria	auth	Requires IV	parallelisable	random access
CBC	×	✓	×	×
CCM	×	✓	×	×
CFB	×	✓	✓	✓
CTR	×	✓	✓	✓
ECB	×	×	✓	✓
GCM	✓	✓	×	×
OCB	✓	×[1]	×	×
OFB	×	✓	×	×
PCBC	×	✓	×	×
XTS	×	✓[2]	✓	×
LRW	×	✓[2]	✓	✓
CMC	×	✓[2]	×	×
EME	×	✓[2]	✓	✓

Table 7.1: comparison of encryption modes in terms of the suitability

7.5 Padding

A plain text stream may have any length. Since we always encrypt in blocks of a fixed size, we need a mechanism to indicate how many bytes of the last encrypted block may be safely discarded.

Different paddings are used at the end of a cipher stream to indicate how many bytes belong to the decrypted stream.

7.5.0.1 RSAES-PKCS1-v1_5 and RSAES-OAEP

This padding is the older of the paddings standardized for PKCS1. It is basically a prefix of two bytes followed by a padding set of non zero bytes and then terminated by a zero byte and then followed by the message. This patting may give a clue if decryption was successful or not. RSAES-OAEP ist the newer of the two padding standards

7.5.0.2 PKCS7

This padding is the standard used in many places when applying symmetric encryption up to 256 bits key length. The free bytes in the last cipher block indicate

[1] included in auth

[2] Requires tweak instead of IV

the number of bytes being used. This makes this padding very compact. It requires only 1 Byte of functional data at the end of the block. All other bytes are defined but not needed.

7.5.0.3 OAEP with SHA and MGF1 padding

This padding is closely related to RSAES-OAEP padding. The hash size is, however, bigger, and thus, the required space for padding is much higher. OAEP with SHA and MGF1 Padding is used in asymmetric encryption only. Due to its size, it is important to note that the payload in the last block shrinks to $keySizeInBits/8 - 2 - MacSize/4$.

In our approach, we have chosen to allow these four paddings. The allowed sha sizes match the allowed mac sizes chosen above. It is important to note that padding costs space at the end of a stream. Since we are always using one block for signing, we have to take care that the chosen signing mac plus the bytes required for padding do not exceed the key size of the asymmetric encryption. While this usually is not a problem for RSA as there are keys 1024+ Bits required, it is an essential problem for ECC algorithms as there are much shorter keys needed to achieve an equivalent strength compared to RSA.

We have introduced an additional type of padding not related to these paddings. We required for the addRedundancy the following unique properties. Unfortunately, we were unable to find any padding which matched the following properties simultaneously:

- Padding must not leak successful decryption
  For our addRedundancy operation, we required padding that had no detectable structure as a node should not be able to tell whether a removeRedundancy operation did generate content or decoy.
- Padding of more than one block
  Due to the nature of the operation, it is required to be able to pad more than just one block.

Details of this padding are described in the section "Add and Remove Redundancy Operations" in A.

8 Censorship Circumvention

Several technical ways have been explored to circumvent censorship. All seem to boil down to the following main ideas:

- Hide data
- Copy or distribute data to a vast amount of places to improve the lifespan of data

- Outcurve censorship measurements

In the following section, we look at technologies and ideas dealing with these circumvention technologies.

8.1 Covert Channel and Channel Exploitations

The original term of covert channels was defined by Lampson[78] as

> not intended for information transfer at all, such as the service program's effect on system load.

This was defined in such a way to distinguish the message flow from

> legitimate channels used by the confined service, such as the bill.

The use of a legitimate channel such as SMTP and hide information within this specific channel is not a usage of a covert channel. We refer to this as channel exploitation.

8.2 Steganography

Steganography is an important part when it comes to unlinking information. In [71] and [148] we get a very rough overview. As some of the types and algorithms address specific topics of steganography (e.g., some hide from automatic detection and others address a human message stream auditor), we need to choose carefully. In our specific case, the main idea is to hide within the sheer mass of Internet traffic. As a human auditor screening all the messages is a minor thread, we focus on machine-based censorship. Most of the images sent in SMTP are jpg images (see table 17.1 on page 106). We limited our search to algorithms capable of hiding binary data within these files. The number of academically researched options was surprisingly low.

After reviewing the options, we decided to go for F5[160]. It is a reasonably well-researched algorithm which attracted many researchers. The original F5 implementation had a detectable issue with artifacts[20] caused by the recompression of the image. This issue was caused only due to a problem in the reference implementation, and the researchers have provided a corrected reference implementation without the weakness.

YASS, as described in [144], was not considered a candidate. Although less researched, researchers found multiple weaknesses[75, 85].

8.3 Timing Channels

Timing channels are a specialized form of covert channels. In timing channels, the information itself hides not within the data of the channel, but the usage of the channel is in such a way that it is capable of reflecting the data. As we do not have control over the timing of the transport channel, this is not an option for us.

8.4 Technical Forms of Censorship

There are many types of censorship available within technical systems. An in-depth understanding of the possibilities is required in order to understand the means of a censoring adversary.

8.4.1 Making Systems Unavailable by Censoring Lookups

This is one of the cheapest method to create censorship. Lookup systems such as DNS servers are modified in such a way that traffic is no longer deliverable or redirected to a system controlled by the censorer.

Many jurisdictions have implemented such measures a very cheap measure of censorship. It is, however, very easy to outcurve. As soon as a user no longer uses adversary controlled lookup services, this form of censorship is ineffective. In the case of DNS this means either:

- Use a public DNS server worldwide available
- Use another protocol to hide the traffic
  - A legit protocol with tunnelling capabilities like SSH to a system outside of the reach of the censoring adversary.
  - Use a fully blown tunnel such as a VPN.
  - Piggyback a legit protocol such as DNS-over-HTTPS (DoH)[63] or DNS-over-XMPP[22]

8.4.2 Making Systems Unavailable by Disrupting System Traffic

Disruption of traffic is commonly done with packet filtering devices commonly referred as firewalls. These firewalls may filter any traffic to a given system. There are some considerable downsides to this system from the adversary's point of view.

First, it requires high bandwidth. All traffic of a jurisdiction or target must pass through this filtering device. This is usually doable for a country.

Secondly, the target must be identifiable on a technical level (e.g., IP address) as content based filtering is only feasible with unencrypted or weakly protected systems. This technical identification is hard as systems may change their addresses dynamically either due to cloud related elasticity or simply due to an incomplete view to a distributed system.

And lastly, The target may not be split (e.g., partial censoring). Censoring platforms such as Youtube or Gmail which are commonly shared among legit and illegit uses from the censorers point-of-view can not be reasonably split. This due to the fact that only the provider of the service can do selective censoring on the system.

8.4.3 Making Systems Unavailable by Interfering with System Traffic

8.5 Spread Spectrum in Networking Protocols

9 Other Related Concepts

9.1 Zero Trust

Zero trust is not an acdemically defined concept. In fact it is widely misused by many marketing departments of well known devices and applications related to security.

We however define zero trust as follows:

> We assume that any participant in a network not explicitely trusted may:
>
> - Make some or all information available to him available to others.
> - Analyze all information within his reach.
> - Willingly break protocol rules in order to gain information or other advantages.

As design principle this means that information is kept hidden as much as possible within the system.

Part III

Anonymous Communication Systems

***

It was the anonymity. He wanted to be unknown, unpossessed by others' knowledge of him. That was freedom.

Ling Ma, Severance

10 Well Known Standard Protocols

SMTP and Related Post Office Protocols (1982)

Today's mail transport is mostly done via SMTP protocol, as specified in [73]. This protocol has proven to be stable and reliable. Most of the messages are passed from an MUA to an SMTP relay of a provider. From there, the message is directly sent to the SMTP server of the recipient and subsequently to the server-based storage of the recipient. The recipient may, at any time, connect to his server-based storage and may optionally relocate the message to a client-based (local) storage. The delivery from the server storage to the MUA of the recipient may happen by message polling or by message push (whereas the latter is usually implemented by a push-pull mechanism).

To understand the routing of a mail, it is essential to understand the whole chain starting from a user(-agent) until arriving at the target user (and being read!). To simplify this, we used a consistent model that includes all components (server and clients). The figure 10.1 shows all involved parties of a typical mail routing. It is essential to understand that mail routing remains the same regardless of the client. However, the availability of a mail at its destination changes drastically depending on the type of client used. Furthermore, control of the mail flow and control is different depending on the client.

The model has three main players storage, agent, and service. Storages are endpoint facilities storing emails received. Not explicitly shown are temporary storages such as spooler queues or state storages. Agents are simple programs taking care of a specific job. Agents may be exchangeable by other similar agents. A service is a bundle of agents that is responsible for a specific task or task sets.

In the following paragraphs (for definitions), the term "email" is used synonymously to the term "Message". "Email" has been chosen over "messages" because of its frequent use in standard documents.

Emails are typically initiated by a Mail User Agent (MUA). An MUA accesses local email storage, which may be the server storage or a local copy. The local copy may be a cache only copy, the only existing storage (when emails are fetched and deleted from the server after retrieval), or a collected representation of multiple server storages (cache or authoritative).

Besides the MUA, the only other component accessing local email storage is the Mail Delivery Agent (MDA). An MDA is responsible for storing and fetching emails from the local mail storage. Emails destined for other accounts than the current one are forwarded to the MTA. Emails destined to a User are persistently stored in the local email storage. It is essential to understand that email storage does not necessarily reflect a single mailbox. It may as well represent multiple mailboxes (e.g., a rich client-serving multiple IMAP accounts) or a combined view of multiple accounts (e.g., a rich client collecting mail from multiple POP accounts). In the

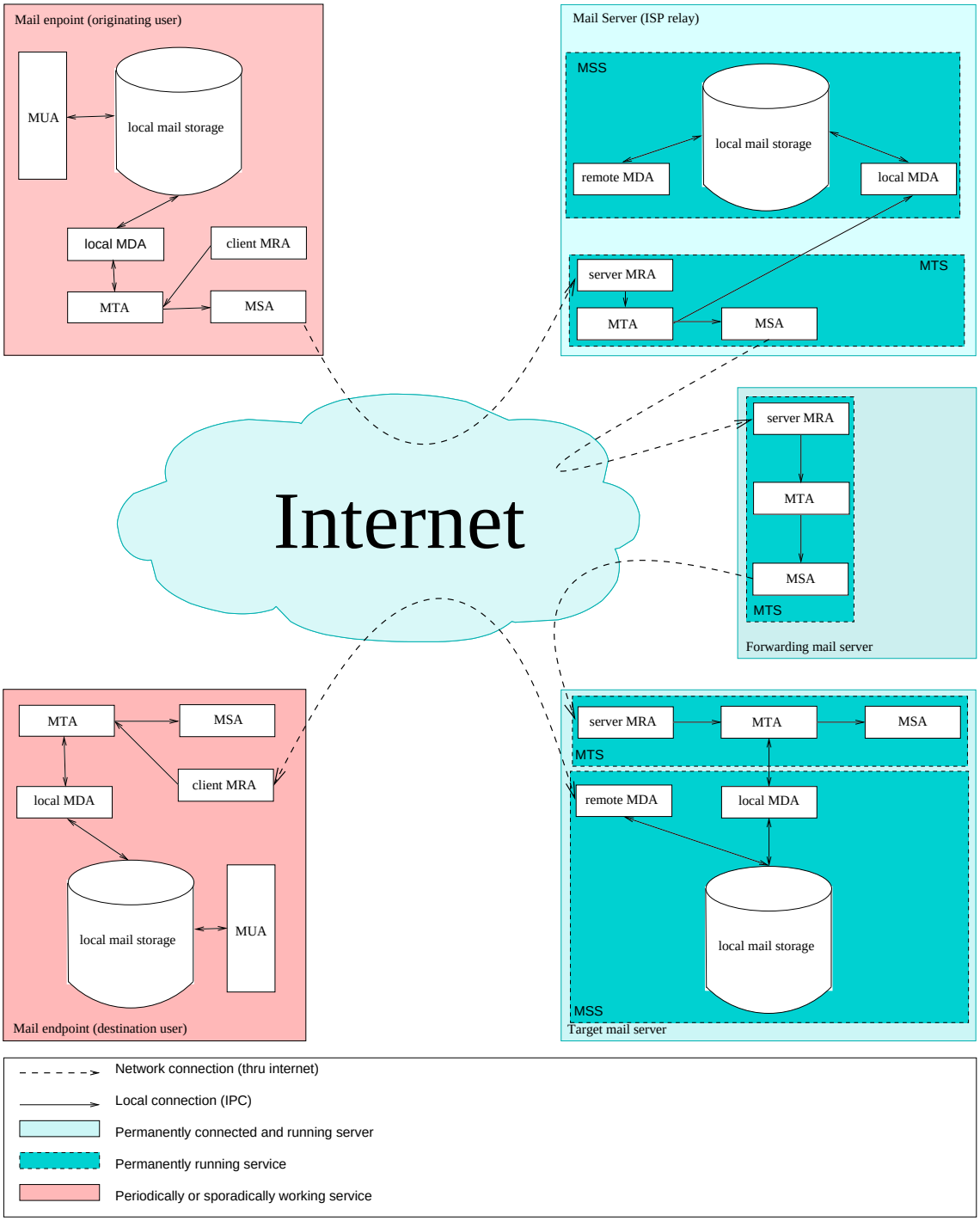


Figure 10.1: Mail Agents

case of a rich client, the local MDA is part of the software provided by the user agent. In the case of an email server, the local MDA is part of the local email store (not necessarily of the mail transport service).

On the server-side, there are usually two components (services) at work. A “Mail Transport Service” (MTS) responsible for mail transfers and a “Mail Storage System” which offers the possibility to store received Mails in a local, persistent store.

An MTS generally consists out of three parts. For incoming connects, there is a daemon called Mail Receiving Agent (Server MRA) is typically a SMTP listening daemon. A Mail Transfer Agent (MTA) which is responsible for routing, forwarding, and rewriting emails. Moreover, a Mail Sending Agent (MSA) which is responsible for transmitting emails reliably to another Server MRA (usually sent via SMTP).

An MSS consists of local storage and delivery agents which do offer uniform interfaces to access the local store. They do also deal with replication issues, and grant should take care of the atomicity of transactions committed to the storage. Typically there are two different kinds of MDAs. Local MDAs offer possibilities to access the store via efficient (non-network based) mechanisms (e.g., IPC or named sockets). This is usually done with a stripped-down protocol (e.g., LMTP). For remote agents there a publicly – network-based – agent available. Common Protocols for this Remote MDA include POP, IMAP, or MS-OXCMAPIHTTP.

Mail endpoints consist typically of the following components:

- A Mail User agent (MUA)
- A Local Mail storage (MUA)
- A Local Mail Delivery Agent (Local MDA)
- A Mail Transfer Agent (MTA)
- A Mail Sending Agent (MSA)
- A Mail Receiving Agent (MRA)

Only two of these components do have external interfaces. These are MSA and MRA. MSA usually uses SMTP as transport protocol. When doing so, there are a couple of specialties.

- Port number is 587 (specified in [56]).
  Although port numbers 25 and 465 are valid and do usually have the same capabilities, they are for mail routing between servers only. Mail endpoints should no longer use them.
- Connections are authenticated.
  Unlike a normal server-to-server (relay or final delivery) SMTP connections on port 25, clients should always be authenticated of some sort. This may be based on data provided by the user (e.g., username/password or certificate) or data identifying the sending system (e.g., IP address)[56]. Failure in doing authentication may result in this port being misused as a sender for UBM.

Mail User Agents (MUA) are the terminal endpoint of email delivery. Mail user agents may be implemented as fat clients on a desktop or mobile system or as an interface over a different generic protocol such as HTTP (Web Clients).

Server located clients are a special breed of fat clients. These clients share the properties of fat clients except for the fact that they do not connect to the server. The client application itself has to be run on the server where the mail storage persists. This makes delivery and communication with the server different. Instead of interfacing with an MSA and a client MDA, they may directly access the local mail storage on the server. On these systems, the local mail storage may be implemented as a database in a user-specific directory structure.

Fat clients

The majority of mail clients are fat clients. These clients score over the more centralistic organized web clients in the way that they may offer mail availability even if an Internet connection is not available (through client-specific local mail storage). They furthermore provide the possibility to collect emails from multiple sources and store them in the local storage. Unlike Mail servers, clients are assumed to be not always online. They may be offline most of the time. To guarantee the availability of a particular email address, a responsible mail server for a specific address collects all emails (the MSS does this) and provides a consolidated view onto the database when a client connects through a local or remote MDA.

As these clients vary heavily, it is mandatory for the MDA that they are well specified. Lack of doing so would result in massive interoperability problems. Most commonly the Protocols IMAP, POP and EWS are being used these days. For email delivery, the SMTP protocol is used.

Fat clients are commonly used on mobile devices. According to [23] in Aug 2012 the most typical fat email client was Apple Mail client on iOS devices (35.6%), followed by Outlook (20.14%), and Apple Mail (11%). *Email Client Market Share*[42] as a more recent source lists in February 2014 iOS devices with 37%, followed by Outlook (13%), and Google Android (9%).

Server located clients

Server located clients build an absolute minority. This kind of clients was common in the days of centralized hosts. An example for a Server Located Client is the Unix command “mail”. This client reads email storage from a file in the users home directory.

Web clients

Web clients are these days a common alternative to fat clients. Most big provider companies use their proprietary web client. According to [42] the most common web clients are "‘Gmail"’, "‘Outlook.com"’, and "‘Yahoo! Mail"’. All these Interfaces do not offer a kind of public plug-in interface. However, they do offer IMAP-interfaces. This important for a future generalistic approach to the problem.

S/MIME (1996)

S/MIME is an extension to the MIME standard. The MIME standard allows in simple text-oriented mails an alternate representation of the same content (e.g., as text and as HTML), or it allows to split a message into multiple parts that may be encoded. It is important to note that MIME encoding is only effective in the body part of a mail.

S/MIME, as described in [118], extends this standard with the possibility to encrypt mail content or to sign it. Practically this is achieved by either putting the encrypted part or the signature into an attachment. It is essential to know that this method leaks significant pieces of the data.

As the mail travels directly from sender to recipient, both involved parties are revealed. Neither message subject nor message size or frequency is hidden. This method does offer limited protection when assuming an adversary with interest in the message content only. It does not protect from the kind of adversary in our case.

The trust model is based on a centralistic approach involving generally trusted root certification authorities.

Pretty Good Privacy (1996)

Exactly as S/MIME, PGP[49] builds upon the base of MIME. Although the trust model in PGP is peer-based. The encryption technology does not significantly differ (as seen from the security model).

Like S/MIME, PGP does not offer anonymity. Sender and endpoints are known to all routing nodes. Depending on the version of PGP, some meta-information or parts of the message content such as subject line, the real name of the sender and receiver, message size is leaked.

A good thing to learn from PGP is that peer-based approaches are offering limited possibilities for trust. The trust in PGP is based on the peer review of users. This peer review may give an idea of how well verified the key of a user is.

11 Information Routing and Distribution for Anonymizing Protocols

Information routing and distribution is not a novelty in privacy research. Researchers arround the globe have searched for means of privacy. One good example was the patent in the intro of Almon B. Strowger[147]. More recent activities are the infamous "How to share a secret"[139], which used Lagrange polynomials to distribute shares of information accross multiple hosts for privacy.

A single polynomial would be attackable. Therefore Shamir applied a $\mod p$ operation to hide characteristics of a curve (as long as p is large and prime). The system had many problems which were addressed by subsequent work such as [153].

Lagrange polynomes form an important part when it comes to networking and privacy. They are commonly used in the form of Reed-Solomon codes for securing unreliable connections (e.g., [4]), distributing data [139].

Our approach will be to use Lagrange not primarily for distributing data but to generate unidentifiable decoy traffic. When applying a Lagrange polynomial to a message all factors contain parts of the original message. Given enough factors of the polynomial, anyone may reconstruct the original message. As a result an adversary is not able which parts of the traffic is decoy and which part is message as all parts have the potential to recover the original message.

11.1 Mixing

Mixes have been first introduced by “Untraceable Electronic Mail, Return, Addresses, and Digital Pseudonyms”[24] in 1981. The basic concept in a mix goes as follows. We do not send a message directly from the source to the target. Instead, we use a kind of proxy server or router in between which picks up the packet, anonymizes it, and forwards it either to the recipient or another mix. If we assume that we have at least three mixes cascaded, we then can conclude that:

- Only the first mix knows the true sender
- All intermediate mixes know neither the true sender nor the true recipient (as the data comes from mixes and is forwarded to other mixes)
- Only the last mix knows the final recipient.

This approach (in this simple form) has several downsides and weaknesses.

- In a low latency network, the message may be traced by analyzing the timing of a message.
- We can emphasize a path by replaying the same message multiple times (assuming we control an evil node), thus discovering at least the final recipient.
- If we can “tag” a message (with content or attribute), we then may be able to follow the message.

In 2003 Rennhard and Plattner analyzed the suitability for mixes as an anonymizing network for masses. They concluded that there are three possibilities to run mixes.

- Commercial, static MixNetworks
- Static MixNetworks operated by volunteers
- Dynamic MixNetworks

They concluded that in an ideal implementation, a dynamic mix network where every user is operating a mix is the most promising solution as static mixes always might be hunted by an adversary.

11.2 Anonymous Remailers

Remailers have been in use for quite some time. There are several classes of remailers, and all of them are somehow related to Mixnets. There are "types" of remailers defined. Although these "types" offer some hierarchy, none of the more advanced "types" seem to have more than one implementation in the wild.

Pseudonymous Remailers (also called Nym Servers) take a message and replace all information pointing to the original sender with a pseudonym. This pseudonym may be used as an answer address. The most well known pseudonymous remailer possibly was anon.penet.fi run by Johan Helsingius. This service has been forced several times to reveal a pseudonyms true identity before Johan Heösingius decided to shut it down. For a more in-depth discussion of Pseudonymous Remailers see 12.2.1

Cypherpunk remailers forward messages like pseudonymous remailers. Unlike pseudonymous remailers, Cypherpunk remailers decrypt a received message, and its content is forwarded without adding a pseudonym. A reply to such a message is not possible. They may, therefore, be regarded as an "decrypting reflector" or a "decrypting mix" and may be used to build an onion routing network for messages. For a more in-depth discussion of type-1-remailers, see section 12.2.2.

Mixmaster remailers are very similar to Cypherpunk remailers. Unlike them, Mixmaster remailers hide the messages, not in an own protocol, but use SMTP instead. While using SMTP as a transport layer, Cypherpunk remailers are custom (non-traditional mail) servers listening on port 25. For a more in-depth discussion of type-2-remailers, see section 12.2.4.

Mixminion remailers extend the model of Mixmaster remailers. They still use SMTP but introduce new concepts. New concepts in Mixminion remailers are:

- Single Use Reply Blocks (SURBs)
- Replay prevention
- Key rotation
- Exit poicies

- Dummy traffic

For a more in depth discussion of Mixminion remailers see section 12.2.8.

11.3 Onion Routing

Onion routing is a further development of the concept of mixes. In onion routers, every mix gets a message which is asymmetrically encrypted. By decrypting the message, he gets the name of the next-hop and the content which he has to forward. The main difference in this approach is that in traditional mix cascades, the mix decides about the next hop. In an onionised routing system, the message decides about the route it is taking.

Onionized messages typically have the problem of a constant size loss throughout the system. Some systems counter this effect, y separating the routing setup from the message path.

While tagging attacks are far harder (if we exclude side-channel attacks to break sender anonymity), the traditional attacks on mixes are still possible. So when an adversary is operating entry and exit nodes, it is straightforward for them to match the respective traffic.

One very well known onion routing network is Tor (https://www.torproject.org). For more information about Tor see section 12.2.6.

11.4 Garlic Routing

Grarlic routing is an improved form of onion routing. To stop onionized messages to continuosly loose contents through their way, a garlic router collects multiple, independent messages into one message before routing. This compensates for the “size loss effect” of onionized systems.

11.5 Crowds

Crowds is a network that offers anonymity within a local group. It works as follows:

- All users add themselves to a group by registering on a so-called “blender”.
- All users start a service (called JonDo).
- Every JonDo takes any received message (might be from him as well) and sends it with a 50% chance either to the correct recipient or to a randomly chosen destination

While crowds as specified in [120] does anonymize the sender from the recipient rather well, the system offers no protection from someone capable of monitoring crowds traffic. The system may, however, be easily attacked from within by introducing collaborating johndos. It has been further developed to D-Crowds [33], ADU/RADU [104], Freenet[28] and others.

Furthermore, the blender is aware of all JonDos and thus of particular interest for any observing or censoring adversary. Control of the blender enables an adversary to split the network into controllable parts, adding a high likelihood of discovering an original sender.

11.6 Mimic Routes

Mimics are a set of statical mixes which maintain a constant message flow between the static routes. If legitimate traffic arrives, the pseudo traffic is replaced by legitimate traffic. An outstanding observer is thus incapable of telling the difference between real traffic and dummy traffic.

If centralized mixes are used, the system lacks the same vulnerabilities of sizing and observing the exit nodes as all previously mentioned systems. If we assume that the sender and receiver operate a mixer by themselves, the system would no longer be susceptible to timing or sizing analyses. The mimic routes put a constant load onto the network. This bandwidth is lost and may not be reclaimed. It does not scale well as every new participant increases the need for mimic routes and creates (in the case of user mixes) a new mimic load. Furthermore, the mixes are easily identifiable as their characteristic data stream contrasts compared to other network service streams.

11.7 Distributed Hash Tables

11.8 Dining Cryptographer Networks

DC networks are based on the work “The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability” by Chaum[25]. In this work, Chaum describes a system allowing a one-bit transfer (The specific paper talks about the payment of a meal). Although all participants of the DC net are known, the system makes it unable to determine who has been sending a message. The message in a DC-Net is readable for anyone. This network has the downside that a cheating player may disrupt communication without being traceable.

Several attempts have been made to strengthen the proposal of Chaum[58, 159, 57, 31]. However, no one succeeded without introducing significant downsides on the privacy side.

12 Proposed Academic Protocols and System Implementations

12.1 Characteristics of Known Anonymity Implementations

Table 12.1 shows the previously analyzed protocols according to the classification scheme introduced in [141].

	Network Structure						Routing Information		Communication Model					Performance and Deployability				
		Connection		Symmetry							Node Selection							
	Topology	Direction	Synchronization	Roles	Hierarchy	Decentralization	Network view	Updating	Routing Type	Scheduling	Determinism	Selection set	selection probability	Latency	Communication mode	Implementation	Code availability	Context/application
MessageVortex	⊠	⟷	≅	• • •	···	○	◐	ϟ	•··	≡	✖	☺	⊛	H	⊠	✔	✔	@
Chaum Mixes[1]	⊠	⟶	≠	••	···	✖	●	✖	•··	≡	✔	Ⓐ	◎	H	⊠	✔	✖	@
Babel[1]	⊠	⟶	≠	••	···	⊙	●	✖	•·· ·•·	≡	✔	Ⓐ	⊛	H	⊠	✖	✖	@
Mixmaster[1]	⊠	⟶	≠	••	···	⊙	◐	✖	•··	≡	✖	Ⓐ	⊛	H	⊠	✔	✔	@
Mixminion[1]	⊠	⟶	≠	••	···	⊙	●	⊙	•··	≡	✖	Ⓐ	⊛	H	⊠	✔	✔	@
Tor[1]	⊏	⟷	≅	•·•·•	···	⊙	●	⊙	•··	≡	✖	🏴	◎	L	⊠	✔	✔	🧅
Crowds[1]	⊠	⟷	≠	• • •	···	⊙	●	ϟ	·•··	≡	✖	Ⓐ	⊛	L	⊠	✔	✖	🧅
MorphMix[1]	⊏	⟷	≠	• • •	···	⊙	◐	ϟ	·•··	≡	✖	🏴	∗	L	⊷	✔	✔	🧅
Tarzan[1]	◻	⟷	≠	• • •	···	○	●	ϟ	•··	≡	✖	🧅	⊛	L	⊷	✔	✔	🧅
Torsk[1]	⊏	⟷	≠	•·•·•	···	⊙	◐	ϟ	•··	≡	✖	🏴	⊛	L	⊷	✔	✖	🧅
AP3[1]	⊏	⟷	≠	• • •	···	○	◐	ϟ	·•··	≡	✖	Ⓐ	⊛	L	⊷	✖	✖	🧅@
Salsa[1]	⊏	⟷	≠	• • • •·•·•	···	○	◐	ϟ	·•··	≡	✖	Ⓐ	⊛	L	⊷	✔	✖	🧅
Chaum´s DCnet[1]	⊠	⟶	≠	• • •	···	✖	●	ϟ	📣	≡	✔	Ⓐ	◎	H	⊠	✖	✖	✉
Herbivore[1]	⊏	⟶	≠	• • •	✤	⊙	◐	ϟ	📣	≡	✔	🏴	◎	M	⊠	✔	✖	✉
Dissent in numbers[1]	⊏	⟶	≠	••	✤	⊙	◐	ϟ	📣	≡	✔	🏴	◎	H	⊠	✔	✔	✉
I^2P^1	◻	⟶	≠	• • •	···	○	●	⊙	•··	◇	✖	🏴	∗	L	⊷	✔	✔	🧅@
$\mathcal{P}^{51}$	⊏	⟶	≠	• • •	✤	⊙	◐	ϟ	📣	◇	✔	☺	◎	H	⊠	✔	✖	✉
Riffle	⊠	⟷	≅	•·•·•	✤	✖	●	✖	·•··	≡	✔	F						
Atom																		
Riposte																		
Pung																		
PIR																		
Karaoke																		
Loopix																		
Stadium																		
Vuvuzela																		
Alpenhorn																		

Table 12.1: Classification table for anonymization protocols

12.2 Resenders, Onion Routers, and MixNets

12.2.1 Pseudonymous Remailers (1981)

A pseudonymous remailer allows to reach people via a pseudonymous email address. On the remailing server all traces of the original sender are removed and a pseudonymous email is inserted instead. The foundation of these reamilers can be found in an early article of David Chaum[24].

One of the most famous remailers was the Penet remailer (anon.penet.fi). This remailer only lasted from 1993 to 1996 and was shut down after at least two compromises involving the Chruch of Scientology. Details of the closure can be found in [84].

12.2.2 Cypherpunk Remailers (approx. 1993)

With the failing of anon.penet.fi, it became clear that the weakest spot of a single server infrastructure the information stored on the server and the vulnerability of their owner. The new type-1-remailers score over the existing type-0-remailers by using encryption for the message. Most of the time PGP was used and custom programmed mail processors on systems to achieve the functionallity. It is unclear when first type-1-remailers were invented. Setting up a type-1-remailer was typically achieved by using procmail together with a small script calling PGP binaries and then sending the resulting message to the next recipient. By combining multiple type-1-remailers, an onion-like structure of the message was achievable.

This approach was promising, but it was still observable. An observation was possible by correlating the message sizes (e.g., strictly decreasing) and timing information. Furthermore, remailers were however still known and authorities were able to ban infrastructure and capable of monitoring their routing activities. Additionally, those remailers allowed to prosecute administrators of such systems.

12.2.3 Babel (1996)

Babel was an academic system defined in a paper by Gülcü and Tsudik in 1996[60]. It has been developed at IBM Zurich Research Laboratory. It was a mixing system using onionized addresses. The sender remains anonymous while he may provide a reply routing block called RPI. If both parties would like to remain anonymous, the RPI of the initiator is deployed in a forum thread. Anyone using this block adds an RPI for its address to the message.

This system has all the disadvantages of a system using MURBs. Traffic highlighting and similar attacks are possible.

12.2.4 Mixmaster-Remailers (1996)

Like Cypherpunk remailers, the Mixmaster remailers were working with onion-like encrypted messages. The protocol was based on Mix-Nets described by Chaum in [24] and further developed by L. Cotrell in 1996.

In contrast to type-1-remailers, the use of cascading systems to remail became systematic. The enduser used specialized software to build and send Mixmaster messages.

Mixmaster messages were still traceable by message size. Reply blocks were not supported by the system. A user had to know all Mixmaster nodes in order to use the system. The last node was typically an exit node sending the message in clear to the final recipient. This behavior still allowed the use of Usenet.

12.2.5 Crowds (1997)

12.2.6 Tor (2000)

Tor is one of the most common onion router networks these days and onionizes generic TCP streams. It is specified in [35]. It might be considered one of the most advanced networks since it has a considerable size, and much research has been done here.

According to [149] Tor is a network consisting of multiple onion routers. Each client first picks an entry node. Then it establishes an identity, gets a listing of relay servers, and chooses a path through multiple onion routers. The temporary identity links to such a path and should be changed on a regular base along with its identity. Transferring data works by splitting the data into equally sized cells of 512 bytes.

There is a centrally organized directory in the Tor network, knowing all tor relay servers. Any Tor relay server may be a directory server as well.

Many attacks involving the Tor networks have been discussed in the academic world such as [110, 11, 12, 14, 15, 34, 43] and some have even been exploited actively. In the best case, the people discovering the attacks did propose mitigation to the attack. Some of these mitigations flowed back into the protocol. Some general thoughts of the attacks should be emphasized here for treatment in our protocol.

Being an exit node may be a problem in some jurisdictions. In general, it seems to be accepted that routing traffic with unknown content (to the routing node) is not regarded as illegal per se. So by being unable to tell malicious or illegal traffic apart from legitimate traffic, this is not a problem. However – being an exit node can mean that unencrypted and illegal traffic is leaving the routing traffic. In this

specific case, operators of a relay node might fear legal prosecution. Tor nodes may proclaim themselves as " non-exit nodes" to avoid the possibility of legal prosecution.

Furthermore, several DoS-Attacks have been carried out to overload parts of the Tor network. Most of them do a bandwidth drain on the network layer.

Attacking anonymization has been done in several ways. First of all, the most common attack is a time-wise correlation of packets if in control of an entry and an exit node. A massive attack of this kind was published in 2014 and has been published on the Tor website (relay early traffic confirmation attack). This attack was possible because tor is a low latency network. Another attack is to identify routes through tor by statistically analyze the traffic density in the network between nodes. More theoretical attacks focus on the possibility of controlling the directory servers to guarantee that an entity may be deanonymized because it is using compromised routers.

Generally, the effectiveness of the monitoring of single nodes or whole networks is disputed. According to a study by Johnson et al. in 2013[69], a system in the scale of PRISM should be able to correlate traffic of 95% of the users within a "few days". Other sources based on the Snowden Papers claim that NSA was unable so far to de-anonymize users of Tor. However, since these papers referenced to "manual analysis", the statement may be disputed when looking at automated attacks as well.

It is, according to `https://www.torproject.org/docs/pluggable-transports`, impossible to use transborder Tor traffic in at least China, Uzbekistan, Iran, and Kazakstan. In censored countries, Tor offers so-called bridged Transports. Currently deployed transports in the standard Tor browser bundle package are obfs4, meek, FTE, and ScrambleSuit. Only meek is listed as working in China. Meek achieves this by hiding its traffic in a standard protocol (https).

[135] is an excellent survey listing recent developments and attacks within the Tor project.

12.2.7 I^2P (2001)

The name I^2P is a derived from "Invisible Internet Project" according to geti2p.net. The first binary relase on sourceforge dates from 2001. The system itself is comparable to Tor for its capabilities. Mayor differences are:

- P2P based
- Packet-switched routing (Tor is "circuit-switched")
- Different forward and backward routes (called tunnels)
- Works pseudonymously

- Supports TCP and UDP

I^2P has not attracted as much attention as Tor so far. So it is hard to judge upon its real qualities.

In 2011 Herrmann and Grothoff presented in [62] an attack. As I^2Ps security model is chosen based on IP addresses, the authors propose to use several cloud providers in different B-Class networks. By selectively flooding peers, an adversary may extract statistical information. The paper proposes an attack based on the heuristic performance-based peer selection. The main critics of the paper were that the peer selection might be influenced by an adversary enabling him to recover I^2P has not attracted as much attention as Tor so far. So it is hard to judge upon its real qualities.

In 2011 Herrmann and Grothoff presented in [62] an attack. As I^2Ps security model is chosen based on IP addresses, the authors propose to use several cloud providers in different B-Class networks. By selectively flooding peers, an adversary may extract statistical information. The paper proposes an attack based on the heuristic performance-based peer selection. The main critics of the paper were that the peer selection might be influenced by an adversary enabling him to recover data on a statistical base.

12.2.8 Mixminion-Remailers (2002)

Mixminion was the standard implementation of a type-3-remailer. It tried to address many issues previously not solved. A Mixminion router splits messages in equally sized chunks and supports SURBs. Furthermore, replay protection and key rotation were available. Unlike the previous remailer types, Mixminion was no longer using SMTP as the transport protocol. Instead, Mixminion introduced a new transport protocol. The sources of this remailer are available on GitHub under `https://github.com/mixminion/mixminion`.

As a received message had to be decoded by the final recipient. Therefore, the final recipient had to be aware of Mixminion system.

Mixminion-Networks have been privacy-wise criticised for the following things:

- Pseudonymous single use reply blocks are broken (Chapter 4.2 in [137]).
- Central directory of mixes.
- Too few users

According to `https://mixminion.net` the first release of the software was in December 2002. And has been discontinued in 2008. Since 2011 the sources are available on GitHub. There have been some forks in 2011 but at the moment all forks seem to be inactive since at least 2016 as there are no new commits.

12.2.9 $\mathcal{P}^5$ (2002)

The Peer-to-Peer Personal Privacy Protocol is defined in [140]. It provides sender-, receiver- and sender-receiver anonymity. According to the project page of $\mathcal{P}^5$, there is only a simulator available for the protocol.

The transport layer problematic has been wholly ignored. As there is no precise protocol specification but only a rough outline about the messaging and the crypto operations, $\mathcal{P}^5$ offers minimal possibilities for analysis.

12.2.10 AN.ON (2003)

AN.ON, as suggested in [46], is a mixing network. It generates messages in equally sized chunks and sends them in fixed time slots after random mixing. Its implementation is called JAP and may be found under https://anon.inf.tu-dresden.de/. JAP is many ways similar to the capabilities of Tor. The network was at the time of writing a lot smaller (10 JonDos compared to 6500 relays in the Tor network).

While the approach is both simple and effective, it is not suitable against a powerful adversary. First, an adversary may be able to snoop the forwarding when on the system. Second, due to the timing behavior, tunnels belonging to each other may be identified, and third, the package size information does leak as well.

12.2.11 AP3 (2004)

AP3, as defined in [100], is an anonymous communication system and very similar to crowds. It performs a random walk over a set of known nodes. Not all nodes are known to anyone, and all nodes are aware of the final recipient.

The system is susceptible to numerous attacks, as shown by [101], and does not withstand our adversary as the final recipient is known to the routing nodes.

12.2.12 Cashmere (2005)

Cashmere is specified in [164]. It defines a protocol for the use of chaum mixes. Unlike most of the protocols, the chaum mixes in cashmere are virtual. So-called relay groups represent them. Each mix in the relay group may be used as an equivalent mix to all other mixes in the same group.

This design means that the failure of one mix does not result in the non-delivery of a message.

No client implementation could be found on the nternet. The project homepage http://current.cs.ucsb.edu/projects/cashmere/ has not been updated since 2005. This suggests that this project is dead or sleeping.

12.2.13 SOR (2012)

SSH-based onion routing (SOR)[40] is blaming the complex and monocultural landscape of anonymizing software and proclaims a very simple approach based on onionized SSH tunnels.

12.2.14 Riffle (2016)

Riffle[77] is developed by MIT in Python and Go as an alternative to Tor addressing some of its flaws. Riffle servers are mixes collecting user information, shuffling it and sends it to the next mixes or targets. The shuffling is secured by a zero-knowledge-proof while the permutation itself is hidden.

The messages are sent in clusters whereas every client sends or receives data in every round (mimicking traffic). The sent blocks are padded to a fixed value to prevent size analysis.

So far the Riffle system has not attracted much interest in the academic world. While being extended, we were unable to find an attack for Riffle.

12.2.15 Atom (2016)

Atom[76] is an asynchronous anonymity service for small messages claiming to be scaleable and transferring up to a million tweet sized messages in 28 minutes. Its PoC implementation is written in Go and was tested by creating a serie of AWS based EC2 instances. It provides a broadcast primitive with limited reach by groupig its servers into small groups. All messages have equal length and groups organize all received messages in batches and distribute them to other server groups. This results in a mix cascade somehow similar to the Mixminion system. However, the system extends the mix cascades with a zero knowledge proof so that tampering may be discovered to a certain extent.

According to the paper many aspects of Atom remain unsolved. Key distribution is proposed to be done by trustworthy third party “directory authorities”. To remain anonymous at least one honest node per group is required. Identifying malicious users in Atom require a collaborative effort involving the publications of the private keys of the entry groups. Mallicious users a proposed to be blacklisted by the directory authorities.

As most of the other protocols Atom implements its own protocol making it susceptible to censorship.

12.2.16 SCION (2017)

SCION[111] is a clean slate Internet protocol. While SCION is not really an anonymizing protocol. It contains, however, many interesting features. Unlike with the traditional networks, we have the possibility of influencing the routing of data within SCION. Furthermore, with PHI[27] and Dovetail[136], SCION may feature strong and fast anonymity features.

Unfortunately, as this is a clean slate Internet design, it is not available commonly currently, and as it is easily identifiable, it enables easy censorship as the relevance is due to its current availability of no importance, and a censoring adversary may just ban and censor SCION entirely.

12.3 Distributed Hash Tables

12.3.1 Tarzan (2002)

Tarzan is a P2P IP protocol using UDP to communicate. It is specified in [52]. Tarzan nodes may be used to anonymize Internet traffic in general. An initiator on the original sender machines encapsulates traffic into a layered UDP package and sends the package through a mix like relayd's. The last relayd acts as an exit node. A replier may send answers the opposite way. Each relayd knows its next and previous relayd. To minimize the impact of observation, Tarzan forwards packets only every 20ms and features replay protection.

12.3.2 MorphMix (2002)

MorphMix is another mix network and specified in [121]. It was a circuit-based mix system for networking anonymity. The core of the network was collision detection. This detection has been circumvented by [150]. Since then, no new papers have been published, and the project seems to be dead.

12.3.3 Salsa (2008)

Salsa was proposed in [106] and described a circuit based anonymization pattern based on distributed hash tables (DHT). An implementation for Salsa is available, but it is not public. [101] claims that by combining active and passive attacks, anonymity can be compromised.

12.4 Dining Cryptographer Based Network

12.4.1 Herbivore (2003)

Herbivore is a network protocol designed by Goel et al. in [57]. It is based on the dining cryptographers paper[25]. At the time of writing, no herbivore client or an actual protocol implementation could be found on the Internet. Wikipedia lists Herbivore as “dormant or defunct”.

12.4.2 Dissent (2010)

Dissent is defined in [31]. It is an anonymity network based on DC-nets. A set of servers forms these DC-nets. At least one of the servers in the used net must be trustworthy, and none may be misbehaving. A server failure results in the stall of all message delivery using this server.

In an attempt to improve Dissent Wolinsky et al. introduced in [162] a modified version. This improved version mainly addresses scalability issues of the original design. Furthermore, some information leakage and scalability flaws in the original approach were addressed.

12.5 Broadcast and Multicast Networks

12.5.1 Hordes (2002)

Hordes was a multicast-based protocol for anonymity specified in [83]. Hordes used the abilities to handle multicast addresses of routers to generate a dynamic set of receivers and then sends messages to them. It assumes that a single observer or router does not know all participating peers.

This assumption is correct for a local observer. Unfortunately, it is not sufficient assuming an adversary as defined in this paper.

12.6 Distributed Storage Systems

12.6.1 Feenet (2000)

Freenet was initially designed to be a fully distributed data store[28]. Documents are stored in an encrypted form. Downloaders must know a document descriptor called CHK containing the file hash, the key, and some background about the crypto being used. A file is stored more or less redundantly based on the number

of accesses to a stored file. The primary goal of Freenet is to decouple authorship from a particular document. It furthermore provides fault-tolerant storage, which improves caching of a document if requested more often.

Precisely as I^2P, Freenet is not analyzed thoroughly by the scientific world.

The Freenet features two protocols FCPv2 acts as the client protocol for participating in the control of the Freenet storage. The Freenet client protocol allows us to insert and retrieve data, to query the network status, and to manage Freenet nodes directly connected to an own node. FCPv2 operates on port 9481, and blocking is thus easy, as it is a dedicated port.

The Freenet project seems to be under active development as pages about protocols were updated in the near past (Last update on the FCPv2 Page was July 5th 2016 at the time of writing).

12.6.2 Gnutella (2000)

Gnutella is not a protocol for the anonymity world in special. Instead, the Gnutella protocol implements a general file sharing on a Peer to peer base. This peer-to-peer approach is the most interesting aspect of Gnutella in this context. Furthermore, Gnutella has proven to be working with a large number of clients.

The current protocol specification may be found under http://rfc-gnutella.sourceforge.net/. While the Gnutella network is defunct. The approaches solving some of the peer-to-peer aspects were very interesting.

12.6.3 Gnutella2 (2002)

Despite its name, Gnutella2 is not the next generation of Gnutella. It was a fork in 2002 from the original Gnutella and has been developed in a different direction. The specification may be found on `http://g2.doxu.org`. Just as its predecessor, Gnutella2 seems to be dead. The last relevant update to the main site or its protocol is dated four years back.

12.7 Unknown (TBD)

12.7.1 Riposte (2015)

[30]

12.7.2 Pung (2016)

[9]

12.7.3 PIR (2018)

[8]

12.7.4 Karaoke (2018)

Karaoke[79] is a low latency messaging system offering an alternative to high latency systems such as Vuvuzela or Stadium. Karaoke claims to have a latency up to 10 times lower than Vuvuzela or stadium.

12.7.5 Loopix (2017)

[115]

12.7.6 Stadium (2017)

[156]

12.7.7 Vuvuzela (2015)

Vuvuzela was presented by Van Den Hooff et al. in [158]. It is a scaleable anonymity system offering a high throughput between millions of users. The system is available as a PoC implementation written in Go. An adversary knows immediatly what clients do use vuvuzela. He is however unable to match up communication peers over time. The Vuvuzela client software is available under and connects to a vuvuzela network forming a centralized infrastructure. A Vuvuzela infrastructure may handle according to its authors up to 10 million users with an average bandwith cost of 3.7KB/s per user.

12.7.8 Alpenhorn (2016)

[80] 

Part IV

The MessageVortex System

***

Thinking is the hardest work there is, which is probably the reason, so few engage in it.

Henry Ford, American industrialist and founder of Ford Motor Co.

In this part, we create a protocol called *MessageVortex*, enabling anonymous communication. Unlike most other academic attempts, we do this on the base of an adversary, which is capable of banning our technology. We, therefore, are not able to focus solely on the anonymity property. Instead, we first collect requirements for such a system in section 13. Based on these requirements, we explain our architectural concepts and decisions in section 14. We then build an outline of our protocol focusing on the protocol's main properties, without going too much into implementation details. In section 15, we describe the protocol and its key concepts in depth. We explain all aspects relevant to the academic solution without going into implementation details.

The details of the implementation, as well as their realization in infrastructure, are in part V. For operational concerns such as route building strategies, refer to part VI.

13 Requirements for an Anonymizing Protocol

In the following sections, we first define a threat model. We then elaborate on the main characteristics of the anonymizing protocol based on the threat model.

ID	Category	Short	Description
RQ1	System	Undetectable	Protocol nodes and their traffic should be undistinguishable from accepted nodes and traffic.
RQ2	System	Equal Nodes	All nodes of the system should have similar functions, capabilities, and behavior.
RQ3	System	Zero Trust	No trust should be imposed on any infrastructure unless it is the senders' or the recipients' infrastructure.
RQ4	System	Unlinkability	Message Requirement A message must not be linkable by an adversary to either a sender or a recipient.
RQ5	System	Anonymizing	A system must be able to anonymize sender and recipient at any point of the transport layer and any point within the system unless on the senders' or the recipients' node.
RQ6	System	Accounting	The system must be able to do accounting for an entity without being linked to a real identity.
RQ7	Message	Untagable	The message should be un-tagable (neither by a sender nor an involved intermediate node).
RQ8	Message	Unbugable	The message should be unbugable (neither by the sender nor by an involved intermediate node).
RQ9	Message	Unreplayable	A message or its behavior must not be replayable.
RQ10	Operational	Bootstrapping	The system must allow to bootstrap from a zero-knowledge or near-zero-knowledge point and extend the network on its own.
RQ11	Operational	Algorithmic variety	The system must be able to use multiple symmetric, asymmetric, and hashing algorithms to immediately fall back to a secure algorithm for all new messages if required.
RQ12	Operational	Easy handleable	The system must be usable without cryptographic know-how and with popular or common tools.
RQ13	Operational	Reliable	From a user's perspective, the system must act predictably. Messages handed over to the system should reach their destination in any case.
RQ14	Operational	Transparent	From a user's perspective, the system must act predictably. He can determine the state of a message at any given point in time.
RQ15	Operational	Available	A user must have access to a working system and its software and updates.
RQ16	Operational	Identifiable sender	A recipient of a message should be able to authenticate a sender of a message beyond a simple authentification.

Table 13.1: Summary table of requirements

These requirements listed in table 13.1 are the goal we would like to achieve. We will measure up for success or failure in section 34).

13.1 Threat Model

In this section, we define in this section two adversaries. The two adversaries are an "observing adversary (mainly doing spying) and a "censoring adversary" (Actively disrupting communication). While equal in their technical capabilities, they have different executive and legislative environments. This difference in adversaries is essential as the usage of our system differs in these two environments.

We refer to "jurisdiction" as a geographical area where a set of legal rules created by a single actor or a group of actors apply. These actors do have executive capabilities (e.g., police, army, or secret service) to enforce this set of legal rules.

We assume for our protocol that adversaries are state-sponsored actors or players of large organizations. We assume that these actors have high funding and elaborated capabilities either themselves or within reach of their sponsor. Actors may join forces with other actors as allies. However, achieving more than 50% on a world scale is excluded from our model. We always assume one or more actors with disjoint interests covering half of the network or more.

We assume the following goals for an adversary:

- An adversary may want to disrupt non-authorized communication.
- An adversary may wish to read any information passing through portions of the Internet.
- An adversary may wish to build and conserve information about individuals or groups of individuals of any aspect of their life.

To achieve these goals, we assume the following properties of our adversary:

- An adversary has elaborated technical know-how to attack any infrastructure. This attack may cover any attack favoring his goals, starting with exploiting weaknesses of popular software (e.g., buffer overflows or zero-day exploits) down to simple or elaborated (D)DoS attacks.
- An adversary may monitor traffic at any location in public networks within a jurisdiction.
- An adversary may modify routing information within a jurisdiction freely.
- An adversary may freely modify even cryptographically weak secured data where a single or a limited number of entities grant proof of authenticity or privacy.
- An adversary may inject or modify any data on the network of a jurisdiction.
- An adversary may create their nodes in a network. He may furthermore monitor their behavior and data flow without limitation.

- An adversary may force a limited number of other non-allied nodes to expose their data to him. For this assumption, we explicitly excluded actors with disjoint interests.

- An adversary may have similar access to resources as within its jurisdiction in a limited number of other jurisdictions.

As adversaries have different capabilities and goals, we should classify them among these boundaries as well. We, therefore, split up the adversaries into the following subclasses:

- A censoring adversary

- An observing adversary

This adversary describes a powerful state-sponsored actor with very high but not unlimited powers. It serves us as a worst-case adversary.

13.1.1 Observing Adversaries

This adversary behaves like a traditional spy. He collects and classifies information while typically hiding its activities. The adversary only observes traffic and tries to extract data from the system.

Unlike the case of a censoring adversary, we imply that in most of the cases no restrictions apply for the use of anonymizing technology from a jurisdictional point of view. If restrictions apply, then such an adversary should be classified as censoring adversary, as the technology is "censored." Such classification must be done in this case, regardless of whether the adversary only tries to collect information or not.

13.1.2 Censoring Adversaries

The primary goal of this adversary is censoring messages and opinions, not within his interests. He does this, regardless of whether the activities of censorship may be observed or not. Therefore, this adversary does not necessarily cloak its activities and typically bans censorship circumventing actions as illegal.

In such environments may k-anonymity, as specified in [3], not be sufficient for such an adversary. Instead, the *MessageVortex*system must hide all activities from such an adversary.

13.2 Required Properties for our Unobservable Protocol

In this section, we collect requirements for our system. We always first list a requirement and then explain why it is essential.

13.2.1 System requirements

RQ1 (Undetectable): *Protocol nodes and their traffic should be undistinguishable from accepted nodes and traffic.*

Users are unable to limit the route of our packets through named jurisdictions. Therefore, we must protect users of *MessageVortex*(users) from being subject to legal prosecution in any country. Therefore, these users need to be anonymous when sending or receiving messages. Unfortunately, most transport protocols (in fact, almost all of them such as SMTP, SMS, XMPP, or IP) use a globally unique identifier for senders and receivers, which are readable by any party which is capable of reading the packets (mainly the routing nodes).

Threat model in section 13.1, we defined the adversary as someone with superior access to the network and its infrastructure. Such an adversary might attack a message flow in several ways:

- Identify the sender.
- Identify the recipient.
- Identify other involved parties.
- Read messages passed or extract meta information.
- Disrupt or modify communication fully or partially. This may or may not include the possible identification of the traffic.

If users want to stay anonymous, they must protect our traffic from outside system influences. As we are unable to protect data from modification, we must hide the traffic of our application. In such a scenario, an adversary cannot block our traffic unless he is willing to disrupt communication entirely by disrupting communication of the transport protocol.

RQ2 (equal nodes): *All nodes of the system should have similar functions, capabilities, and behavior.*

This requirement protects all involved parties from possible legal prosecution. As we are unable to introduce our infrastructure or protocols, any categorization from outside or inside would lead to an information leak.

We have to assume that all actions taken by a potential adversary are not subject to legal prosecution. This assumption based on the fact that an adversary trying to establish censorship may be part of the government of jurisdiction. We may safely assume that there are legal exceptions in some jurisdictions for such entities. This means he may legally introduce nodes into our system.

To withstand an adversary outlined in section13.1, the messages sent even within the system requires to be unidentifiable by attributes or content. "Attributes" may include any meta information including, but not limited to, frequency, timing, message size, sender, protocol, ports, or recipient. If we want to guarantee that a node is not identifiable as an endpoint of a message, all involved nodes must carry out equivalent operations. As soon as we have differences between routing nodes and endpoints, we can identify participating persons at entry or exit nodes.

Furthermore, it must be impossible from an observing adversary to identify message endpoints. All nodes must look equal from the outside in terms of traffic, as well as in terms of their offered functions and behavior.

The term "Equal nodes" does not necessarily mean that nodes must be indistinguishable. It merely means that given the functions, capabilities, and behavior of a node, no further information can be deduced.

RQ3 (zero trust): *No trust should be imposed on any infrastructure unless it is the senders' or the recipients' infrastructure.*

The requirements above protect from an adversary outside the system. Looking from the inside, an adversary may have access to much more information. An adversary will likely create nodes in an open system. As a consequence, trust in infrastructure is minimal.

In our model, we will work with suspicion into the infrastructure. As every infrastructure node learns from each transaction (e.g., the usage of the network or size of messages), we have to minimize or ideally eradicate such information gains. The main problem is that we are unable to hide peer senders or recipients when routing messages. In jurisdictions where such infrastructure usage is illegal, we need to protect the presence of our routing messages from any party not trusted. Such hiding concludes that we need to be able to control which nodes are involved when sending messages. We refer to this concept as controllable trust.

In terms of trust, we have to conclude that:

1. We trust in infrastructure because it is under full control of either the sender or the recipient. If we are unable to trust these infrastructures, information may be leaked without any problems. So trusting these infrastructures is inevitable.
2. We should not trust all other infrastructure as an adversary can misuse data passed through it.

RQ4 (unlinkability): *A message must not be linkable by an adversary to either a sender or a recipient.*

We need a requirement guaranteeing the unlinkability between the sender and recipient from an adversary point of view. This prevents building social graphs and narrowing down groups of individuals.

RQ5 (anonymization): *A system must be able to anonymize sender and recipient at any point of the transport layer and any point within the system unless on the senders' or the recipients' node.*

Unobservability requires, according to [114], that an item of interest (IoI) is undetectable from an uninvolved entity and anonymous for the involved entities. We therefore require:

As a result of the architecture of today's common networks, the anonymization of a sender or a receiver is not simple. A relay may allow at least the anonymization of the original sender given trust into the proxy. By combining it with encryption, we may even achieve a simple form of a sender and receiver pseudonymity, even for a weak outside observer. This has been done in cypherpunk remailers (see section 12.2). If cascading more relay like infrastructures and combining it with cryptography, we may achieve sender and receiver anonymity. When introducing anonymous remailing endpoints, we may additionally achieve both simultaneously. These are the standard approaches in remailers and mixes. We have seen real-world attacks on such systems in the past, and some were successful (e.g., [84]).

[114] defines anonymity as:

> Anonymity of a subject means that the subject is not identifiable within a set of subjects, the anonymity set.

If we apply our threat model, we find that we require all users to be anonymous, regardless of whether a specific user is sending messages or not. Otherwise, such a user may become subject to legal prosecution.

RQ6 (accounting): *The system must be able to do accounting for an entity without being linked to a real identity.*

As a system may be flooded with messages, we need means to control the burden of processed messages. To separate message flows, we need means to control it by identity. Unlike other protocols, we have no identifier as we work based on the previous requirement anonymously. However: We will need some type of accounting to restrict single attackers in flooding.

13.2.2 Message Requirements

From the message point-of-view, we need to conserve privacy, which has been built up in the previous section.

RQ7 (untagable): *The message should be un-tagable (neither by a sender nor by an involved intermediate node).*

To protect a message from being followed or observed, a message needs to have certain properties. First, a message should not have, by design, any properties which can be observed when passing through the system. Any node should remove all parts which were under the control of the previous node.

RQ3 (zero trust) implies that a node may try to introduce such features into the message. As we cannot keep a node from doing so, we can define that such tags must be removed by the next node.

RQ8 (unbugable): *The message should be unbugable (neither by the sender nor by an involved intermediate node).*

Another way of breaking anonymity is that, instead of following a message through the system, an adversary may modify (bug) it so that the receiving or any intermediate node leaks its presence. In traditional messaging such bugging is done by introducing remotely hosted data or by introducing revocable certificate operations into the message stream and then observe the VA of a PKI for respective OCSP calls or CRL accesses. Our protocol handling must not depend on such external lookup mechanisms to ensure that bugging is not possible:

As with RQ7 (untagable), a recipient or an intermediate node may be identified by a download or lookup behavior of a recipient or any intermediate routing node.

RQ9 (unreplayable): *A message or its behavior must not be replayable.*

In a generic sense a node may also replay a message to highlight a messages property (e.g., the path or a size), which leads us to the following requirement:

13.2.3 Operational Requirements

In order to be realistically operated, our system needs to fulfill some additional requirements.

RQ10 (bootstrapping): *The system must allow to bootstrap from a zero-knowledge or near-zero-knowledge point and extend the network on its own.* Up until here, we described a system that is not centrally controlled. If not relying on broadcast domains, which is not feasible on a global scale, each node needs to

know other nodes that may be contacted for routing purposes. We refer to the initial process of collecting routing nodes as bootstrapping.

RQ11 (algorithmic variety): *The system must be able to use multiple symmetric, asymmetric, and hashing algorithms to immediately fall back to a secure algorithm for all new messages if required.*

It is quite common that weaknesses in algorithms are discovered. We may, therefore, not rely on a single algorithm. Instead, we must create a protocol supporting crypto agility, as described in [21].

RQ12 (easy handleable): *The system must be usable without cryptographic know-how and with popular or common tools.*

Academic systems are usually not known for focusing on user-friendliness. Users, on the other hand, are not known for their willingness to give up a lot of functionality or usability in trade for security. If we want that our system is used, many users need to use the system. This would lower the bar for bootstrapping and increase the size of anonymity sets. We, therefore, conclude that the system must be easy to handle for a user. Usually, this would be a decision related to a GUI or an end-user application but not to a system. However, if we want our system to be easy to handle, we need to take this as a requirement into account.

RQ13 (reliable): *From a user's perspective, the system must act in a predictable manner. Messages handed over to the system should reach their destination in any case.*

Any message-sending protocol needs to be reliable in its functionality. If the means of message transport are unreliable, users tend to use different means for communication[163].

RQ14 (transparent): *From a user's perspective, the system must act in a predictable manner. He is able to determine the state of a message at any given point in time.*

Transparent behavior is a prerequisite for reliability. If something is generating a behavior, but a user is unable to determine the reason for it (i.e., if a user is expecting a different behavior), he usually assumes a malfunction. Therefore, "reliable" means not only stable by its behavior. It also means diagnosable. A user's perception will not be "reliable" if he is not able to determine causes for differences in observed and expected behavior (e.g., [107]).

RQ15 (available): *A user must have access to a working system and its software and updates.*

If a user should be able to use the system, he needs access to other nodes and the required software, as well as its updates. This has to be considered even in an environment with a censoring adversary. So the system needs to be available.

Availability, in this specific context, may have two meanings, which do differ. A system is available if...

1. a sender and a recipient have (or may have) the means of using it.
2. the infrastructure provides the service (as opposed to: "is running in a degraded or faulty state and, therefore, unable to provide the service").

RQ16 (identifiable sender): *A recipient of a message should be able to authenticate a sender of a message beyond a simple authentification.*

A messaging system offering unlinkability may offer sender anonymity from a recipient's perspective. If so, a sender should be identifiable in such a way, that a classification of senders is possible for a legit recipient, and impersonation is not achievable. It is important to understand that an identifiable sender does not necessarily mean that users can identify a sender as a specific party. It only means that two senders may be identified as the same sender.

14 Rationale

In this chapter, we set the course for our system. We explain why we built the protocol the way it is. We try to elaborate on our decisions and explain why the system is not built differently.

The system we describe is a four-layered system (transport, blending, routing, and accounting layer) in which each layer fulfills a specific duty. The transport layer is equal to an unmodified, common internet data transport protocol. The blending layer inserts and extracts our protocol messages into the transport layer. The routing layer does disassemble and reassemble the messages received and applies specially crafted operations, and the accounting layer keeps track of quotas and protects the resources of the system. The three *MessageVortex* layers (all layers except "transport") run on common internet end-user devices such as mobile phones or tablets.

14.1 System Design and Infrastructure

All anonymity systems listed in part III have in common that they do rely on dedicated servers providing an anonymity related service. Such specialized servers make operators of such servers vulnerable in an environment where a censoring adversary (as described in section 13.1) exists. Our approach should, therefore,

be different. Instead of creating our own protocol, we describe a system where we use preexisting standard servers without modification. If we succeed to piggyback such a protocol invisibly, we may inherit the regular usage of this infrastructure as decoy traffic. Piggybacking and mimicking of protocols is not new. Protocols such as Skypemorph[103] or pluggable transports for Tor (e.g., Meek, FTE, or OBFS4) use this technology successfully for censorship circumvention.

We will do this piggybacking in a protocol-agnostic manner. On the protocol level, this requires that we separate embedding of messages into the transport protocol from the rest of the system. To do this in such a way makes the system even harder to observe as routing graphs taking multiple protocols into account increase the complexity exponentially through their different properties.

The content of the message in the transport layer protocol is provided by the routing node and not by anyone else. This restriction is based on the fact that if we allow anyone else except the routing node itself to control visible aspects of the transport layer message, the system could be misused for sending transport layer messages. To give an example: Such a system could be misused for blackmailing of a user not participating in the system. We simply create a message obfuscating the source and then exit the system by embedding the true blackmailing message.

As we rely on third-party infrastructure with our approach, now we have to take special care when designing our approach not to violate requirement RQ3 (zero trust). For obvious reasons, a direct connection between the sender and recipient via any named transport protocol would violate the requirement RQ4 (unlinkability). A single intermediate node would minimally imply trust in this node and its anonymization capabilities, which is not acceptable due to the requirement RQ3 (zero trust). When using multiple nodes, other anonymization protocols typically use three to five intermediate nodes due to their arguing. Typically we have at least three anonymization nodes for obvious reasons and sometimes an entry and exit node summing up to five nodes. This implies that the routing of our protocol is required. As we have an RQ3 (zero trust) policy, decisions for routing may no longer take place on the routing node but must be dictated externally.

For routing, we will use end-user devices. This decision is further backed by the requirement RQ2 (equal nodes). It, however, opposes the requirement of RQ13 (reliable), as such system participants are likely to be unreliable due to missing network connectivity, device failure due to drained batteries, or simply because they no longer participate in a network. To counter this, we implement measures on the message level.

14.2 Message and Routing

One of the biggest weaknesses of all protocols is the information leakage they have by design and the inability to restrict access to their functionality. We will build the messages with the following design guidelines:

- No routing controlled content shall survive a hop.
  For us, this means that by design, a message is received and dismantled. Any content visible or manipulable by the previous node must be removed. Only new content or content inaccessible to the previous node may be used to build new messages. Following this criterion, we automatically fulfill the requirement RQ7 (untagable).

- A routing node may efficiently identify a message sender.
  The sender must be efficiently identifiable. At first sight, this requirement is non-fitting as it opposes heavily to RQ5 (anonymization) and RQ4 (unlinkability). On the other hand, not providing these means makes it next to impossible to create a system that may not be misused and flooded. As the identification is pseudonymous, it must be short-lived, and multiple identities of the same sender must not be linked to each other. We will refer to this identity as an ephemeral ID (eID). This eID is handled in such a way that not complete decoding of the message is required to authorize the user. Instead, we build a message in such a way that tamper-proof, small-sized parts of the message are decoded first, and possibly bloated message content may be decoded after it is clear that the content is acceptable. If we assign "costs" to the creation of eIDs, it effectively protects the system from flooding.

- The routing operations must not leak more information than the next hop.
  We will apply a transformation on each routing hop to the message. This prevents following the message throughout the system. In most of the systems, messages are mainly disassembled and reassembled or onionized. The traffic is then cloaked in mimic traffic, making it impossible for an outside observer to identify message flows. The node generating the traffic is, however, well aware of the true message flow. In our system, we will add, instead of mimicking traffic, redundancy information (or remove it). By doing so, a routing node has no longer the insight which part of the traffic is relevant to the message and which not. We may, furthermore, introduce the possibility of distributing the message content throughout multiple paths in such a way that each path has not sufficient content to rebuild the message. In fact, depending on the complementing missing message, any content may be valid.

- Messages are protected from being replayed.
  In former systems, message paths were highlighted by injecting additional information. Our system is already protected from such injections by the eID concept, which identifies the sender. There are, however, other means to highlight traffic. An adversary may either inject message payload (corrupting the message flow) or replay the message. While we cannot keep anyone from sticking to rules, we may at least implement replay protection. Furthermore, we may later observe that we are able to control willingly induced, size mismatching content.

- Messages in the routing system are "store and forward" .
  All synchronous routing systems have in common that message observation

is relatively easy for an outside or inside observer unless mimick routes are used. This is why we allow the message to be stored and picked up or sent at a later stage.

- Use the Reed-Solomon-Function as the main routing operation.
  Originally [119] introduced a system allowing the use of polynomes to create error-correcting codes. In [26] Chaum, Crépeau, and Damgard, they have shown that the codes are suitable for distributing data assuming enough parties are honest and not malfunctioning. Unlike Chaum, Crépeau, and Damgard proposition, we are not using the Reed Solomon function to achieve anonymity or privacy. Instead, we use it for decoy traffic generation. We are splitting a message into multiple parts at several points when routing and assemble it again on different nodes. By doing so, we achieve two vital things. First, we introduce the possibility of recovering errors due to misbehaving nodes, and secondly, the real traffic can no longer be differentiated from decoy traffic. The operation has proven to be insecure as it leaks properties of the operation applied in the result. We will show this in section 28.3.1.3. Thus, We hardened the operation to remove the negative effects.

- *MessageVortex* must provide a variety of algorithms and operations to build a message.
  As all systems and algorithms applied to the system may be weakened or fail, a system needs to have the possibility to choose from multiple algorithms, protocols, and infrastructures. This choice should be made by a trustworthy system that restricts us to either the sender or the receiving system. The German Federal Office for Information Security (BSI) makes in [21] recommendations for systems and protocols, which we will follow.

  The main text can be boiled down to the following recommendations:

  - A protocol or system should be crypto agile.
  - A protocol or system should use hash-based signatures for updates.
  - The document furthermore recommends using symmetrical keys with a key length of 128 bit or more.
  - The document recommends a combination of big long term keys and small short term keys.
  - The document recommends using a combination of multiple independent algorithms in cascaded forms so that if one algorithm fails, the other one is still able to protect the data.
  - For key exchange, BSI recommends lattice-based cryptography.

14.3 Summarizing Chosen Approaches for *MessageVortex*

In this section, we have taken the following decisions for *MessageVortex*:

- Piggybacks common protocols.
- Does not require specialized hardware within the Internet.
- No proprietary systems on the Internet.
- Runs on commodity hardware.
- Sends messages in an asynchronous mode.
- Creates unidentifiable decoy traffic by using a Reed-Solomon-Function.
- Have no strict message size and avoid strictly increasing or decreasing sizes in any type of message or message part.
- Does not enforce specific attributes such as transport protocol, message size, message timing, or providers.
- Run special routing operations instead of traditional mixing and recombination methods.
- Offer a choice of algorithms to use when routing.

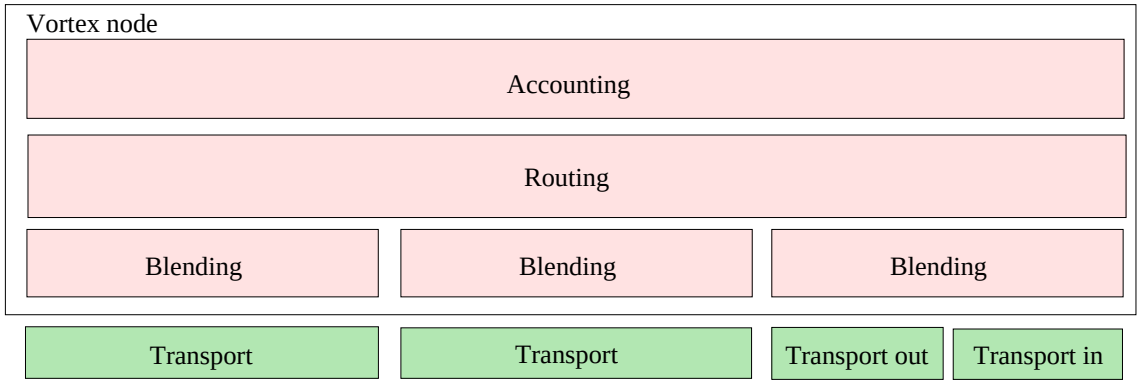


Figure 14.1: The protocol layers

The protocol is a four-layer protocol, as shown in figure 14.1. We communicate with standard protocols, which we refer to as the transport layer. While included in the message flow, they do not form a part of the *VortexNode*. The *VortexNode* itself consists out of the three layers “Blending”, “Routing”, and “Accounting”.

The blending layer is the bridging part linking a transport layer to the *VortexNode*. It injects and extracts messages from the transport layer and passes the extracted messages to the routing layer. It may be either used as a protocol bridge (e.g., in the case of XMPP) or act as a sophisticated router (e.g., in the case of the email protocols, where mails are fetched or received on push event via POP3 or IMAP while sending messages using SMTP).

The routing layer receives unified standard messages from the blending layer processes them, possibly extracts messages for local delivery, and passes subsequently created messages to the blending layer.

This design is definitely implementable on a consumer device. On the other hand, it is scalable and suitable for a clustered environment. Blending can be done in a stateless manner, is even suitable for serverless computing, and thus largely scalable. Routing may be implemented either with horizontal partitioning along with a set of eIDs or on a serverless base with a unified storage in the background. The accounting layer acts as a controller and may be implemented as well as stateless service with a minimal NOSQL-storage for all eIDs.

15 Protocol

MessageVortex is a protocol piggybacking standard transport protocols somehow similar to S/MIME[41] or PGP[55]. Unlike these protocols, we need the capability to keep the presence of our messages secret. The message itself should only be visible to an intended node.*MessageVortex* itself is agnostic to the transport, but we do require appropriate blending to hide within the transport protocol. The information processed on a node and its associated meta-information should not leak any information about the processed message.

Our system sends so-called *VortexMessages*. These messages are hidden within a transport protocol (e.g., SMTP or XMPP) with a blending mechanism (e.g., the steganographic algorithm F5) and extracted by a blending layer. The extracted *VortexMessage* is handed over to a routing layer. The *VortexMessage* itself contains a header block, a routing block, and possibly some payload blocks. The header block contains all the information required to protect the system. The routing block contains instructions (so-called “operations”) how the payload blocks have to be processed and where to send the resulting blocks. Those operations are one of the keys as information leaking happens in this step in most of the systems. We, therefore, crafted all operations very carefully to keep as much information secret as we could. These operations are the key to the system as they allow us to increase and decrease the size of a message without leaking what part of the data is a decoy and what not.

A payload may either be kept by the system for later processing with other messages, processed (possibly with different) payload blocks, or displayed to the "local user" as a message.

The general idea of the protocol is to form a network from nodes that mix and route messages between the sender and receiver. A routing block builder (RBB), which is typically identical to the sender, has full control over almost all attributes of the message, and nodes are unable to learn anything from the message while routing. Each user has a node, and there may be additional nodes (public routing nodes) without a user connected to it.

The message is either onion-like encrypted, split into parts and remerged, or

blown up with redundancy information.

This behavior results in a mixing like a system with a decoy generation in which even decoy generating nodes are unable to differentiate between real traffic and decoy as all blocks always contain parts of the message. Routing decisions are controlled by the builder of the routing block, and redundancy is possible and controlled by the routing block builder to make the system more stable.

In the following sections, we describe this protocol in detail. First, we build a terminology implicitly used in the previous chapters. Then we describe the key concepts and techniques of the protocol without in-depth analysis or reasoning. Implementation and operational aspects are discussed in part V and part VI.

15.1 Protocol Terminology

For our protocol, we use the following terms:

- **sender:** The user or process originally composing the message.
- **recipient:** The user or process destined to receive the message in the end.
- **user:** Any entity, running a *MessageVortex* node.
- **router:** Any node which is processing the message. Please note that all *VortexNode*s are routers. This includes the senders' and recipients' node.
- **message:** The "real content" to be transferred from the sender to the recipient.
- **VortexMessage:** The encoded message passed from one node to another one. The *VortexMessage* is considered before any embedding takes place. If embedded, we refer to such a message as "embedded *VortexMessage*".
- **payload:** Any data transported in a *VortexMessage* between routers with exception to the routing and header block, regardless of the meaningfulness or relevance to the *VortexMessage*.
- **decoy traffic:** Any payload transported between routers that have no relevance to the message at the final destination.
- **identity:** A tuple of a routable address and a public key. This tuple is a long-living tuple but may be exchanged from time to time. An Identity is always assigned to a node, but one node may have multiple identities.
- **eID:** An identity created on any node with a limited lifetime anyone possessing the private key (proven by encrypting with it) is accepted as representative of that identity.
- **Routing Block Builder (RBB):** An entity, which is building a routing block. Typically identical to either sender or recipient.

15.2 Key Components

The following sections list some of the key components of the system. Their understanding is essential for the understanding of the protocol as a whole.

We first describe a single node and its identity. This node is always equivalent to a potential recipient.

We then introduce the concept of workspaces and ephemeral identities. These concepts are essential for the routing and accounting layers. They dictate memory and storage requirements and lay a foundation for the routing layer.

A node always consists of three layers and one or more transports connected to it. The understanding of their inner workings is essential to the understanding of the project as a whole. We emphasize on their main function and their inner workings without going into implementation details. These details are further discussed in part V. We mainly focus on the data and the high-level processing done within these layers.

15.2.1 Nodes and their identities

We refer to a *VortexNode* (node) as a system run by an individual containing a processing software processing *VortexMessages*. Each node is connected to a transport layer protocol service (e.g., an IMAPv4 server as an endpoint for email or an XMPP server).

Each node o has at least one identity reflected by an asymmetric key pair K_{host_o}. Any node p communicating with node o must have the public key $K^1_{host_o}$ of the node.

A node requires the key $K^1_{host_o}$ to encrypt a message for node o. This key know-how enables environments with censoring adversaries to withstand probing attacks, as, without the knowledge of such keys, no reply from a node is received. The transport endpoint itself is not secret. The usage as *VortexNode*, however, is kept secret as long as the key is not known.

The protocol itself has the possibility to answer clear-text requests. So-called "public nodes" (see 23.3.2) make use of such messages. They are, however, an exception. In general, all *VortexMessages* are encrypted.

15.2.2 Workspaces and Ephemeral Identities

We dumped the approach for a system global unified storage for all message processing as such a design would allow an adversary to flood our storage. Instead, we introduced temporary storages suitable for a set of transaction belonging to a single identity or limited set of entities which collaborate. In our system, every transaction on a nose is assigned to an ephemeral identity (eID). An eID has a

limited lifetime and is represented by an asymmetric key pair and has to be created on each *VortexNode* taking part in a message processing. Each eID has a storage assigned to which we refer as “workspace”.

An eID is unique on every host and created on each *VortexNode* by the routing block builder (RBB). To create an eID, an RBB first sends a message with only a header block to the respective *VortexNode*. The request contains the new identity, a reply block, and a request to create a new identity. The receiving *VortexNode*will then typically sends a challenge back. A challenge may be the start of a hash bit sequence (also referred to as "puzzle"). The requester has then to resend the request with a header block. The requester must insert additional data in such a way that the start hash in its binary form matches the bit sequence provided. Another possibility is to request a payment in cryptocurrency. This allows us to commercialize routers in some countries where the usage of such routers is generally allowed.

The length of the requested bit sequence is chosen by the accounting layer at its own will. If the request is not answered in a given time, the eID will be discarded. Analogous to an SYN-Flood attack, an adversary may try to overwhelm a *VortexNode* with eID creation requests. Such flooding will be much more costly for the adversary than for the *VortexNode*, and such a node may decide to temporarily no longer accept new eID requests without affecting already existing eIDs.

Each eID has a lifetime, a maximum number of messages to be processed, and a maximum number of bytes to be sent assigned to it. The lifetime of an eID is typically days and maybe up to a low number of months. Lifetimes may not be extended and are defined by the sender of the request. A node may accept or decline the request if the lifetime of the request or the state of the node does not meet its expectation. The puzzle sent in return may be a fixed value or related to the nodes’ current state and load.

This system guarantees that a sender must invest considerable work (in terms of CPU time required) prior to using resources of a *VortexNode*. A *VortexNode* may raise the complexity of its puzzles when having a high load. This allows for a single user to still obtain an eID while increasing costs for an attacker considerably. Even if someone floods a node with new eIDs, already created eIDs are not affected as their workspace has already been allocated.

The workspace itself contains chunks of the messages (payload blocks) mapped to IDs and operations. The operations transform one or more source IDs onto one or more target IDs. Any of these payload blocks may be assigned to a subsequent message as payload block by a routing block. An operation or a payload block share the lifetime of the respective message header. If operations overlap in output blocks the newest operation (arrived latest) wins. Arriving *VortexMessages* map their payloads onto IDs of the respective workspace of the eID. to allow such mapping the first IDs are special IDs either mapping to the ID 0 (message for local delivery) or IDs 1-127 (always reflect the current message [ingoing or outgoing]).

This concept has certain downsides related to the expiry of eIDs. We will address them in section19.3.1 and section 19.3.3.

15.2.3 Protocol Layers

As already introduced in 14.3, the protocol is built on multiple software layers. On the logic side, the protocol is split into two parts:

1. Transport Layer
   Standard Internet infrastructures provide this layer. The primary goal is to hide or blend our protocol into regular traffic within that layer. Typical examples for such layers are SMTP or XMPP servers.

2. Blending and subsequent layers (the Vortex infrastructure)
   Any user of the Internet may provide these layers. Since these layers may be only Vortex routing nodes or valid endpoints, the nodes may or may not be publicly known. In a first implementation, we build this system as a standard Java application. The primary goal is to compile it to native code afterward and run it on an SoC like infrastructure such as a RaspberryPi or port it to an android device.

   We may further split the Vortex infrastructure layers into

   (a) dBlending layer
   This layer receives messages from the Vortex system and creates transport layer conformant messages and vice-versa. In an ideal case, the messages generated by this layer are indistinguishable from any regular message traffic of the transport layer, and the embedded message is only detectable by the receiving node.

   (b) Routing layer
   The routing layer disassembles and reassembles messages.

   (c) Accounting layer
   The accounting layer has three jobs. First, he has to authorize the message processing after the decryption of the header block by the blending layer. Secondly, he handles all header request blocks and the reply blocks. And third, it keeps track of the accounting regarding the sent messages. Its main purpose is to protect the system from misuse or flooding.

In total, we have four layers. The bottom-most layer consists of unmodified standard infrastructure for transport within the Internet, and the three layers on top build a single *VortexNode*.

15.2.4 Transport Layer

The transport layer is a standard protocol within the Internet. It is neither a *MessageVortex*-specific infrastructure, nor has it been modified for the purpose. Instead, it serves the purpose as a store and forward medium. This medium solves two major problems. First, no NAT traversal technology such as "TCP hairpins"

or "hole punching" is required. And secondly, it compensates for short outages due to regional routing problems to the end-user.

A transport layer should have some generic properties:

- Widely adopted
- Reliable
- Symmetrical built

For a more detailed description of the criteria, see section 16.5.1.

For our first tests, we used a custom transport layer, allowing us to monitor all traffic quickly, and build structures in a very flexible way. This transport layer works locally or in a broadcast-based network with a minimum amount of work for setup and deployment. The API we used may, however, be used to support almost any kind of transport protocol.

In section 16.5, we make a short analysis going through some common protocols outlining the strength and weaknesses of common transport protocols within the Internet.

After that, we focussed on the protocols identified in the previous sections for transport:

- SMTP
- XMPP

For the prototype, we have implemented an SMTP transport agent and the respective blending layer.

15.2.4.1 Blending Layer

The blending layer is taking care of multiple problems:

- It is translating the message block into a suitable format for transport
  This translation includes jobs such as embedding a block as encoded text, as a binary attachment or hide it within a message using steganography. Another demanding task in this context is to create credible content for the transport message itself.
- Extract incoming blocks
  Identify incoming messages containing a possible block and extract it from the message.
- Do housekeeping on the storage layer of the transport protocol
  Access protocols such as POP and IMAP require that messages are deleted from time to time to stay below the sizing quotas of an account. To manage this transport layer account is the job of the blending layer.

There is no specification on the housekeeping part of the blending layer, as this part is specific to the requirements of the account owner. We do, however, recommend to handle messages precisely as if the messages would be handled on an account handled by a human. This means that some messages.

The blending is currently done by merging the *VortexMessage*using either F5 as described in [160] or by doing plain blending, which is a binary embedding. This means that we require jpeg images included in the SMTP message.

Processing of a Message received from the Transport Layer
We define the blending layer to work as follows when receiving messages:

1. Log arrival time on the transport layer.
2. Extract possible *VortexMessage*.
3. Apply decryption on a suspected header block of *VortexMessage*.
4. Identify the header block as valid by querying the accounting layer.
5. Extract and decrypt subsequent blocks.
6. Pass extracted blocks and information to the routing layer.

Processing of a Message received from the Routing Layer
We define the blending layer to work as follows for sending messages:

1. Assemble message as passed on by the routing layer.
2. Using the blending method specified in the routing block, build an empty message.
3. Create a message decoy content.
4. Send the message to the appropriate recipient using the transport layer protocol.

Credible Content Creation for the Transport Layer
One of the most demanding tasks of the blending layer is to create transport protocol messages. In [68], Houmansadr, Brubaker, and Shmatikov expresses that it is easy for a human to determine decoy traffic as the content is easily identifiable as generated content. While this is up to all that we know true, there is a possibility here to generate "human-like" data traffic to a certain extent. For the blending layer, it is not necessarily required to mimic human messages. Instead, the blending layer may generate messages such as password recovery messages, monitoring messages, and even UBM-like content. All these messages have

required properties in common. First, all of them are machine-generated messages which are modified quite often. All of these messages are known to be sent and possibly adapted individually.

For the blending itself, we required a steganographic algorithm. After reviewing the options, we decided to go for F5[160] as a steganographic algorithm. It is a reasonably well-researched algorithm, which attracted many researchers. The original F5 implementation had a detectable issue with artifacts[20] caused by the recompression of the image. This issue was caused only due to a problem in the reference implementation, and the researchers have provided a corrected reference implementation without the weakness.

We looked for other steganographic algorithms, but were unable to find any other suitable algorithm apart from F5, which fulfilled the following set of criteria:

- Unbroken.
- Researched.
- Suitable for embedding in lossy compressed, common image formats (e.g., jpeg).
- An implementation or a well-specified algorithm exists.

We decided to keep our plain embedding algorithm in the implementation. It already requires an in-depth analysis or a human to detect embedding, and the message itself is, even if detected, well-protected. Its biggest most strength is, however, its efficiency. This algorithm is, however, only suitable for public nodes matching up to an observing adversary (as defined in section 13.1). It must not be used in environments where a censoring adversary is suspected.

When using F5, jpeg images are required. Imagery requires to be at least eight times the size of the message embedded. Unlike other approaches harvesting random pics or obtaining them from a local repository, we recommend using machine-generated images such as rendered content. We recognize that custom Gravatars, router and usage graphs of services, or render services are suitable imagery material for our purpose. The message content would obviously be machine-generated content and not being suspect. This would effectively render the dead parrot problem as described in [68] ineffective.

15.2.4.2 Routing Layer

A routing layer needs to receive all payload and routing blocks, and process them (For an exact outline of the routing block, see section 15.2.5). These blocks are stored in a suitable list within the workspace of the eID identified by the header block.

$$\mathbf{ROUTING_o} = \langle [\mathbf{ROUTINGCOMBO}]*, \mathbf{replyBlock}, mapping* \rangle \quad (15.1)$$

$$\mathbf{ROUTINGCOMBO} = \langle processIntervall, K_{peerN+1}, recipient, \mathbf{nextMP}, \mathbf{nextHP}, \mathbf{nextHEADER}, \mathbf{nextROUTING}, \mathbf{assemblyInstructions} \rangle \quad (15.2)$$

$$\mathbf{PL} = \langle \text{payload octets} \rangle* \quad (15.3)$$

$$\mathbf{nextMP} = E^{K^1_{host_{o+1}}} \left(K_{peer_{o+1}} \right) \quad (15.4)$$

$$\mathbf{nextHP} = E^{K^1_{host_{o+1}}} \left(K_{sender_{o+1}} \right) \quad (15.5)$$

$$\mathbf{nextHEADER} = E^{K_{sender_o}} \left(\mathbf{HEADER_{o+1}} \right) \quad (15.6)$$

$$\mathbf{nextROUTING} = E^{K_{sender_o}} \left(\mathbf{ROUTING_{o+1}} \right) \quad (15.7)$$

$$\mathbf{operations} = \langle \text{list of operations} \rangle \quad (15.8)$$

$$\mathbf{assembyInstructions} = \langle blendingInformation, nextHop, \langle \text{mapping operation+} \rangle \rangle \quad (15.9)$$

Figure 15.1: Simplified representation of a routing block

Within the routing block, we find a set of instructions, next *VortexNodes*information, and the encrypted routing blocks for the messages to be assembled. A simplified representation of a routing block is shown in figure 15.1.

The routing of a message is simple. A workspace of an eID contains routing blocks and payload blocks. A routing block has an active time window defined in the header block. Anytime during that time window, a routing layer processes the routing instructions contained in the assembly operations of the routing block. If successful, the message will be sent using the specified blending layer and target address.

The routing layer stores the main information assigned to the operation of routing messages. The following data has to be kept for routing within the eIDs workspace:

- **Build**$[]\langle expiry, buildOperation \rangle$
  The array **Build**[] is a list of building instructions for a message. The server may decide at any time to reject a too big list or long-living message. Thus, he may control the size of this list as well. However, controlling the size of this list will most likely result in the non-delivery of a message.

  The $buildOperation$ is extracted by enumerating $operation*$ while $expiry$ is the upper bound of the $processIntervall$.

- **Payload**$[]\langle expiry, payload, id \rangle$
  The array $Payload[]$ reflects a list of all currently active payloads. Servers may decide to store derivatives of payloads. However, as derived payloads inherit their expiry from the generating operation, such behavior may be safely omitted and operations executed if their result is required.

- **Route**[]$\langle processIntervall,\ blendingInformation,\ nextHop,$ **nextMP**, **nextHP**, **nextHeader**, **nextRouting**, $K_{peer_{o+1}}$, **assemblyInstructions**$\rangle$
  The list of routing information triggers processing. At a randomly chosen time defined in the $processIntervall$, a message is composed. The message is assembled by doing $\langle$**nextMP**, $E_{K_{peer_{o+1}}}\langle$**nextHP**, **nextHEADER**, **nextROUTING**, $payload*\rangle\rangle$. The payloads are created with the help of the arrays $build$[] and $payload$[], and as soon as the message is authorized by accounting and passed to the blending layer, the entry in this list is discarded.

15.2.4.3 Accounting Layer

The accounting layer keeps tracks of all information required assigned to ephemeral identities (eID). It is queried by the blending Layer and routing Layer for authorization of the operations. The accounting layer manages the following tuples of information:

- **eID**[]$\langle expiry, pubKey, mesgsLeft, bytesLeft\rangle$
  The **eID** tuple is the longest living tuple. It reflects an ephemeral identity and exists as long as the current identity is valid. All other tuples are short living lists. As the server decides if he accepts new identities or not, the size of this data is controllable.

- **Puzz**[]$\langle expiry, request, puzzle\rangle$
  The array **Puzz**[] is a list of not yet solved puzzles of this eID. Every puzzle has a relatively short lifespan (typically below 1d). A routing node controls the size of this list by only accepting requests to a certain extent. Typically this list should not surpass two entries as we should have either a maximum of two quota requests or one identity creation request open.

- **Replay**[]$\langle expiry, serial, numberOfRemainingUsages\rangle$
  The array **Replay**[] is a list of serials. List entries are created upon their first usage and remain active until the block is expired.

15.2.5 VortexMessages

A *VortexMessage* is built by combining multiple loosely interconnected blocks. We first name the blocks and their function, and then we explain the inner working of the blocks and do some reasoning why the block has been built as it is.

Figure 15.2 shows an outline of the block structure of a message destined to $host_o$. For a mathematical representation, see figure 15.3.

The first block is the message prefix block $\mathbf{MPREFIX_o}$, which has been encrypted with the public key of the receiving node $K^1_{host_o}$. This block contains the key for decrypting the whole rest of the message. Each PREFIX block contains a symmet-

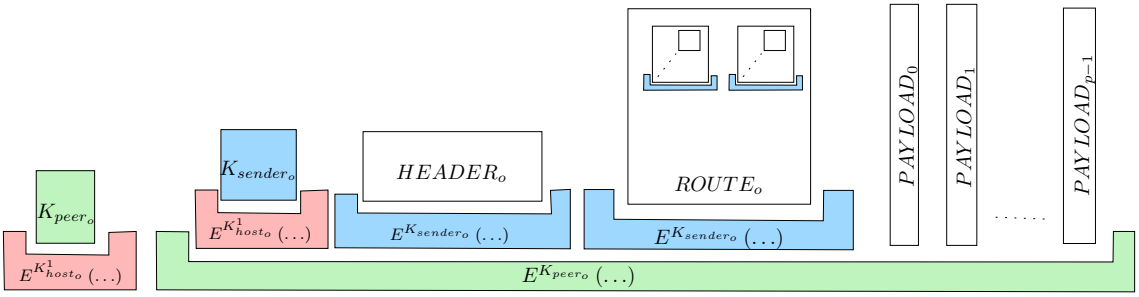


Figure 15.2: Simplified message outline visually and in math

rical ey and the specification on how to encrypt or decrypt with it (mode, padding, IV, and other possibly required parameters) in ASN.1 encoding.

Immediately following the message prefix block, we have the inner message block. This message blocks contain three more blocks and a variable number of payload blocks. The inner message is encrypted with the symmetrical peer key K_{peer_o}. This peer key is specific to this message and nowhere reused. It is only known by the two peer hosts $host_o$ and $host_{o-1}$, and the routing block builder (RBB). More importantly, $host_{o-1}$ does not need to know the host key of $host_o$. Therefore, relaying a message to $host_o$ does not enable $host_{o-1}$ to communicate with $host_o$.

The blocks $HEADER_o$ and $ROUTING_o$ are protected with an additional key K_{sender_o}. The decryption key is obtained by $host_o$ from the header prefix block **HPREFIX**. After only decrypting the header block **HEADER** and verifying its signature, the accounting layer may check if further processing is authorized. The splitting of the two keys allows us to...

- ...send a message to $host_o$ without $host_{o-1}$ knowing the host key of $host_o$.
- ...hide the structure of the message itself.
- ...keep the content of $\mathbf{HEADER_o}$, and $\mathbf{ROUTING_o}$ secret from $host_{o-1}$.

After authorization by the accounting layer, the header block is processed as outlined in section 19.2.2. Basically, we just add the routing blocks and payload to the respective workspace and wait for the routing layer to process the information.

Looking at a full *VortexMessage*, we get the protocol outline, as shown in (33.1) on page 182.

The routing log block is an onionized block. It contains at least a $forwardSecret$, which must match up with the header blocks $forwardSecret$. This mechanism is required to guarantee that routing blocks are not exchanged. The $replyBlock$ provides a possibility, if provided, to contact the original sender of the message without knowing him. It is just a routing block with instructions on how to prepare

$$\mathbf{VORTEXMESSAGE} = \langle \mathbf{MP}^{K^{-1}_{host_o}}, INNERMESSAGE \rangle \tag{15.10}$$

$$\mathbf{INNERMESSAGE} = \langle \mathbf{CP}^{K^{-1}_{host_o}}, \mathbf{H}^{K_{sender_o}}, E^{K^{-1}_{sender_o}}(H(\mathbf{HEADER})), \left[\mathbf{R}^{K_{senderN}}\right], [\mathbf{PL}] * \rangle^{K_{peerN}} \tag{15.11}$$

$$\mathbf{MP}^{K^{-1}_{hostN}} = E^{K^{-1}_{hostN}}\left(\mathbf{PREFIX}\langle K_{peerN}\rangle\right) \tag{15.12}$$

$$\mathbf{HP}^{K^{-1}_{hostN}} = E^{K^{-1}_{hostN}}(\mathbf{HPREFIX}\langle K_{senderN}\rangle) \tag{15.13}$$

$$\mathbf{H}^{K_{senderN}} = E^{K_{senderN}}(\mathbf{HEADER}) \tag{15.14}$$

$$\mathbf{HEADER} = \langle K^{1}_{senderN}, serial, maxReplays, validity, [requests, requestRoutingBlock], [puzzleIdentifier, proofOfWork] \rangle \tag{15.15}$$

$$\mathbf{R}^{K_{senderN}} = E^{K_{senderN}}(\mathbf{ROUTING}) \tag{15.16}$$

$$\mathbf{ROUTING} = \langle [\mathbf{ROUTINGCOMBO}] *, forwardSecret, \mathbf{replyBlock}, \mathbf{operations} \rangle \tag{15.17}$$

$$\mathbf{ROUTINGCOMBO} = \langle processIntervall, K_{peerN+1}, recipient, \mathbf{nextMP}, \mathbf{nextHP}, \mathbf{nextHEADER}, \mathbf{nextROUTING}, \mathbf{assemblyInstructions} \rangle \tag{15.18}$$

$$\mathbf{nextMP} = E^{K^{1}_{host_{o+1}}}\left(K_{peer_{o+1}}\right) \tag{15.19}$$

$$\mathbf{nextHP} = E^{K^{1}_{host_{o+1}}}(K_{sender_{o+1}}) \tag{15.20}$$

$$\mathbf{nextHEADER} = E^{K_{sender_o}}(\mathbf{HEADER_{o+1}}) \tag{15.21}$$

$$\mathbf{nextROUTING} = E^{K_{sender_o}}(\mathbf{ROUTING_{o+1}}) \tag{15.22}$$

$$\mathbf{operations} = \langle \text{list of operations} \rangle \tag{15.23}$$

$$\mathbf{assembyInstructions} = \langle blendingInformation, nextHop, \langle \text{list of mapping operations} \rangle \rangle \tag{15.24}$$

$$\mathbf{PL} = \langle \text{payload octets} \rangle * \tag{15.25}$$

(15.26)

Figure 15.3: Detailed representation of a VortexMessage

the message to be sent. The routing combos contain all the necessary information and prebuilt blocks to create the subsequent messages.

At the very end, we got the payload. These blocks are simply added to the eIDs workspace.

The double encryption of the routing and header block, are doubly encrypted. We could argue that the inner message block should not be encrypted with a peer key. This looks like a flaw at first sight but is, in fact, a feature that is very important. Without this key, any independent observer with knowledge about the blending capabilities of a receiving node may...

- Easier to identify the block structure.
  This statement remains regardless of whether ASN.1 or length prefixed struc-

tures are used. If the structure of a *VortexMessage*is easily identified, the messages may be logged or dropped.

- Identify the routing block size.
  The value of this information is only minimal as it only reflects the complexity of the remaining routing information indirectly.
- Identify the number of payload blocks and their respective sizes.
  Sizing information is valuable when following the path of a message.

15.2.5.1 Message Structure Related to Censorship Circumvention

It is important to note that there is no structure dividing the encrypted peer key from the Inner message block. The size of the peer key block is defined by the key and algorithm of the host key.

The whole *VortexMessage*is, looking from outside, a structureless blob with a maximum of entropy caused by the encryption employed.

This is intentional and by design. Plain embedding uses furthermore a method of splitting, which allows a message block to be embedded in chunks in carrier information. By design, neither the message nor their embedding display detectable attributes allowing them to identify the message.

Exactly as with the routing operations, great care has been applied. Any random sequence of bytes may be interpreted as valid chunking. For more exact implementation details on chunking, see section 18.1.2.

15.2.5.2 Message Structure Related to Information Leaking

From the inside, the **INNERMESSAGE** (see 15.11) is built as a structure leaking the absolute minimum of information. A node receiving and decoding the message will learn the following information:

- IP of the sender of the transport layer.
- The address and embedding schemes of all receiving transport layers.
- The size of the payload blocks.
- The size of the subsequent routing blocks.
- The peer key K_{peer_o}.
- The size of the prefix blocks.

It is unable to extract the following information:

- The required keys to communicate with the suspected peer node.
- Any information related to message size, content, or recipient.

15.2.6 Routing Operations

The routing operations build the core as they define the capabilities of the mixing. We decided to introduce three different classes of operations. Wherever we employ crypto, operations, we may choose the operation required for the operation. No choices exist for the core Reed-Solomon-Function, the padding and spitting operation related to it, and the split and merge operations.

15.2.6.1 *addRedundancy* and *removeRedundancy* Operations

I this section we focus on the core operation of our system. The *addRedundancy* and *removeRedundancy* allow growing message sizes in our system without allowing to identify the decoy traffic.

The Lagrange functions have been proposed in [139] and further more generalized in [94] for sharing secrets. The general idea about all proposed schemes is to distribute informations and restrict access to it so that only if a specified number of shares are captured a secret may be rebuilt. Unlike proposed in these papers we do not apply privacy to our protocol by sharing the data among many points. Instead we use Lagrange functions to create decoy traffic. By doing so even a creator of traffic is unable to tell message traffic from decoy traffic apart.

These operations build the core of the routing capabilities of a node. The operation allows an RBB to add to a message redundancy information or to rebuild a block from a chosen set of information.

The operation itself is shown in figure 15.4.

It may be subdivided into the following operations:

- Pad the original message block in such a way, that all resulting blocks are a multiple of the block size $(C_1, \ldots, C_n)$ of the encrypting cipher.
- Apply a Reed Solomon operation in a given GF space with a vanderMonde matrix.
- Encrypt all resulting blocks with unpadded, symmetrical encryption.

The padding applied in the first step is non-standard padding. The reason for this lies in the properties required by the operation. The presence of standard padding may leak, whether the block has been successfully decrypted or not. Therefore, we created a padding with the following properties:

- The padding must not leak whether the rebuild cycle of the operation was successful or not.
- Anyone knowing the routing block content and the transmitted message must be able to predict any treated block, including all padding bytes.

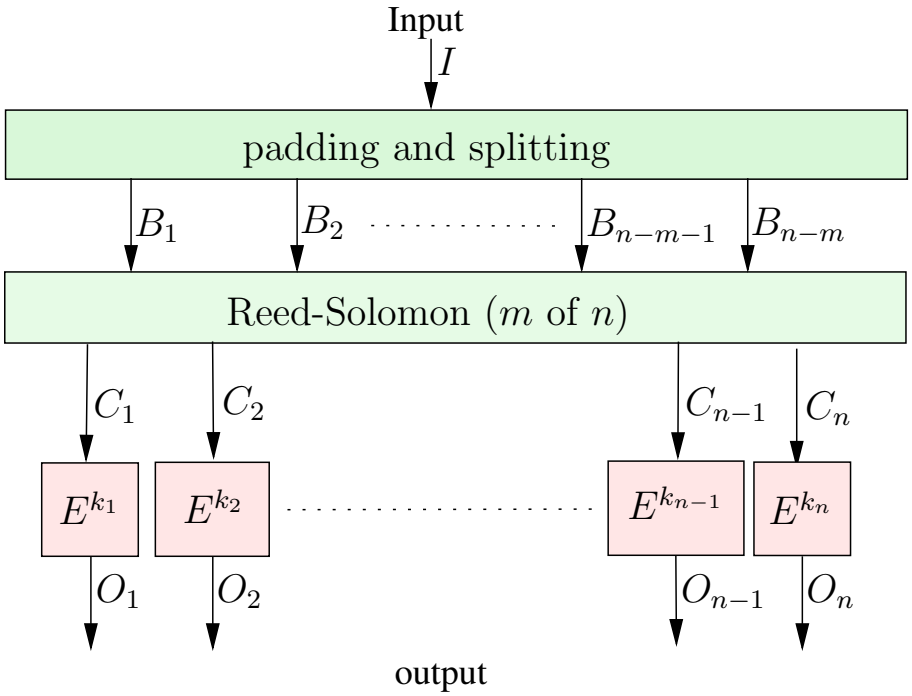


Figure 15.4: Outline of the addRedundancy operation

- The padded content must provide resulting blocks of required size to enable non-padded encryption after the RS operation
- The padding must work with any size of padding space.
- The padded and encrypted block must not leak an estimate of the original content.

The padded block $\mathbf{X}$ is created from a padding value p, the unpadded block $\mathbf{M}$ and a series of padding bytes. We build $\mathbf{X}$ for a function $RS_{\text{m of n}}$ (allows adding m redundancy blocks) and an encryption block $\mathbf{M}$ sized K as follows:

$$
\begin{aligned}
i &= len(\mathbf{M}) && (15.27)\\
e &= lcm\left(blocksize\left(E^K\right), n\right) && (15.28)\\
l &= \left\lceil \frac{i+4+C2}{e} \right\rceil \cdot e && (15.29)\\
p &= i + \left(C1 \cdot l \pmod{\left\lfloor \frac{2^{32}-1-i}{l} \right\rfloor \cdot l}\right) && (15.30)\\
\mathbf{X} &= \langle p, \mathbf{M}, R_t(s, l-i) \rangle && (15.31)\\
&= \langle p, \mathbf{M}, R_t(s, l-(p \pmod{len(\mathbf{X})-4})) \rangle
\end{aligned}
$$

Variable i denotes the length. By calculating e as the least common multiplyer of the encryption block size and the number of output blocks, we determine the block size required for our operation so that no subsequent padding is required.

The remainder of the input block, up to length l, is padded with random data. The random padding data may be specified by RBB though a PRNG spec t and an initial seed value s. The message is padded up to size L. All resulting, encrypted blocks do not require any padding. This because the initial padding guarantees that all resulting blocks are dividable by the block size of the encrypting function. If not provided by an RBB, an additional parameter $C1$ is chosen as random positive integer and $C2 = 0$ by the node executing the operation.

To reverse a successful message recovery information of a padded block $\mathbf{X}$, we calculate the original message size by extracting p and doing $i = len(\mathbf{M}) = p \pmod{len(\mathbf{X}) - len(p)}$.

This padding has many advantages:

1. The padding does not leak if the rebuilding of the original message was successful. Any value in the padding may reflect a valid value.

2. Since we have a value $C2$, the statement that a message size is within $len(\mathbf{X}) >= size > (len(\mathbf{X}) - e)$ is no longer true and any value smaller $len(\mathbf{X}) - e$ may be correct as well.

3. An RBB may predict the exact binary image of the padded message when specifying $C1$, $C2$, and $R_t(s,)$.

4. A node knowing the original parameters $C1$, $C2$, and the initial PRNG seed s can detect successful decryption.

Apart from being non-standard padding, the padding has additional downsides:

- The padding is inefficient compared to simple paddings such as PKCS#7

- The padding requires an initialized PRNG to generate the padding.

- Depending on the chosen parameters, the padding overhead may become significant.

After the padding, the date is ready for the Reed-Solomon part of the operation. We first group the data vector into a matrix $\mathbf{A}$ with m columns to do the operations efficiently. The previous padding guarantees that all columns have a length, which is dividable by the block size of the encryption step applied later.

$$t = n - 1 \tag{15.32}$$

$$inbytes\mathbf{A} = vec2mat\left(\mathbf{X}, \frac{len\left(\mathbf{X}\right)}{m}\right) \tag{15.33}$$

$$\mathbf{V} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & \cdots & 0^{(m-1)} \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{(m-1)} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t^0 & t^1 & t^2 & \cdots & t^{(m-1)} \end{pmatrix} \tag{15.34}$$

$$\mathbf{P} = \mathbf{VA}\left(GF\left(2^{\omega}\right)\right) \tag{15.35}$$

$$\langle \mathbf{Q_1}, \ldots, \mathbf{Q_n} \rangle = row2vec(P) \tag{15.36}$$

$$R_i = E^{K_i}\left(Q_i\right) \tag{15.37}$$

We apply the Reed-Solomon function by employing a Vandermonde matrix ($\mathbf{V}$). We build the data matrix ($\mathbf{A}$) by distributing the data into $\frac{len(\mathbf{X})}{m}$ columns. This results in a matrix with m rows. Unlike in error-correcting systems, we do not normalize the matrix so that the result of the first blocks is equivalent to the original message. Instead, the error-correcting information is distributed over all resulting blocks ($\mathbf{Q_i}$). Since the entropy of the resulting blocks is lowered as shown in figure 28.3 and may thus leak an estimate of how a resulting block may have been treated, we added the encryption step to equalize entropy again. The previously introduced padding guarantees that there is no further padding on block-level required. The key used to encrypt the single blocks must not be equivalent. Equivalent keys have the side effect encrypting equal blocks into the same cyphertext. We observed faint but statistically relevant reminders of the unencrypted graphs when treating the same block with the same key and different redundancy parameters.

15.2.6.2 *encrypt* and *decrypt* Operations

The encrypt and decrypt operations are essential for the requirement that tagging should not be possible. Unlike the *addRedundancy* and *removeRedundancy*, the splitting operations do not feature any encryption step after splitting or merging. Reusing a payload block that has only been split or merged would repeat the payload pattern on multiple nodes during transfer. That is why we require to have encryption.

The reason for not building this step into the split and merge function was simple. We needed a separate encryption step to be able to work as an onionizing system, and there were use cases where integrated encryption did not make sense. For further details on this topic, see section 24.3.

15.2.6.3 *mergePayload* and *splitPayload* operation

The *splitPayload* operation splits a payload block into two chunks of different or equal sizes. The parameters for this operation are:

- source payload block pb_1
- fraction f
  A floating-point number which is describing the size of the first chunk. If the fraction is "1.0", then the whole payload is transferred to the second target chunk

If $len(pb_1)$ expresses the size of a payloadblock called pb_1 in bytes, then the two resulting blocks of the SpitPayload Operation pb_2 and pb_3 have to follow the following rules:

$$split(f, pb_1) = \langle pb_1, pb_2 \rangle \tag{15.38}$$
$$pb_1.startsWith(pb_2) \tag{15.39}$$
$$pb_1.endsWith(pb_3) \tag{15.40}$$
$$len(pb_2) = floor(len(pb_1) \cdot f) \tag{15.41}$$
$$len(pb_1) = len(pb_2) + len(pb_3) \tag{15.42}$$

The *mergePayload* operation combines two payload blocks into one. The parameters for this operation are:

- first source payload block pb_1
- second source payload block pb_2

If $len(pb)$ expresses the size of a payloadblock called pb in bytes then resulting block of the MergePayload Operation pb_3 have to follow the following rules:

$$merge(pb_1, pb_2) = pb_3 \tag{15.43}$$
$$pb_3.startsWith(pb_1) \tag{15.44}$$
$$pb_3.endsWith(pb_2) \tag{15.45}$$
$$len(pb_3) = len(pb_1) + len(pb_2) \tag{15.46}$$

Unlike other operations, this operation has no encryption step attached to it. We usually attached an encryption step to remove repeating patterns from the *VortexMessage*stream.

It has to be mentioned that this operation tuple has some issues when it comes to floating-point implementations. They are solvable but had to be specified unexpectedly precisely in order to enable a true cross-platform implementation. For more information regarding the issue and exact implementation, see section 19.2.4.

15.3 Summary

The *MessageVortex*-Protocol is split into the four layers “Transport” (a common internet standard protocol), “blending” (extracting and embedding *VortexMessages*), “Routing” (A layer reassembling messages according to received instruction), and “Accounting” (Keeps track of all stored data and discards expired information).

All nodes are realized in decentral devices such as computers or mobile phones. Messages are hidden with either plain embedding or (referred) F5 in the transport layer message. The routing layer processes messages by applying operations to messages. Valid operations are encrypt or decrypt a message chunk, split a message chunk into two parts, merge two parts into one, or add or remove redundancy information. The last operation is the most valuable. This operation allows by employing an extended Reed-Solomon-Operation to add decoy traffic to the message flow without enabling a node to identify such traffic. Furthermore, it allows a sender to send parts of a message through multiple chains of routing nodes to a recipient. Each message itself does not leak the message content since depending on the completing block, any message with appropriate length may be valid.

The routing itself is done in a temporarily allocated storage called “workspace”, which is tied to an ephemeral identity (eID) represented by an asymmetric key pair. To get an eID, a sender typically solves a crypto puzzle.

Payloads of *VortexMessages* are mapped into the workspace and are assigned a unique ID within that workspace. The subsequent routing blocks and their operations are added as well and processed in a time interval defined by the RBB.

Part V

Implementation

***

No matter how hard you work,
someone else is working
harder.
Elon Musk, entrepreneur

The implementation differs from the academic model in some details. It is formeost more precise than the academic model. Furthermore, it requires a strict definition of the implementation in order to guarantee the interoperability between different implementations.

This section focuses therefore on the details of the reference implementation.

16 Selection of Algorithms, Encodings, and Protocols

In this chapter, we choose the following mandatory supported algorithms:

- Encoding: ASN.1
- Encryption
  - AES128/256
  - Camellia128/256
- Modes
  - ECB
  - GCM
- Paddings
  - PKCS#1
  - PKCS#7
- MACs
  - SHA256/512
  - RIPE-MD256
- PRNG
  - mrg32k3a
  - blumMicali

Where security relevant we always choose two independent algorithms. As our protocol has the means of signaling them, we may support additional algorithms without affecting communication while improving the variety of available algorithms.

In the following sections, we emphasize on the choice and the encoding used on the protocol level.

For all algorithms we apply the following criteria:

- Always focus on common standards
- Focus on interoperability when selecting standards
- Focus on efficiency (wherever possible use simple, parallelizable algorithms)
- When sensible and possible chose at least two unrelated algorithms (e.g., cryptographic algorithms or MACs)

16.1 Encoding Scheme

As encoding scheme we specified ASN.1[36]. It is more compact than the originally selected XML-Standard. It is very common in the fields of telecommunication and encryption (e.g. X509 certificates are represented in ASN.1). To maintain biggestmost interoperability, we choose DER encoding as it has exactly one possible representation for every value. This is especially important when dong signing or solving puzzles in our case.

On the downside ASN-1 encoding is, unlike XML, not human readable. As we hide the messages anyway, we considered this as minor flaw, as we need to have an extracting program anyway to see the content of a message.

16.2 Cipher Selection

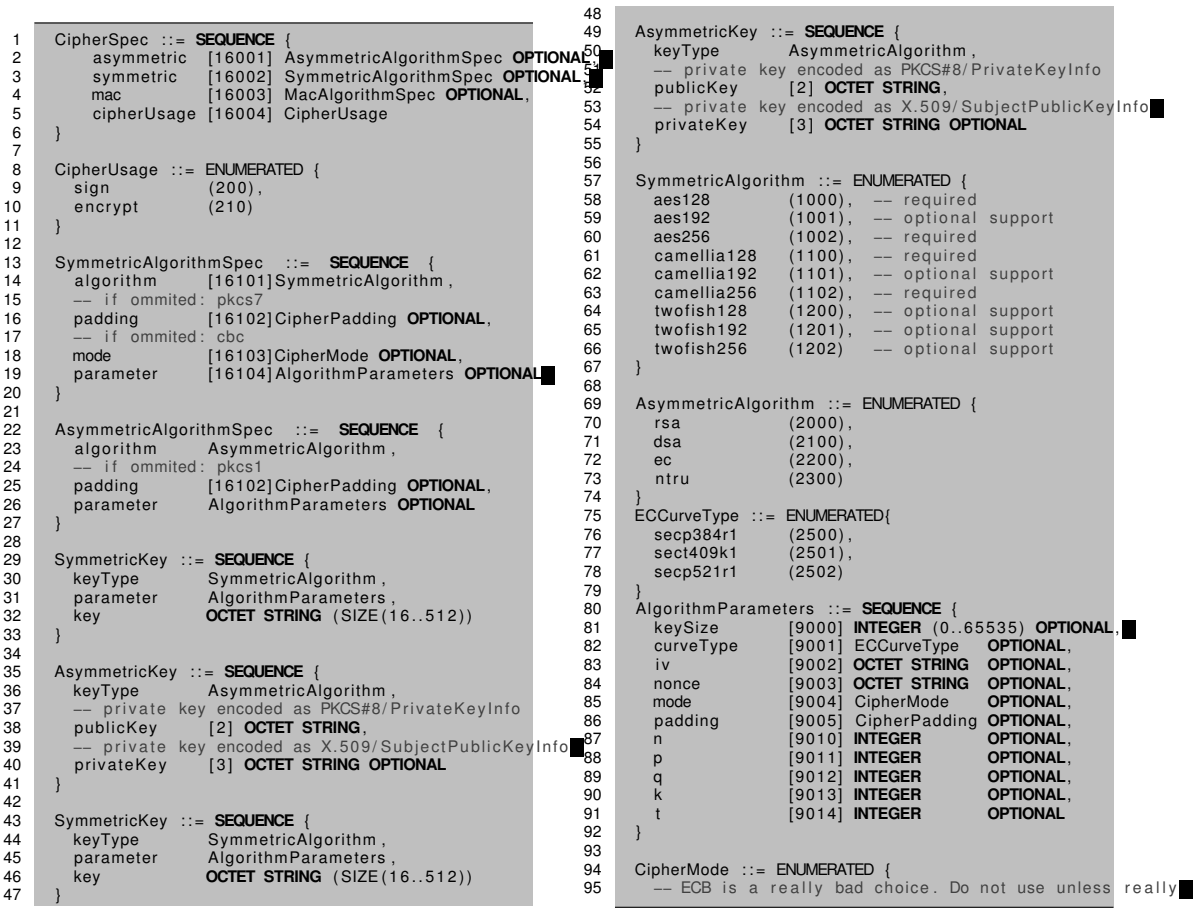
In this protocol, a lot of encryption and hashing algorithms have to be used. This choice of these algorithms should be explained.

We decided to define fixed key sizes for symmetric ciphers as we went with block ciphres. For the asymmetric ciphres we encode the keysize in the parameters as they are due to their differences far more often flexible.

From the requirements side, we have to follow the following principle: First of all, we need a subset of encryption algorithms all implementations may rely on. Defining such a subset guarantees interoperability between all nodes regardless of their origins.

Secondly, we need to have a spectrum of algorithms in such a manner that it may be (a) enlarged if necessary and (b) there is an alternative if an algorithm (or a mathematical problem class) is broken (so that algorithms may be withdrawn if required without affecting the function in general).

And third, due to the onion-like design described in this document, asymmetric encryption should be avoided in favor of symmetric encryption to minimize losses



```
CipherSpec ::= SEQUENCE {
    asymmetric  [16001] AsymmetricAlgorithmSpec OPTIONAL,
    symmetric   [16002] SymmetricAlgorithmSpec OPTIONAL,
    mac         [16003] MacAlgorithmSpec OPTIONAL,
    cipherUsage [16004] CipherUsage
}

CipherUsage ::= ENUMERATED {
  sign          (200),
  encrypt       (210)
}

SymmetricAlgorithmSpec  ::=  SEQUENCE  {
  algorithm     [16101]SymmetricAlgorithm,
  -- if ommited: pkcs7
  padding       [16102]CipherPadding OPTIONAL,
  -- if ommited: cbc
  mode          [16103]CipherMode OPTIONAL,
  parameter     [16104]AlgorithmParameters OPTIONAL
}

AsymmetricAlgorithmSpec  ::=  SEQUENCE  {
  algorithm     AsymmetricAlgorithm,
  -- if ommited: pkcs1
  padding       [16102]CipherPadding OPTIONAL,
  parameter     AlgorithmParameters OPTIONAL
}

SymmetricKey ::= SEQUENCE {
  keyType       SymmetricAlgorithm,
  parameter     AlgorithmParameters,
  key           OCTET STRING (SIZE(16..512))
}

AsymmetricKey ::= SEQUENCE {
  keyType       AsymmetricAlgorithm,
  -- private key encoded as PKCS#8/PrivateKeyInfo
  publicKey     [2] OCTET STRING,
  -- private key encoded as X.509/SubjectPublicKeyInfo
  privateKey    [3] OCTET STRING OPTIONAL
}

SymmetricKey ::= SEQUENCE {
  keyType       SymmetricAlgorithm,
  parameter     AlgorithmParameters,
  key           OCTET STRING (SIZE(16..512))
}
```

```
AsymmetricKey ::= SEQUENCE {
  keyType       AsymmetricAlgorithm,
  -- private key encoded as PKCS#8/PrivateKeyInfo
  publicKey     [2] OCTET STRING,
  -- private key encoded as X.509/SubjectPublicKeyInfo
  privateKey    [3] OCTET STRING OPTIONAL
}

SymmetricAlgorithm ::= ENUMERATED {
  aes128        (1000),  -- required
  aes192        (1001),  -- optional support
  aes256        (1002),  -- required
  camellia128   (1100),  -- required
  camellia192   (1101),  -- optional support
  camellia256   (1102),  -- required
  twofish128    (1200),  -- optional support
  twofish192    (1201),  -- optional support
  twofish256    (1202)   -- optional support
}

AsymmetricAlgorithm ::= ENUMERATED {
  rsa           (2000),
  dsa           (2100),
  ec            (2200),
  ntru          (2300)
}
ECCurveType ::= ENUMERATED{
  secp384r1     (2500),
  sect409k1     (2501),
  secp521r1     (2502)
}
AlgorithmParameters ::= SEQUENCE {
  keySize       [9000] INTEGER (0..65535) OPTIONAL,
  curveType     [9001] ECCurveType    OPTIONAL,
  iv            [9002] OCTET STRING   OPTIONAL,
  nonce         [9003] OCTET STRING   OPTIONAL,
  mode          [9004] CipherMode     OPTIONAL,
  padding       [9005] CipherPadding  OPTIONAL,
  n             [9010] INTEGER        OPTIONAL,
  p             [9011] INTEGER        OPTIONAL,
  q             [9012] INTEGER        OPTIONAL,
  k             [9013] INTEGER        OPTIONAL,
  t             [9014] INTEGER        OPTIONAL
}

CipherMode ::= ENUMERATED {
  -- ECB is a really bad choice. Do not use unless really
```

Figure 16.1: Definition of the structures related to ciphers

due to the key length and the generally higher CPU load opposed by asymmetric keys.

If the algorithm is generally bound to specific key sizes (due to S-Boxes or similar constructs), the key size is incorporated into the definition. If not, the key size is handled as a parameter.

The key sizes have been chosen in such a manner that the key types form tuples of approximately equal strength. The support of Camelia192 and Aes192 has been defined as optional. However, as they are wildly common in implementations, they have already been standardized as they build a possibility to step up security in the future.

Having these criteria for choice, we chose to use the following keys and key sizes:

- Symmetric
  - AES (key sizes: 128, 192, 256)
  - Camellia (key sizes: 128, 192, and 256)
- Asymmetric

  - RSA (key size: 2048, 4096, and 8192)
  - Named Elliptic Curves
    * secp384r1
    * sect409k1
    * secp521r1
- Hashing
  - sha3-256
  - sha3-384
  - sha3-512
  - RIPE-MD160
  - RIPE-MD256
  - RIPE-MD320

Within the implementation, we assigned algorithms to a security strength level:

- LOW
  AES128, Camellia128, RSA1024, sha3-256
- MEDIUM
  AES192, Camellia 192, RSA2048, ECC secp384r1, sha3-256
- HIGH
  AES256, Camellia256, RSA4096, ECC sect409k1, sha3-384
- QUANTUM
  AES256, Camellia256, RSA8192, ECC secp521r1, ntru, sha3-512

This allows categorizing the used algorithms to a strength. This list, however, should only serve the purpose of selecting algorithms for people without cryptological know-how.

16.3 Mode Selections

We evaluated the most common cipher modes for suitability. For MessageVortex, we focussed on modes that have the properties parallelizable, random access, and do not do authentication. The main focus, besides the characteristics mentioned before, was on the question of whether there is an open implementation available in java, which is reasonably tested.

Figure 16.2 shows the selected paddings and their requirement level.

Very important was, that we quite often reencrypt already encrypted content. As a result we had not to exclude algorithms such as ECB.

```
CipherMode ::= ENUMERATED {
  -- ECB is a really bad choice. Do not use unless really
  -- necessary and you are sure that the content is already
  -- encrypted
  ecb           (10000), -- optional support
  cbc           (10001), -- required
  eax           (10002), -- optional support
  ctr           (10003), -- required
  ccm           (10004), -- optional support
  gcm           (10005), -- optional support
  ocb           (10006), -- optional support
  ofb           (10007), -- optional support
  none          (10100)  -- required
}
```

Figure 16.2: Enumeration definition of modes in ASN.1 with support requirements.

- ECB (Electronic Code Book)
  ECB is the most basic mode. Each block of the cleartext is encrypted on its own. This results in a big flaw: blocks containing the same data will always transform to the same ciphertext. This property makes it possible to see some structures of the plain text when looking at the ciphertext. This solution allows the parallelization of encryption, decryption, and random access while decrypting. Due to these flaws, we rejected this mode.

- CBC (Cypher Block Chaining)
  CBC extends the encryption by xor'ing an initialization vector into the first block before encrypting. For all subsequent blocks, the ciphertext result of the preceding block is taken as xor input. This solution does not allow parallelization of encryption, but decryption may be paralleled, and random access is possible. As another downside, CBC requires a shared initialization vector. As with most IV bound modes, an IV/key pair should not be used twice, which has implications for our protocol.

- PCBC (Propagation Cypher Block Chaining)
  CBC extends the encryption by xor'ing, not the ciphertext but a xor result of ciphertext and plaintext. This modification denies parallel decryption and random access compared to CBC.

- EAX
  EAX has been broken in 2012[99] and is therefore rejected for our use.

- CFB (Cypher Feedback) CFB is specified in [39] and works precisely as CBC with the difference that the plain text is xor'ed and the initialization vector, or the preceding cipher result is encrypted. CFB does not support parallel encryption as the ciphertext input from the preceding operation is required for an encryption round. CFB does, however, allow parallel decryption and random access.

- OFB
  [39] specifies OFB and works exactly as CFB except for the fact that not the ciphertext result is taken as feedback but the result of the encryption before

xor'ing the plain text. This denies parallel encryption and decryption, as well as random access.

- OCB (Offset Codebook Mode)
  This mode was first proposed in [125] and later specified in [126]. OCB is specifically designed for AES128, AES192, and AES256. It supports authentication tag lengths of 128, 96, or 64 bits for each specified encryption algorithm. OCB hashes the plaintext of a message with a specialized function $H_{OCB}(\mathbf{M})$. OCB is fully parallelizable due to its internal structure. All blocks except the first and the last can be encrypted or decrypted in parallel.

- CTR
  CTR is specified in [87] and is a mixture between OFB and CBC. A nonce concatenated with a counter incrementing on every block is encrypted and then xor'ed with the plain text. This mode allows parallel decryption and encryption, as well as random access. Reusing IV/Key-pairs using CTR is a problem as we might derive the xor'ed product of two messages. This problem only applies where messages are not uniformly random such as in an already encrypted block.

- CCM
  Counter with CBC-MAC (CCM) is specified in [161]. It allows to pad and authenticate encrypted and unencrypted data. It furthermore requires a nonce for its operation. The size of the nonce is dependent on the number of octets in the length field. In the first 16 bytes of the message, the nonce and the message size is stored. For the encryption itself, CTR is used. It shares the same properties as CTR.

  It allows parallel decryption and encryption as well as random access.

- GCM (Galois Counter Mode)
  GCM has been defined in [95], and is related to CTR but has some major differences. The nonce is not used (just the counter starting with value 1). To authenticate the encryption, an authentication token $auth$ is hashed with H_{GFmult} and then xor'ed with the first cipher block. All subsequent cipher blocks are xor'ed with the previous result and then hashed again with H_{GFmult}. After the last block the output o is processed as follows: $H_{GFmult}(o \bigoplus (len(A)||len(B))) \bigoplus E^{K^0}(counter_0)$. As a result, GCM is not parallelizable and does not support random access.

  The mode has been analyzed security-wise in 2004 and showed no weaknesses in the analyzed fields [96].

  GCM supports parallel Encryption and decryption. Random access is possible. However, authentication of encryption is not parallelizable. The authentication makes it unsuitable for our purposes. Alternatively, we could use a fixed authentication string.

- XTS (XEX-based tweaked-codebook mode with ciphertext stealing)
  This mode is standardized in IEEE 1619-2007 (soon to be superseded). A

rough overview of XTS may be found at [91]. It was developed initially for Disks offering random access and authentication at the same time.

- CMC (CBC-mask-CBC) and EME (ECB-mask-ECB)
  In [61] Halevi and Rogaway introduces a cipher mode which is extremely costly as it requires two encryptions. CMC is not parallelizable due to the underlying CBC mode, but EME is.

- LRW
  LRW is a tweakable narrow-block cipher mode described in [154]. This mode shares the same properties as EBC but without the weakness of the same clear text block resulting in the same ciphertext. Similarly to XEX, it requires a tweak instead of an IV.

We decided to go with mainly cbc. However, most of the implementations are available and lightweight, we therefore were not as restrictive as ususal when defining a minimal set

16.4 Padding selection

A plain text stream may have any length. Since we always encrypt in blocks of a fixed size, we need a mechanism to indicate how many bytes of the last encrypted block may be safely discarded.

We have chosen for the paddings outlined in figure 16.3 to be supported.

```
1  CipherPadding ::= ENUMERATED {
2    none            (10200), -- required
3    pkcs1           (10201), -- required
4    rsaesOaep       (10202), -- optional support
5    oaepSha256Mgf1  (10202), -- optional support
6    pkcs7           (10301), -- required
7    ap              (10221)  -- required
8  }
```

Figure 16.3: Enumeration definition of paddings in ASN.1 with support requirements.

16.4.1 RSAES-PKCS1-v1_5 and RSAES-OAEP

This padding is the older of the paddings standardized for PKCS1. It is basically a prefix of two bytes followed by a padding set of non zero bytes and then terminated by a zero byte and then followed by the message. This padding may give a clue if decryption was successful or not. RSAES-OAEP ist the newer of the two padding standards

16.4.2 PKCS7

This padding is the standard used in many places when applying symmetric encryption up to 256 bits key length. The free bytes in the last cipher block indicate the number of bytes being used. This makes this padding very compact. It requires only 1 Byte of functional data at the end of the block. All other bytes are defined but not needed.

16.4.3 OAEP with SHA and MGF1 padding

This padding is closely related to RSAES-OAEP padding. The hash size is, however, bigger, and thus, the required space for padding is much higher. OAEP with SHA and MGF1 Padding is used in asymmetric encryption only. Due to its size, it is important to note that the payload in the last block shrinks to $keySizeInBits/8 - 2 - MacSize/4$.

In our approach, we have chosen to allow these four paddings. The allowed sha sizes match the allowed mac sizes chosen above. It is important to note that padding costs space at the end of a stream. Since we are always using one block for signing, we have to take care that the chosen signing mac plus the bytes required for padding do not exceed the key size of the asymmetric encryption. While this usually is not a problem for RSA as there are keys 1024+ Bits required, it is an essential problem for ECC algorithms as there are much shorter keys needed to achieve an equivalent strength compared to RSA.

16.4.4 Honorable Mention: A Padding for *redundancy* Operations

We have introduced an additional type of padding not related to these paddings. We required for the *addRedundancy* the following unique properties. Unfortunately, we were unable to find any padding which matched the following properties simultaneously:

- Padding must not leak successful decryption
  For our *addRedundancy* operation, we required padding that had no detectable structure as a node should not be able to tell whether a *removeRedundancy* operation did generate content or decoy.
- Padding of more than one block
  Due to the nature of the operation, it is required to be able to pad more than just one block.

This padding is the only padding for the *addRedundancy* and *removeRedundancy* operations.

16.4.5 Pseudo Random Number Generator Selection

For our *addRedundancy* and *removeRedundancy* operations we needed a pseudo random number generator (PRNG). For our implementation we did not research deeply this part as it seemed irrelevant. The only criteriion was, that it had to create content indistinguishable from an encrypted message. This criterion arose as we use it for padding invisibly an already encrypted message.

The PRNG used for our implementation is an xorshift+ generator. It is based on the XSadd PRNG[90] and passes the bigcrush PRNG test suite. It is a fast, xor based PRNG which has two internal 64 bit seed states s_0 respectively s_1 and is defined as follows:

$$\begin{aligned} x &= s_0 && (16.1)\\ s_0 &= s_1 && (16.2)\\ x &= x \oplus (x \ll 23) && (16.3)\\ s_1 &= x \oplus s_1 \oplus (x \gg 17) \oplus (s_1 \gg 26) && (16.4)\\ nextNumber &= s_1 + s_0 && (16.5) \end{aligned}$$

We have chosen this comparably weak PRNG for practical reasons. It is fast, simple, and is based on operations easy to implement on hardware. As we do not need a cryptographically strong PRNG, it is our primary choice so far.

As the protocol is heavily dependent on security we have introduced everywhere at least one alternate algorithm which may be used if one of the choices may become a problem. In order to have a second choice for the PRNG we define the Blum-Micali PRNG as described in [16]. This PRNG is a cryptographically secure PRNG and is defined as follows:

p is prime and g is a primitive root modulo p. x_0 reflects the seed state.

$$x_{i+1} = g^{x_i} \mod p \qquad (16.6)$$

16.5 Transport Layer Protocol Selection

The following sections list common Internet protocols. We analyze those protocols for the fitness as transport layer of message vortex.

We will identify SMTP and XMPP as good transport layer protocols for the *MessageVortex*approach, as they have all required properties.

All sections are structured the same way. We first refer to the protocol or standard and describe it in the simplest possible form. We refer to subsequent standards if required to consider extensions where sensible. We then apply the previously

referenced criteria and make a concise summary of the suiting of the protocol as a transport layer. The findings of this section is listed in table 16.1. The list here does not reflect the quality or maturity of the protocols. It is a simple analysis of suiting as a transport layer.

All sections are structured the same way.

- Description
  We first refer to the protocol or standard and describe it in the simplest possible form. We refer to subsequent standards if required to consider extensions where sensible.
- Apply criteria

  We then apply the previously referenced criteria and make a concise summary of the suiting of the protocol as a transport layer. The findings of this section is listed in table 16.1. The list here does not reflect the quality or maturity of the protocols. It is a simple analysis of suiting as a transport layer.

16.5.1 Applied Criteria

- Widely adopted (Ct1)
  The more widely adopted and used a protocol is, the harder it is due to the sheer mass for an adversary to monitor , filter, or block the protocol. This is important for censorship resistance of the protocol.
- Reliable (Ct2)
  Message transport between peers should be reliable. As messages may arrive anytime from everywhere, we do not have means to synchronize the peer partners on a higher level without investing a considerable effort. Furthermore, the availability of information when what type of information should be available at a specific point in the system would drastically simplify the identification of peers. To avoid synchronization, we do look for inherently reliable protocols.
- Symmetrical built (Ct3)
  The transport layer should rely on a peer to peer base. All servers implement a generic routing that requires no prior knowledge of all possible targets. This criterion neglects centralized infrastructures. This criterion may be dropped, assuming that the blending layer or a specialized transport overlay is responsible for routing.

16.5.2 Analyzed Protocols

We were unable to find a comprehensive list of protocols being used within the Internet and their bandwidth consumption. A weak reference is [163]. This

wekaness is founded in the fact that traffic in this report is callified among two critera: Know server or known port. As streaming services consume a considerable part of the Internet bandwidth (acording to the report more than 60% download). The focus on the report lies on the bandwidth intense figures. However, leavin aside all sources which are strictly one way or dominated by a small number of companies worldwide, the “top 10” list of the report shrinks to the two categories “File sharing” (Rank 5; 4.2% download and 30.2% upload) and “Messaging” (Rank 8; 1.6% download and 8.3% upload bandwidth).

In lack of such material we first collected a list of all common Internet messaging protocols (synchronous and asynchronous). We added furthermore some of the most common transfer protocols such as HTTP and FTP and analyzed this list.

- Messaging Protocols
  - SMTP
  - CoAP
  - MQTT
  - AMQP
  - XMPP
  - WAMP
  - SMS
  - MMS
- Other Protocols
  - FTP, SFTP, and FTPS
  - TFTP
  - HTTP

The following protocols have been discarded as we have considered them as outdated:

- MTP[143] (obsoleted by SMTP)
- NNTP[45] (outdated and has only a small usage acording to [72])

We furthermore discarded all RPC-related protocols as they would by definition violate Ct3.

16.5.3 Analysis

16.5.3.1 HTTP

The HTTP protocol allows message transfer from and to a server and is specified in RFC2616 [102]. It is not suitable as a communication protocol for messages due to the lack of notifications. There are some extensions that would allow such communications (such as WebDAV). Still, in general, even those are not suitable as they require a continuous connection to the server to get notifications. Having a "rollup" of notifications when connecting is not there by default but could be implemented on top of it. HTTP servers listen on standard ports 80 or 443 for incoming connects. The port 443 is equivalent to the port 80 except for the fact that it has a wrapping encryption layer (usually TLS). The incoming connects (requests) must offer a header part and may contain a body part that would be suitable for transferring messages to the server. The reply to this request is transferred over the same TCP connection containing the same two sections.

HTTP0.9-HTTP/1.1 are clear text protocols which are human-readable (except for the data part which might contain binary data). The HTTP/2[13] protocol is using the same ports and default behavior. Unlike HTTP/0.9-HTTP/1.1, it is not a clear text but encodes headers and bodies in binary form.

To be a valid candidate as storage, unauthenticated WebDAV support, as specified in [38], must be assumed.

The protocol does satisfy the first two main criteria (Ct1: Widely Adopted and Ct2: Reliable). The main disadvantage in terms of a message transport protocol is that this protocol is not symmetrically. A server is always just "serving requests" and not sending information actively to peers. This Request-Reply violates criteria (Ct3: Symmetrically built) and makes the protocol not a primary choice for message transport.

It is possible to add such behavior to the blending layer using HTTP servers as pure storage. Such a behavior would however be most likely detectable and thus no longer be censorship resistant.

16.5.3.2 FTP

FTP is defined in RFC959[116]. This Protocol is intended for authenticated file transfer only. There is an account available for general access ("anonymous"). This account does normally not offer upload rights for security reasons. It is possible to use FTP as a message transfer endpoint. The configuration would work as follows: the user "anonymous" has upload rights only. It is unable to download or list a directory. A node may upload a message with a random name. In case a collision arises, the node retries with another random name. The blending layer picks messages up using an authenticated user. This workaround has multiple downsides. At first, handling FTP that way is very uncommon and usually requires an own dedicated infrastructure. Such beavior would make the proto-

col again possibly detectable. Secondly, passwords are always sent in the clear within FTP. Encryption as a wrapping layer (FTPS) is not common, and SFTP (actually a subsystem of SSH) has nothing in common with FTP except for the fact that it may transfer files as well.

Furthermore, FTP may be problematic when used in active mode for firewalls. All these problems make FTP not very suitable as a transport layer protocol. FTPS and SFTP feature similar weaknesses as the FTP version in terms of detectability of non-standard behavior.

Like in HTTP, a disadvantage of FTP in terms of a message transport protocol is that this protocol is not symmetrically. A server is always just “serving requests” and not sending information actively to peers. This Request-Reply violates criteria (Ct3: Symmetrically built) and makes the protocol not a primary choice for message transport. The Protocol, however, satisfies the first two criteria (Ct1: Widely Adopted and Ct2: Reliable).

16.5.3.3 TFTP

TFTP has, despite its naming similarities to FTP, very little in common with it. TFTP is a UDP based file transfer protocol without any authentication scheme. The possibility of unauthenticated message access makes it not suitable as a transport layer. The protocol is due to the use of UDP in a meshed network with redundant routes. Since the Internet has a lot of these redundant routes, this neglects the use of this protocol.

TFTP is rarely ever used on the Internet, as its UDP based nature is not suitable for a network with redundant routes. Not being common on the Internet violates criterion one (Ct1: Widely Adopted). TFTP is not symmetrically. This means that a server is always just “serving requests” and not sending information actively to peers. This Request-Reply violates criteria (Ct3: Symmetrically built) and makes the protocol not a primary choice for message transport. The Protocol furthermore violates Ct2 (Ct2: Reliable) as it is based on UDP without any additional error correction.

16.5.3.4 MQTT

MQTT is an ISO standard (ISO/IEC PRF 20922:2016) and was formerly called MQ Telemetry Transport. The current standard as the time of writing this document was 3.1.1 [7].

The protocol runs by default on the two ports 1883 and 8883 and can be encrypted with TLS. MQTT is a publish/subscribe based message-passing protocol that is mainly targeted to m2m communication. This Protocol requires the receiving party to be subscribed to a central infrastructure in order to be able to receive messages. This makes it very hard to be used in a system without centralistic infrastructure and having no static routes between senders and recipients.

The protocol does satisfy the second criterion (Ct2: Reliable). It is in the area of end-user (i.e., Internet) not widely adopted, thus violating Criteria 1 (Ct1: Widely Adopted). In terms of decentralization design, the protocol fails as well (Ct3: Symmetrically built).

16.5.3.5 Advanced Message Queuing Protocol (AMQP)

The Advanced Message Queuing Protocol (AMQP) was initially initiated by numerous exponents based mainly on finance-related industries. The AMQP-Protocol is either used for communication between two message brokers, or between a message broker and a client[6].

It is designed to be interoperable, stable, reliable, and safe. It supports either SASL or TLS secured communication. The use of such a tunnel is controlled by the immediate sender of a message. In its current version 1.0, it does, however, not support a dynamic routing between brokers[6].

Due to the lack of a generic routing capability, this protocol is therefore not suitable for message transport in a generic, global environment.

The protocol satisfies partially the first criterion (Ct1: Widely Adopted) and fully meets the second criterion (Ct2: Reliable). However, the third criterion is violated due to the lack of routing capabilities between message brokers (Ct3: Symmetrically built).

16.5.3.6 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a communication Protocol which is primarily destined to m2m communication. It is defined in RFC7252[18]. It is defined as a lightweight replacement for HTTP in IoT devices and is based on UDP.

The protocol does partially satisfy the first criteria (Ct1: Widely Adopted). The second criterion (Ct2: Reliable) is only partially fulfilled as it is based on UDP and does only add limited session control on its own.

The main disadvantage in terms of a message transport protocol is that this protocol is not (like HTTP) symmetrically. This means that a server is always just “serving requests” and not sending information actively to peers. This Request-Reply violates criteria (Ct3: Symmetrically built) and makes the protocol not a primary choice for message transport.

16.5.3.7 Web Application Messaging Protocol (WAMP)

WAMP is a web-sockets based protocol destined to enable M2M communication. Like MQTT, it is publish respectively subscribe oriented. Unlike MQTT, it allows remote procedure calls (RPC).

The WAMP protocol is not widely adopted (Ct1: Widely Adopted), but it is reliable on a per-node base (Ct2: Reliable). Due to its RPC based capability, unlike MQTT, a routing like capability could be implemented. Symmetrical protocol behavior is therefore not available but could be built in relatively easy.

16.5.3.8 XMPP (jabber)

XMPP (originally named Jabber) is a synchronous message protocol used in the Internet. It is specified in the documents RFC6120[129], RFC6121[130], RFC3922[128], and RFC3923[127]. The protocol is a very advanced chat protocol featuring numeros levels of security including end-to-end signing and object encryption[127]. There is also a stream initiation extension for transferring files between endpoints [152].

It has generic routing capabilities spanning between known and unknown servers. The protocol offers a message retrieval mechanism for offline messages similarily to POP [112].

The protocol itself seems to be a strong candidate as a transport layer as it is being used actively on the Internet.

16.5.3.9 SMTP

The SMTP protocol is currently specified in [73]. It specifies a method to deliver reliably asynchronous mail objects through a specific transport medium (most of the time, the Internet). The document splits a mail object into a mail envelope and its content. The envelope contains the routing information, which is the sender (one) and the recipient (one or more) in 7-Bit ASCII. The envelope may additionally contain optional protocol extension material.

The content should be in 7-Bit-ASCII (8-Bit ASCII may be requested, but this feature is not widely adopted). It is split into two parts. These parts are the header (which does contain meta-information about the message such as subject, reply address, or a comprehensive list of all recipients), and the body which includes the message itself. All lines of the content must be terminated with a CRLF and must not be longer than 998 characters, excluding CRLF.

The header consists of a collection of header fields. Each of them is built by a header name, a colon, and the data. The exact outline of the header is specified in [123] and is separated with a blank line from the body.

[73] furthermore introduces a simplistic model for SMTP message-based communication. A more comprehensive model is presented in section **??** as the proposed model is not sufficient for a detailed end-to-end analysis.

Traditionally the message itself is mime encoded. The MIME messages are mainly specified in [50] and [51]. MIME allows to send messages in multiple representations (alternates), and attach additional information (such as possibly

inlined images or attached documents).

SMTP is one of the most common messaging protocols on the Internet (Ct1: Widely Adopted), and it would be devastating for the business of a country if, for censoring reasons, this protocol would be cut off. The protocol is furthermore very reliable as it has built-in support for redundancy and a thorough message design making it relatively easy to diagnose problems (Ct2: Reliable). All SMTP servers usually are capable of routing and receiving messages. Messages going over several servers are common (Ct3: Symmetrically built), so the third criterion may be considered as fulfilled as well.

SMTP is considered a strong candidate as a transport layer.

16.5.3.10 SMS and MMS

SMS capability was introduced in the SS7 protocol. This protocol allows the message transfer of messages not bigger than 144 characters. Due to this restriction in size, it is unlikely to be suitable for this type of communication as the keys being required are already sized similarly, leaving no space for Messages or routing information.

The 3rd Generation Partnership Project (3GPP) maintains the Multimedia Messaging Service (MMS). This protocol is mainly a mobile protocol based on telephone networks.

Both protocols are not widely adopted within the Internet domain. There are gateways providing bridging functionalities to the SMS/MMS services. However, the protocol itself is insignificant on the Internet itself.

16.5.3.11 MMS

This protocol is just like the SMS protocol accessible through the Internet by using gateways but not directly usable within the Internet.

16.5.4 Results

We have shown that all common M2M protocols failed mainly at Ct3 as there is no need for message routing. In M2M communication contacting foreign machines is not common. Therefore M2M protocolls are typically using static M2M communication over prepared channels. Such behavior is, however unsuitable for a generic messaging protocol.

Pure storage protocols fail at the same criteria as they typically have a defined set of data sources and data sinks, whereas usually at least the data sources are limited in number. This makes those protocols unsuitable again.

We can clearly state that according to the criteria, only a few protocols are suitable. Table **??** on page 105 shows that only SMTP and XMPP are suitable protocols. Eventually, similar protocols such as HTTP (with WebDAV) or FTP may be usable as well.

Criteria / Protocol	Ct1: Widely adopted	Ct2: Reliable	Ct3: Symmetrically built
HTTP	✓	✓	×
FTP	✓	✓	×
TFTP	×	×	×
MQTT	~	✓	×
AMQP	~	✓	×
CoAP	~	~	×
WAMP	×	✓	~
XMPP	✓	✓	✓
SMTP	✓	✓	✓

Table 16.1: comparison of protocols in terms of the suitability as transport layer

The findings of this short analysis suggested that we should use the two protocols, SMTP and XMPP, for our first standardization. We require at least two to prove that the protocol is agnostic to the transport.

17 Transport Layer Implementation

17.1 Implementation of a Dummy Transport Layer

For better diagnosability and fast setup, we implemented a custom transport layer working on a config-less manner in a localhost or broadcast-domain environment. The transport layer is based on the Hazelcast distributed hashmap. Implementation may be found under `net.messagevortex.transport.dummy.DummyTransportTrx`.

17.2 Implementation of an Email Transport Layer

Email supports a conglomerate of protocols. Looking at the client side, we will find that email is sent with an authenticated SMTP connection. The SMTP connection is somewhat different than than the connections used to send emails to the destination. First of all the client port was shifted in the past to a specific submission port (SMTPS: Port 465; Submission: Port 587). These submmission ports are authenticated (either by username and password, by IP or by certificates) and usually privileged (no UBM checks). On the retrieval side, SMTP is not capable of handling these tasks sensibly. Instead, POP3 and IMAPv4 are used. POP3 is a deposit box for email where a device fetches the mail and stores it locally. This is commonly in use for automated processing of mails but these days, where the same user owns multiple devices no longer adequate. IMAPv4 offers to organize mails on the server. This allows a user to have the same folder structure of mails in a synchronized manner on all devices.

For an ideal implementation we would do the following: Organize our *MessageVortex* mails in a separate account. The account is accessed through a local proxy relaying our "ordinary outgoing mails" through the SMTP server of our regular provider and all *MessageVortex* related traffic through the provider of our *MessageVortex* mailbox. Keeping the two mailboxes separate is sensible and important as we will see in part VI. The housekeeping on the account used for *MessageVortex* is done automatically and in a sensible way, comparable to a human (e.g., handle draft, sent and trash bin folders sensibly and keep all mails in a flat structure deleting old mails from time to time). The mails from the regular and the *MessageVortex* account are merged by the proxy in a transparent way. Keeping the messages apart but offering a unified look.

We were unable to find any scientifical data regarding what type of traffic or attachment is common on the Internet. We, therefore, tried to analyze mail logs (SMTP) of a mail provider. We were scanning 567594 emails for attachment properties after the spam elimination queue. 16.5% of all scanned messages had an attachment. The top 20 attachment types distributions are shown in table 17.1.

Type	%
image/jpeg	27.4
application/ms-tnef	13.7
image/png	13.3
application/pdf	10.7
image/gif	7.4
application/x-pkcs7-signature	5.4
message/rfc822	7.0
application/msword	3.1
application/octet-stream	3.0
application/pkcs7-signature	2.3
application/vnd.....wordprocessingml.document	1.4
message/disposition-notification	1.1
application/vnd.ms-excel	0.8
application/vnd.....spreadsheetml.sheet	0.6
application/zip	0.5
application/x-zip-compressed	0.5
image/pjpeg	0.4
application/pkcs7-mime	0.4
video/mp4	0.4
text/calendar	0.4

Table 17.1: Distribution of top 20 attachment types

As expected, the number of images within mail was very high ($\approx 50\%$). Unfortunately, we were unable to analyze the content of ms-tnef attachments retrospectively. It seems that based on these figures, information hiding within images in email traffic is a good choice.

For our implementation, we worked with F5 blending into jpeg images, as this choice seemed to undermine credible content based on table 17.1.

In our current implementation the house keeping part has been skipped. Instead, we are just fetching the newly arrived messages and put in a local storage. The email presented to the client is provided by a local IMAP server. Persistence of these messages is not yet implemented.

17.3 Implementation of an XMPP Transport Layer

The XMPP protocol (formerly called Jabber, as specified in [129], is natively not capable of transferring anything else but text messages. Unlike email, XMPP is capable of true end-to-end signing and object encryption without solving the problem of the initial trust. While we may use the end-to-end encryption for additional security, relying on this feature is not sensible as we would put trust into the security features of an intermediate node. This would effectively violate RQ3 (zero trust) requirement. We decided to use the extension defined in [133] to transfer our messages, as it is simple and reliable.

To transfer a *VortexMessage*, we could embedd a MIME message just as with SMTP. While this would be technically feasible, the usage of MIME is not common and even discouraged. Instead, the inner structure if an XMPP message relies on XML.

XMPP has an improvment process based on XEPs. For including binary contents such as attachments in messages multiple XEPs exists. Table 17.2 shows all idenified candidates.

Name	Status (as of 06-2020)	Purpose
XEP-0047: In-band bytestreams[70]	Final Standard	Allows sending chunked, base64 encoded data within the jabber connections.
XEP-0066: Out of Band Data[132]	Draft Standard	Allows sending URIs of remotely hosted binary data.
XEP-0096: SI File Transfer[152]	Depreciated (ref. XEP-0234)	Improvement of [132] allowing to send metadata and alternative URIs
XEP-0135: File Sharing[131]	Deferred (inactive)	Inband or Out-of-band file discovery and referral service. May be used in conjunction with FTP, HTTP, SCP, or [152].
XEP-0231: Bits of binary[133]	Draft Standard	Allows sending inband small unchunked files and referring within the message similarly to [92].
XEP-0234: Jingle File Transfer[134]	Deferred (inactive)	Based on [88] allowing out-of-band content negotiation of complex data streams

Table 17.2: Overview of XEPs related to transporting binary data.

Relevant documents have either reached standard, draft standard, or have been deferred due to inactivity. We used "*XEP-0231: Bits of binary*"[133] for our protocol. It is simple to implement as a transport layer, used in many clients (e.g., Prosody, Pigdin, or CoyIM), and already aa draft standard which minimizes the risk of using later deprecated technology. As it is a client-only XEP, a node may use any XMPP server regardless of any additional support for XEP-0135.

Embedding works exactly the same as with email with the same supported blending options. Instead of searching all attachments, we just search through all data

objects for relevant *VortexMessages*.

The blending layer may generate decoy message analogue to the messages generated in the case of email. Some adoptions in terms of texts might be advisable.

17.4 Distributed Configuration and Runtime Store of processing content

A distributed storage is advisable if working as a reliable service. This is why we defined ASN.1 structures for all elements kept in memory as shown in figure 1. Wisely applied, they may be used to store in a transport storage for access of a redundant set of devices all maintaining the same set of data.

```
-- States reflected:
--   Tuple()=Val()
--           [vallidity; allowed operations]{Store}
--
-- - Tuple(identity)=Val(messageQuota,transferQuota,
--   sequence of Routingblocks for Error Message
--   Routing) [validity; Requested at creation; may
--   be extended upon request] {identityStore}
-- - Tuple(Identity,Serial)=maxReplays ['valid' from
--   Identity Block; from First Identity Block; may
--   only be reduced] {IdentityReplayStore}

MessageVortex-NonProtocolBlocks DEFINITIONS
                                EXPLICIT TAGS ::=
BEGIN
  IMPORTS PrefixBlock, InnerMessageBlock,
          RoutingBlock,
          maxWorkspaceID
                FROM MessageVortex-Schema
          UsagePeriod, NodeSpec, BlendingSpec
                FROM MessageVortex-Helpers
          AsymmetricKey
                FROM MessageVortex-Ciphers
          RequirementBlock
                FROM MessageVortex-Requirements;

  -- maximum size of transfer quota in bytes of an
  -- identity
  maxTransferQuota          INTEGER ::= 4294967295
  -- maximum # of messages quota in messages of an
  -- identity
  maxMessageQuota           INTEGER ::= 4294967295

  -- do not use these blocks for protocol encoding
  -- (internal only)
  VortexMessage ::= SEQUENCE {
    prefix        CHOICE {
        plain           [10011] PrefixBlock,
        -- contains prefix encrypted with receivers
                -- public key
        encrypted       [10012] OCTET STRING
    },
    innerMessage  CHOICE {
        plain     [10021] InnerMessageBlock,
        -- contains inner message encrypted with
        -- Symmetric key from prefix
        encrypted [10022] OCTET STRING
    }
  }

  MemoryPayloadChunk ::= SEQUENCE {
    id              INTEGER (0..maxWorkspaceID),
    payload         [100] OCTET STRING,
    validity        UsagePeriod
  }

  IdentityStore ::= SEQUENCE {
    identities SEQUENCE (SIZE (0..4294967295))
               OF IdentityStoreBlock
  }

  IdentityStoreBlock ::= SEQUENCE {
    valid           UsagePeriod,
    messageQuota    INTEGER (0..maxMessageQuota),
    transferQuota   INTEGER (0..maxTransferQuota),
    -- if omitted this is a node identity
    identity        [1001] AsymmetricKey OPTIONAL,
    -- if ommited own identity key
    nodeAddress     [1002] NodeSpec      OPTIONAL,
    -- Contains the identity of the owning node;
    -- May be ommited if local node
    nodeKey         [1003] SEQUENCE OF AsymmetricKey
                           OPTIONAL,
    routingBlocks   [1004] SEQUENCE OF RoutingBlock
                           OPTIONAL,
    replayStore     [1005] IdentityReplayStore,
    requirement     [1006] RequirementBlock OPTIONAL
  }

  IdentityReplayStore ::= SEQUENCE {
    replays   SEQUENCE (SIZE (0..4294967295))
              OF IdentityReplayBlock
  }

  IdentityReplayBlock ::= SEQUENCE {
    identity          AsymmetricKey,
    valid             UsagePeriod,
    replaysRemaining INTEGER (0..4294967295)
  }

END
```

Listing 1: Definition of the structures related to a distributed storage

The configuration should be stored sensibly in the transport storage to match regular usage patterns. A good storage may be organized as follows:

- All configuration items are blended with F5 and protected by a key phrase to be encrypted with an appropriate KDF.
- Draft folder contains one draft message with the current, short living configuration.
- Long living configuration is written to draft and then moved to the Sent Items folder.
- A configuration is first fetched from the drafts folder and then the first config object of the “sent items” folder is fetched.
- All items in the sent folder are deleted after a defined timespan (say 30 days). Items not yet expired are rewritten into a new config object into the sent folder prior delete.

18 Blending Layer Implementation

18.1 Embedding Spec

We always embed VortexMessages as attachments in SMTP and XMPP messages.

The embedding supports some properties. A receiving host chooses the supported properties. We describe valid properties by the blending specification in listing 2.

```
1 plainEmbedding = "("plain:"<#BytesOffset>[,<#BytesOffset>]*")
2 F5Embedding    = "(F5:"<passwordString>[,<PasswordString>]*")"
```

Listing 2: Definition of the embedding specs

Both specifications allow embedding of a *VortexMessage* and are described in the following section. In both cases a byte stream is extracted consisting of a prefix block containing the peer key K_{peer_o} immediately followed by the symmetrically encrypted InnerMessageBlock as described in figure 3. The string is not necessarily terminated correctly. The presence of a valid PrefixBlock signals an existing *VortexMessage* to the blending layer. That way we make sure that the size of a potential message leakes its presence. To detect the presence of a *VortexMessage*, the hosts private key $K^{-1}_{host_o}$ for decoding the is required.

18.1.1 Extraction of the Blended Message

In this section we describe the extraction of a VortexMessage by the blending layer. We describe the Plain Embedding alolowing a detectable yet unreadable

```
PrefixBlock ::= SEQUENCE {
  version          [0] INTEGER OPTIONAL
  key              [2] SymmetricKey,
}

InnerMessageBlock ::= SEQUENCE {
  padding    OCTET STRING,
  prefix     CHOICE {
      plain           [11011] PrefixBlock,
      -- contains prefix encrypted with receivers
      -- public key
      encrypted       [11012] OCTET STRING
  },
  header  CHOICE  {
    -- debug/internal use only
    plain     [11021] HeaderBlock,
    -- contains encrypted identity block
      encyrpted [11022] OCTET STRING
  },
  -- contains signature of Identity [as stored in
  -- HeaderBlock; signed unencrypted HeaderBlock without
  -- Tag]
  identitySignature OCTET STRING,
  -- contains routing information (next hop) for the
  -- payloads
  routing      [11001] CHOICE {
    plain      [11031] RoutingBlock,
    -- contains encrypted routing block
    encyrpted [11032] OCTET STRING
  },
  -- contains the actual payload
  payload      SEQUENCE (SIZE (0..maxPayloadBlks))
               OF OCTET STRING
}
```

Listing 3: Definition of the outer message blocks.

message including the chunking applied to minimize detection. Furthermore we describe the more elaborated methode of using F5 blending, which results in undetectable messages at the cost of roughly eight times higher protocol overhead.

18.1.2 Plain Embedding

In this section we explain *plainEmbedding* and how *VortexMessages* with *plainEmbedding* may be extracted. This embedding is mainly suitable for simple, observable message transferal.

The *plainEmbedding* is a simple embedding replacing parts of the original file with the content of the *VortexMessage*. To maintain the header information of a file, we introduced an offset as a set of fixed values. Plain embedding is simply detectable. While offset and chunking may allow us to maintain a valid file structure, the original content of the file is, however, normally destroyed. We use plain embedding mainly for our experiments. For better readability, we used a specialized blending layer using unchunked, plain embedding with an offset of 0. The decoy message is the ASN.1 block representation of the encoded block. The chosen encoding simplified to see the inner workings of the protocol. For production use, we apply F5 embedding with a generated payload. The current implementation of the blending layer employing plain embedding is thus not suitable for production use as the messages remain identifiable or at least suspicious.

Chunking of Plain Embedded Messages

In this section we describe the chunked embedding into plain messages. Chunking is done by pre-pending two numeric values to a data chunk. The first number (modulo the remaining number of bytes of the file) reflects the size of the chunk immediately following the second value. The second value (again modulo the same number) reflects the number of bytes to be skipped after the chunk for getting to the next header.

Each value is encoded in one to four bytes forming an integer value. The first seven bits are the least significant bits of the value. If the eight bit is set, we signal

an additional relevant octet. The second and third byte (if any) are interpreted equivalently. The fourth byte is always interpreted without any signal bit. Instead the eight bit is used as the most significant bit in that case. All bits collected together form an integer value. This value is taken modulo the remaining bytes of the file, starting with the first byte of the first header value. The chain formed by these headers have no terminator and may surpass the file end.

The byte layout is chosen in such a way that any byte sequence from two to eight bytes form a valid chunk header. The lack of termination guarantees that no information is leaked through the interpretation of any header.

Table 18.1 shows a number of valid chunking header bytes and their interpretation (without the modulo). Listing 4 shows an implementation of the algorithm.

```
1  long i=0;
2  unsigned char b=0;
3  char m;
4  char c=0;
5  do {
6          b=getNextByte();
7          if ( c<3 ) {
8                  m = 127;
9          } else {
10                 m = 255;
11         }
12         i = i | (long)((m & b) << (7*c));
13         c=c+1;
14         printf( "got 0x%02x; new value is %d (byte:%d)\n",b,i,c );
15 } while ((c<4) && ((b & 128)==128));
16 printf( "RESULT:%d\n\n",i);
```

Listing 4: Reference implementation for extraction of a chunking value in C

bytes				Results
0x83	0x0a			1283
0x81	0x00			1
0xfb	0x01			251
0x00				0
0x77				119
0xaa	0xaa	0xaa	0xaa	357209386
0xff	0xff	0xff	0xff	536870911

Table 18.1: Example interpretation of bytes in offsets

When plain embedding messages we have the problem that most of the files have recurring logical structures. Such structures should not be broken when embedding in plain as such embedding as it would leak in an easy detectable manner the presence of a broken file.

18.1.2.1 Implementation of F5 Blending

In this section we introduce the implementation of F5 blending. It is a better blending than the rather simple $plainBlending$ discused in the previous section. While F5 is very old (2002, it has not been broken. In the reference implementation of F5 was a detectable unintentional double compression[48]. The authors of the

reference implementation fixed this issue [20] and we were unable to find newer breaches. Newer derivates, such as nsF5[54] or MSET[67], were proposed. We did, however, not consider these as candidates as an appropriate reference implementation seemed to be unavailable.

F5 hides its information in JPEG, BMP, and GIF images by matrix encoding its information in the image data. According to [160] it has a capacity exceeding 13% of the steganograms' size.

The implementation of F5 uses a "password" for the initialization of the random number generator. Without this password the extracted message is random. As a *VortexMessage*is encrypted, we were unablein our analysis to differentiate random output from a *VortexMessage*. Only decoding with the host key $K^{-1}_{host_o}$ resulted in detection of a *VortexMessage*.

As shown in listing 2, we publish this password and keep detection to the decoding part of our blending layer. In therory we could have kept this password specific to the eID. This would, however, increase the decoding complexity and the password would be needed by the node blending the content, which would leak a synonym to the eID used on the next host to the current host.

18.1.3 Message Processing by the Blending Layer

If a *VortexMessage*is detected, the $prefix$ with the sender key K_{sender_o} is decrypted to decrypt the header block $header$. Verfying the identity signature (which may be done even before decrypting the header block) guarantees that the original sender is the owner of the eID. WIth the help of the accounting layer the *VortexMessage* is authorized for processing.

Depending on the current quota (messages) and the status of the identity (temporary or established) further procesing by the routing layer is acknowledged. For an overarching description of the whole message processing see section 19.2.

18.2 Blending Decoy Content Generation

The decoy content of a message is an important part for the *MessageVortex*system. It creates meaningful content for the traffic to be hidden within.

When using F5 or similar mechanisms for blending, we decided to make sure that our content does not have to pass a turing test. Normal email conversations is two way and has many properties such as references to previous messages and similar contexts. In order not to fall in such traps, we use common machine generated one way messages with generated images. Examples for such messages are password recovery requests with gravatars or monitoring messages with generated graphs (such as current running processes on a system). Such messages are easy to generate in various sizes and are obviously machine generated.

To make it harder for an attacker to identify context of messages the sending address on the transport media should not be equal to the receiving address. This makes generation of interaction graphs much harder as we will see in section 29.1.2.7.

19 Routing Layer Implementation

In this chapter we describe the routing layer as our main workhorse for processing *VortexMessages*. The routing layer keeps a workspace for each eID and discards old or unused entries. When receiving routing blocks, it processes those and generates new messages. We, furthermore, shed light on some decisions specific to our implementation such as encoding formats or message layout.

19.1 ASN.1 DER encoding scheme for *VortexMessages*

Originally, we implemented the protocol as XML encoded messages. This encoding had, however, several flaws. First the huge ammount of encrypted data within the document made the messages bulky and at the same time loose one of its main strengths: readability for humans. The encoding required for binary data caused messages to increase ion size due to their onionized structure.

Furthermore, the some XML features such as external entities or the possibility to define tags introduced a series of new possible attacks such as DoS attacks (e.g., a Billion Laughts) or information stealing attack (e.g., xxe attacks). Furthermore XML stuctures are hard to sign and have many possible ways of layouting data.

To counter these downsides, we reimplented our client with ASN.1 based DER encoding. This type of encoding fits well the purpose of encrypted structures and is commonly used for related tasks such as key storage or signing maessages and certificates.

DER encoding of ASN.1 structures even enables us to foresee the content of an encoded message down to each bit. This is important as it enables in depth analysis of message flows as we will see in section 24.4.

ASN.1 offers three common encoding schemes:

- BER (Basic Encoding Rules)
- CER (Canonical Encoding Rules)
- DER (Distinguished Encoding Rules)

Where DER and CER are a subset of BER being more strictly defined. We decided to go with DER as this ruleset was available in the library used.

19.2 Processing of messages

In this secion we focus on the processing of messages. Messages are processed either upon their arrival or if a routing block is processed. The processing of a routing block is typically relative to the delivery of the message containing the routing block. as an immediate result of processing a routing block a new message is generated for a routing block or a message for the current node.

19.2.1 Workspace Layout

The workspace itself contains payload blocks assigned to workspace IDs. The ID space is divided into three parts as shown in table 19.1.

ID			Purpose
0			Message for local delivery
1	-	127	Payload block of current routing block
128	-	32766	Reserved
32767			Reply block
32768	-	65535	Payload block in workspace

Table 19.1: Workspace layout of IDs

19.2.2 Processing of Incoming Messages

In this section we focus on the operations carried out by a routing layer on each message extracted by the blending layer.

A message extracted by the blending layer is passed to the routing layer for further processing. The source of the message (e.g., protocol of the message or sender address) is irrelevant and discarded by the blending layer.

First step of the processing is the extraction of the identity. The identity can be found in the header block (see figure 5 $identityKey$) and then verified with the signature $identitySignature$ (figure 3)

If verification is successful the message is authenticated but not necessarily ready for further processing. Unless the Header contains an identity creation request, next step is then the authorization for further processing. For proper authentication the following preconditions must be met:

- Message must be outside a replay blocking intervall
- The identity is not temporary (19.3.1)

```
HeaderBlock ::= SEQUENCE {
  -- Public key of the identity representing this
  -- transmission
  identityKey          AsymmetricKey,
  -- serial identifying this block
  serial               INTEGER (0..maxSerial),
  -- number of times this block may be replayed
  -- (Tuple is identityKey, serial while
  -- UsagePeriod of block)
  maxReplays           INTEGER (0..maxNumOfReplays),
  -- subsequent Blocks are not processed before
  -- valid time.
  -- Host may reject too long retention.
  -- Recomended validity support >=1Mt.
  valid                UsagePeriod,
  --  contains the MAC-Algorithm used for signing
  signAlgorithm        MacAlgorithmSpec,
  -- contains administrative requests such as
  -- quota requests
  requests             SEQUENCE
                       (SIZE (0..maxNumOfRequests))
                       OF HeaderRequest ,
  -- Reply Block for the requests
  requestReplyBlock    RoutingCombo,
  -- padding and identitifier required to solve
  -- the cryptopuzzle
  identifier [12201] PuzzleIdentifier OPTIONAL,
  -- This is for solving crypto puzzles
  proofOfWork[12202] OCTET STRING OPTIONAL
}

RoutingBlock ::= SEQUENCE {
  -- contains the routingCombos
  routing     [331] SEQUENCE
                      (SIZE (0..maxRoutingBlks))
                    OF RoutingCombo,
  -- contains the mapping operations to map
  -- payloads to the workspace
  mappings    [332] SEQUENCE
                    (SIZE (0..maxPayloadBlks))
                    OF MapBlockOperation,
  -- contains a routing block which may be used
  -- when sending error messages back to the quota
  -- owner this routing block may be cached for
      -- future use
  replyBlock [332]   SEQUENCE {
       murb            RoutingCombo,
       maxReplay       INTEGER,
       validity        UsagePeriod
  } OPTIONAL
}

RoutingCombo ::= SEQUENCE {
  -- contains the period when the payload should
  -- be processed.
  -- Router might refuse too long queue retention
  -- Recommended support for retention >=1h
  minProcessTime INTEGER
                      (0..maxDurationOfProcessing),
  maxProcessTime INTEGER
                      (0..maxDurationOfProcessing),
  -- The message key to encrypt the message
  peerKey        [401] SEQUENCE
                         (1..maxNumOfReplays)
                       SymmetricKey OPTIONAL,
  -- contains the next recipient
  recipient      [402] BlendingSpec,
  -- PrefixBlock encrypted with message key
  mPrefix        [403] SEQUENCE
                         (1..maxNumOfReplays)
                       OF OCTET STRING OPTIONAL,
  -- PrefixBlock encrypted with sender key
  cPrefix        [404] OCTET STRING OPTIONAL,
  -- HeaderBlock encrypted with sender key
  header         [405] OCTET STRING OPTIONAL,
  -- RoutingBlock encrypted with sender key
  routing        [406] OCTET STRING OPTIONAL,
  -- contains information for building messages
  -- (when used as MURB)
  -- ID 0 denotes original/local message
  -- ID 1-maxPayloadBlks denotes target message
     -- ID 32767 denotes a solicited reply block
  -- 32768-maxWorkspaceId shared workspace for all
  -- blocks of this identity)
  assembly       [407] SEQUENCE
                 (SIZE (0..maxAssemblyInstr))
                 OF PayloadOperation,
  -- optional for storage of the arrival time
  validity       [408] UsagePeriod OPTIONAL,
```

Listing 5: Definition of the inner message blocks.

If the identity is not temporary header requests are executed upon authorization. The only header request executed on a temporary eID is a *createIdentity* request.

As soon as the header requests are executed, the content is processed. The operations in the routing block are added to the workspace and the mapping operations remain in the routing combo.

19.2.3 Processing of Outgoing Messages

In this section, we focus on the creation of new messages sent to a next hop router. The message creation is triggered on a timed manner based on the content of the *RoutingCombo* and then passed to the blending layer for blending.

The processing of a sending block is triggered by a routing block in the workspace, as shown in figure 19.2. The assembly instructions are processed to collect the payload blocks. Then the encryption is applied to the message and passed on to the blending layer for processing.

All mapping operations are then carried out. If a payload has not yet been calculated, appropriate operations in the workspace are searched and executed to

create the missing payloads. If a payload is not created successfully the payload in the message is omitted.

The message is assembled by building the $InnerMessageBlock$ with $cPrefix$, $header$, $routing$ from the routing combo and the payloads generated. This block is DER encoded and then encypted with $peerKey$ the resulting octet stream is prepended with $mPrefix$ from the routing combo and then passed to an appropriate blending layer for the requested transport using $blendingSpecc$.

The resulting message is a valid *VortexMessage*but the generating node has no relevant knowledge about the message or its content except for the recipient address.

19.2.4 Implementation of Operations

In this section we focus on the implemented operations. The operations outlined in section 15.2.6 were implemented in exact the described manner. Additionally we implemented a mapping operation, copying the content of one payload ID to another one. The implementation and its test showed some weaknesses related to platform and implementation specifics, which are outlined further down.

For our implementation we used a HashMap to keep a list of all operations. Key of the HashMap is the output id of the resulting operations. Instead of proactively executing all operations to get all possible payload IDs, we build a dependency tree of all required prerequisites. A caching structure allows us to efficiently work with the results of all operations. If an operation expires, all cached output of the respecthive operations is invalidated. If a payload block expires or is overridden, all outputs taking input from this payload directly or indirectly are invalidated. This allows us to keep a very efficient and compact representation of the payload space not wasting any memory without necessity.

The mapping operation became a necessity when defining the system of specialized IDs as outlined in table 19.1. This usage of specialized workspace IDs make the mapping of values from one ID to another one a necessity. While theoretically doable in a two-step operation by applying an operation and its reverse, the mapping operation is far more efficient.

Some operations showed weaknesses. The splitPayload Operation was mathematically well designed. Due to differences in floating point calculations (FP ops)when carried out on ARM and AMD based platforms, the result may differ when working with this operation. As an immediate result we defined that all FP ops must be carried out as specified in [1]. This allows us to have exact the same output of the splitting operation on all platforms and thus a constant result. Luckily in java such behavior may be achieved by applying the `strictfp` keyword, which saved a lot of troubles and work.

Another problem which arose in practice was that applying a Gauloise field (GF) in the $addRedundancy$ and $removeRedundancy$ operations different to 8 or 16 make practical problems due to their resulting sizes. To apply the math matching a

computer working with primarily 8 bits per byte only, only the the values 8 and 16 are easy to realize. While we did not want to remove this capability, we added a possibility for the node to signal which sizes of GFs are supported.

A GF of size not equal to 8 or 16 requires the operation first to reorganize the data in appropriate bit numbers, then apply thef GF operations and after converting back realigning with 8 bits.

19.3 Request handling

In this section we focus on the handling of requests and the replies to requests required by the protocol. As the replies are required, but need to have the same properties as normal messages, we absolutely needed routing blocks for replies.

In general any host may send a request to any other host. These requests normally involve the requirement for sender anonymity. The request itself is included in the HeaderBlock. The reply block provided in $requestReplyBlock$.

Requests are identified shown in figure 6. The tagging of the requests is necessary to identify the request provided.

```
1  HeaderRequest ::= CHOICE {
2    identity       [0] HeaderRequestIdentity,
3    capabilities   [1] HeaderRequestCapability,
4    messageQuota   [2] HeaderRequestIncreaseMessageQuota,
5    transferQuota  [3] HeaderRequestIncreaseTransferQuota,
6    quotaQuery     [4] HeaderRequestQuota,
7    nodeQuery      [5] HeaderRequestNodes,
8    replace        [6] HeaderRequestReplaceIdentity
9  }
```

Listing 6: Definition of a request

The routing blocks for replies must be different for normal routing blocks as they may otherwise be misused as ordinary sending blocks. A reply block for request should always map to payload ID 32767. Whereas a reply block for a normal user (to keep sender anonymity) always should map in workspace ID 0. That way it is impossible to misuse reply blocks for normal messages.

A reply is sent as a special message block and must be mapped to workspace ID 128. A *VortexNode* may accept a special block delivered to ID 0, but such behavior should never be assumed. Figure 7 shows the definition of a reply. A reply is expressed in a special block. This special block contains a status of the request is either OK or a failure and may provide aditional information such as the outcome of a request.

```
SpecialBlock ::= CHOICE {
  capabilities   [1]  ReplyCapability,
  requirement    [2]  SEQUENCE (SIZE (1..127))
                      OF RequirementBlock,
  quota          [4]  ReplyCurrentQuota,
  nodes          [5]  ReplyNodes,
  status         [99] StatusBlock
}

StatusBlock ::= SEQUENCE {
  code             StatusCode
}

StatusCode ::= ENUMERATED {

  -- System messages
  ok                              (2000),
  quotaStatus                     (2101),
  puzzleRequired                  (2201),

  -- protocol usage failures
  transferQuotaExceeded           (3001),
  messageQuotaExceeded            (3002),
  requestedQuotaOutOfBand         (3003),
  identityUnknown                 (3101),
  messageChunkMissing             (3201),
  messageLifeExpired              (3202),
  puzzleUnknown                   (3301),

  -- capability errors
  macAlgorithmUnknown             (3801),
  symmetricAlgorithmUnknown       (3802),
  asymmetricAlgorithmUnknown      (3803),
  prngAlgorithmUnknown            (3804),
  missingParameters               (3820),
  badParameters                   (3821),

  -- Mayor host specific errors
  hostError                       (5001)
}
```

Listing 7: Definition of a request

19.3.1 Requesting a new Ephemeral Identity

One of the main requests for the protocol is the request for generating a new ephemeral identity. The goal of this operation is to create a non-hijackable workspace on a node while remaining anonymous. If having multiple eIDs on the same host they must be unlinkable. Furthermore it should be hard for an adversary to flood a *VortexNode* with workspace requests to make a denial-of–service (DoS) attack.

```
HeaderRequestIdentity ::= SEQUENCE {
  period UsagePeriod
}
```

Listing 8: Definition of an identity request

Requesting a new identity is easy. The only information required is the lifetime requested (see listing 8). A *VortexNode*may do any of the following operations:

- Deny the request (may be without an error message)
- Accept the request without a “puzzle”
- Accept the request under the condition a “puzzle” is solved

Denial of a request does not necessarily lead to an error message. A *VortexNode* sends only an error message if the node is a public node. All other nodes (stealth and hidden; see sectionsec:vortexNodeTypes) do not send an error message in order not to leak their existence.

If a request is accepted, the *VortexNode*replies either with an “ok” or with a “puzzle required” status.

As currently supported puzzles two possible answers are forseen by the protocol:

```
RequirementBlock ::= CHOICE {
  puzzle   [1] RequirementPuzzleRequired,
  payment  [2] RequirementPaymentRequired
}

RequirementPuzzleRequired ::= SEQUENCE {
  -- bit sequence at beginning of hash from
     -- the encrypted identity block
  challenge      BIT STRING,
  mac            MacAlgorithmSpec,
  valid          UsagePeriod,
  identifier     INTEGER (0..4294967295)
}

RequirementPaymentRequired ::= SEQUENCE {
  account        OCTET STRING,
  ammount        REAL,
  currency       Currency
}

Currency ::= ENUMERATED {
  btc       (8001),
  eth       (8002),
  zec       (8003)
}
```

Listing 9: Definition of a requirement

- Solving a CPU bound hash puzzle
- Paying a fee in a digital currency

The CPU bound puzzle is a hash based puzzle. The *VortexNode* provides a bit string for the identity. The header has to be resent in such a way that the requested hash of the DER encoded header starts with the bit sequence provided. there are two ways of keeping track of these puzzles:

- Generate puzzles in a reproducible way
  This is the more elegant way of puzzles. Instead of keeping track of the puzzles, we generate the hash by applying the following function $left(MAC(K^1_{identity}| < \text{host secret} > | < \text{date and hour} >), \text{hourly complexity in bits})$. This method has some ups and downs. On the positive side we do not need to keep track of all puzzles provided to identities. Instead we just check, if a puzzle provided matches an appropriate challenge of the last hours. This host cannot be flooded with identity creation requests as it does not need to keep track of the requests. Instead it must keep a list of successful serials that requested a quota increase as there it would be possible to replay the request in order to increase the quotas. This is not comparable to the costs for an attacker as we have only to keep a list of integers where the PoW has been solved.

- Store random puzzles during their validity time
  This method is straight forward. It requires an entry in a table per puzzle and only for the lifetime considered.

The second approach has a huge downside: A DoS attack is feasible. Given the fact that we need to store the key (1KB max) , the date and time of expiry 4 bytes (epoch), and the bit sequence (up to 8 bytes). Means that we require millions of requests to flood a host. Since the keys do not need to be strong (an adversary does not intend to use them; It is just a DoS attack), this attack is doable with considerable efforts. This is why we favor the first approach.

19.3.2 Replacing an Existing Node Specification or Proving a Sender Identity

As users tend to change transport layer addresses, keys might become insecure, or transport services are no longer available, we need means of upgrading keys or replacing them with newer transport addresses. This may be done with a $HeaderRequestReplaceIdentity$ request as shown in figure 10. This request allows in a cryptographically secured way to exchange keys and transport endpoints by the respective owners.

```
1  HeaderRequestReplaceIdentity ::= SEQUENCE {
2    replace           SEQUENCE {
3        old             NodeSpec,
4        new             NodeSpec OPTIONAL
5    },
6    identitySignature OCTET STRING
7  }
```

Listing 10: Definition of an identity replace request

By signing the request the sender proves that he is in possession of the old key. By omitting the new node specification, a user may bind an existing eID to a real world identity. This is useful for securing endpoint identities if required. Such a secured identity should, however, only be used for endpoint messages and not for routing as this would shorten the secured path of the message.

A *VortexNode* may reply with a “quotaStatus” message, if the node owner decides to assign different (possibly unlimited) quota to the identity.

19.3.3 Replacing an Existing Reply Block

For sender anonymity, a sender may provide a reply block for single or multiple uses (SURBS and MURBS). These routing blocks use eIDs, which have by definition a limited lifespan. In this section we focus on the implementation details for requests replacing such reply blocks.

A routing block has a limited lifespan which is directly limited by the eIDs involved. The first expiring eID invalidates the block unless redundant paths are included. In this case only redundancy would be reduced. To keep a message intact even if a reply block of an anonymous sender expires, the sender may replace any existing reply block with a new routing block.

To replace an existing routing lock with a new one it is sufficient to send an empty message to the respective eID. Within the routing block we provide one or more new $replyBlock$ replacing all old existing ones. As the header is signed by the private key of the eIDs owner this operation is safe.

20 Accounting Layer Implementation

The accounting layer is keeping track of all operations allowed to a message. In this section we list the tasks fulfilled by the accounting layer and make a precise outline of them.

The accounting layer Keeps a list of the following information:

- **eID**$[]\langle expiry, pubKey, mesgsLeft, bytesLeft\rangle$

- **Puzz**$[]\langle expiry, requestHeader, puzzle\rangle$
  or
  Puzz$[]\langle dateAndHour, puzzleSizeNewIdentity, basePuzzleSizeQuta\rangle$

- **Replay**$[]\langle expiry, serial, numberOfRemainingUsages\rangle$

The list of all eIDs is kept in the accounting layer together with their quotas and expiry. The accounting layer triggers the deletion of the workspace assigned to it upon its expiry. Each eID has assigned two quotas. The $messageQuota$ limits the ammount of messages containing payload blocks to be routed. This quota is measured upon arrival of a message (inbound only). The $bytesLeft$ quota is a sizing quota and is measured outbound. This quota is applied on all outbound messages regardless of their content.

The $puzz[]$ list with $requestHeaders$ is only required if relying on random user puzzles. This would lead to an implementation which is simple but may be flooded with eID requests. The second list requires only an entry per hour. The number of entries is limited by the number of hours a puzzle is accepted. The puzzle is built by calculating $left(MAC(K^1_{eID}|globalSecret|dateAndHour), puzzleSizeNewIdentity)$. That way a DoS attack by flooding the puzzle table is no longer feasible.

The last list to be kept is the list for replay protection $Replay[]$. This list is a list of serials and their remaining usages is an effective replay protection. A serial is only allowed to be processed if the serial has not reached the maximum number of replays. As a header block has typically only a limited lifespan this is a very short list. Flooding is not very effective as a host may limit the number of entries in this list. The only identity suffering from that measure would be the identity assigned to the serial as serials from this eID would suffer incomplete replay protection and thus endanger its quotas and .

In table 20.1, we show under what circumstances a reply to a header request should be sent. The capitalized words MAY, MUST, SHOULD, and SHOULD NOT are used as defined in RFC2119[19].

Request \ Criteria	unknown identity cleartext	unknown identity encrypted	expired identity encrypted	known identity encrypted
newIdentity	SHOULD NOT	MAY	Invalid (Error)	Invalid (Error)
queryPeer	MUST NOT	MUST NOT	MAY	MAY
queryCapability	SHOULD NOT	MAY	MAY	MUST
messageQuota	MUST NOT	MUST NOT	MAY	MUST
transferQuota	MUST NOT	MUST NOT	MAY	MUST

Table 20.1: Requests and the applicable criteria for replies

21 Usability Related Implementation Details

Usability is one of the foremost criteria for user acceptance. As we have no chance to create a nice User interface competing with existing ones, we went for a different approach. We use our *VortexNode*as an IMAP/SMTP proxy. That way we can send with any email client *VortexMessages*. To do so, we had to introduce an addressing scheme compatible with with email and the support of their clients but not creating any collisions with the existing email address schemes.

These scheme is discussed in the next section. After that, we address the problem of linkking to user agents and transparency issues.

21.1 Addressing and address representations

An endpoint requires always a public key and a transport endpoint. As we have no central infrastructure, we need a defined way to exchange addresses. These adresses need to be uniquely identifiable and have to work with clients. In this section we focus therefore on the implementation details of such an address.

If we want to use common email or XMPP clients we must support an address format which is compatible to the client but produces no collisions with ordinary addresses. Luckily experiments showed that clients are not very restrictive in the acceptance of addresses. Most of the clients required either an at sign between two letters or additionally at least a dot in the domain part of the address. [73] and [123] specify the format for email addresses and [129] does the same for XMPP. For both formats a double dot (“..”) in the local part is illegal. Clients do not seem to catch this exception. We therefore defined our addresses as follows.

For email:

```
localPart         = <local part of address>
domain            = <domain part of address>
email             = localPart "@" domain
keySpec           = <BASE64 encoded AsymmetricKey [DER encoded]>
smtpAlternateSpec = localPart ".." keySpec ".." domain "@localhost"
smtpUrl           = "vortexsmtp://" smtpAlternateSpec]
```

For XMPP:

```
localPart         = <local part of address>
domain            = <domain part of address>
resourcePart      = <resource part of the address>
jid               = localPart "@" domain [ "/" resourcePart ]
keySpec           = <BASE64 encoded AsymmetricKey [DER encoded]>;
jidAlternateSpec  = localPart ".." keySpec ".."
```

```
7 domain "@localhost" [ "/" resourcePart ]
8 jidUrl             = "vortexxmpp://" jidAlternateSpec]
```

This allows using of a regular client to host an *VortexMessage* endpoint address. To avoid unintentional routing of an address to through a non-*VortexNode* we defined “localhost” as the general domain part. The local part in email is restricted to 64 bytes whereas XMPP specifies 1024 bytes as size limit for the local part. Our experiments showed, however, that none of the clients enforce these limit.

The respective URLs are defined in the standard to provide a unified mean for URLs to be properly identified by a system. This allows a unified usage of mechanisms such as QR codes across all platforms.

21.2 Linking to Common User Agents

From an academic perspective the protocol linking as a proxy is easy. Real world implementation showed however many caveats. We will focus on these problems this section and note workarounds where possible.

When combining data on asynchronous message protocols, we have always two possibilities. Either work as transparent proxy for a single view or combine multiple sources. Another option is always creating a local repository with the downside that such a repository may not be shared with other devices.

For all protocols we have to mention that using the *VortexNode*as a transparent proxy is not always doable for two reasons. First we must do a man in the middle (MITM) attack when proxying outbound or inbound connection. If such connections are encrypted this is a problem, due to the breach of the trust chain involved. Solving tis in an enterprise environment is easy as we have the possibility to control the trust store. Working with mobile operating systems such as android or iOS access to trust stores are complex and under some circumstances even prohibited. Another problem is that such MITM attacks are easily detected when employing DANE ([64, 37]) or similar technologies. Within all protocols analyzed certificate based authentication is very uncommon. But such authentication would break if we do a MITM.

When sending mail we can use authenticated SMTP on the client submission ports. This may be realized either as transparent proxy or as store and forward solution with very little downsides. When working as store and forward we have the downside that in case of networking failure the node may delay or lose (in a worst case scenario) the message without the user knowing it as the client successfully sent the message. For this scenario we developed an easy workarround: Our SMTP implementation binds on 127.0.0.1 only and accepts a dummy password As soon as the data sequences simultanously we build a second connection to the providers SMTP channel and authenticate. As soon as the envelope is complete we decide whether the recipient is a*VortexNode*(easily identifiable by the address). If not we send the envelope to the providers SMTP connection and

strictly forward from there on all traffic between the two. If the recipient is a *VortexNode*, we use a pseudo blending layer which packs an apropriate routing block and the plan text message as a single payload into a pseudo *VortexMessage*and deliver this message to the routing layer. The routing layer then, unaware of the pseudo nature of the message, handles the message does the first encryption operation required and applies then the operations to send the message to all next hops.

When receiving messages by mail things get quickly more complex. For our experiments we used POP3 as a protocol. This protocol is somewhat similar to SMTP and allows normal store and forward operation. This means that we may fetch mail from a central infrastructure. This fetching is triggered by the fetching of the client, which is thus almost delay-less. As with POP3 mails are stored locally, we have no problems as the client fetches and stores the mail. Considering IMAPv4 we have a couple of very relevant differences. Unlike POP3 IMAPv4 stores and organizes messages on the server. The mainn advantage is that due to the central storage, multiple devices may access the messages simultaneously. Since all clients use the same storage a unified view is possible. Unfortunately all atempts generating a globally unique id for messages failed so far and client support for such a feature is sparse. In an ideal world we would have a unified view out of one or more *MessageVortex* transport layer accounts and our regular mail, whereas the *VortexMessages* are stored in the respective transport layer account and dynamically merged into the regular email store.

Such a store would have huge benefits compared to the current solution. It would allow a unified storage and offering simultaneous access for multiple devices. The problems to be solved are numerous. We need a unified storage for configurations including eIDs and workspaces. Furthermore, we need a lock to avoid concurrency issues with simulatneously running *VortexNodes*. The unified view requires intelligence so that it is able to keep all *VortexMessages* on the transport layer account whereas ordinary emails are kept on the respective account. The housekeeping of the transport layer account needs to be done in a credible way.

22 Efficiency Related Implementation Details

In the following subsection we focus on the storage management of *VortexNodes*. As they do run on mobile and similar devices, an adequate ressource consumption is essential for our system. We mainly focus on memory and CPU consumptions. Network bandwith overhead and their related problems are discussed in part VII.

22.1 Node Storage Management

In most mobile devices storage is very limited. This applies to the disk storage, but is exceptionally true for the RAM of such devices. Our protocol supports

minimization of storage footpints in two ways.

1. Every node may minimize the storage footprint by signaling that only a small footprint is possible through the capability block.

2. Every node may minimize the number of eIDs accepted.

The runtime portion of RAM required may be minimized as the concurently needed RAM is limited to the event triggered Routing blocks respectively their trigger blocks. Listing 11 defines two type of windows. The absolute time (`AbsoluteUsagePeriod`) denominates the time interval the item is valid in an absolute UTC-based manner. The relative timing (`RelativeUsagePeriod`) furthermore limits the validity window measured relatively to the time of arrival. The real validity time is formed as the intersection out of the two timings whereas both may be ommited by definitions.

```
1  UsagePeriod ::= CHOICE {
2    absolute [2] AbsoluteUsagePeriod OPTIONAL,
3    relative [3] RelativeUsagePeriod OPTIONAL
4  }
5
6  AbsoluteUsagePeriod ::= SEQUENCE {
7    notBefore        [0]     GeneralizedTime OPTIONAL,
8    notAfter         [1]     GeneralizedTime OPTIONAL
9  }
10
11 RelativeUsagePeriod ::= SEQUENCE {
12   notBefore        [0]     INTEGER OPTIONAL,
13   notAfter         [1]     INTEGER OPTIONAL
14 }
```

Listing 11: Definition of a timing trigger

The `ReplyCapability` as shown in listing 12 allows a *VortexNode* to effectively limit the memory usage.

22.1.1 Storage Management of Ephemeral Identities, Operations, and Payload Blocks

The ephemeral identity (eID) is the overarching unit of a users data. In a normal message server it may be comparable with the storage required for the queues. Unlike a message queue, *VortexMessages* are not only kept until sent. *VortexMessages* have different properties as we have a timed store-and-forward behavior. As a general rule, no data lives longer than its eID.

When an eID is requested an absolute `UsagePeriod` (a timezone bound time) is specifies with an `AbsoluteUsagePeriod` is specified (see Figure 11). Unless used for reply blocks (MURBs), eID have a very limited lifespan of a couple hours. This minimizes any storage footprint associated with an eID.

```
ReplyCapability ::= SEQUENCE {
  -- supported ciphers
  cipher            SEQUENCE (SIZE (2..256)) OF CipherSpec,
  -- supported mac algorithms
  mac               SEQUENCE (SIZE (2..256)) OF MacAlgorithm,
  -- supported PRNGs
  prng              SEQUENCE (SIZE (2..256)) OF PRNGType,
  -- maximum number of bytes to be transferred (outgoing bytes in vortex message without blending)
  maxTransferQuota  INTEGER (0..4294967295),
  -- maximum number of messages to process for this identity
  maxMessageQuota   INTEGER (0..4294967295),
  -- maximum simultaneously tracked header serials
  maxHeaderSerials  INTEGER (0..4294967295),
  -- maximum simultaneously valid build operations in workspace
  maxBuildOps       INTEGER (0..4294967295),
  -- maximum payload size
  maxPayloadSize    INTEGER (0..4294967295),
  -- maximum active payloads (without intermediate products)
  maxActivePayloads INTEGER (0..4294967295),
  -- maximum header lifespan in seconds
  maxHeaderLive     INTEGER (0..4294967295),
  -- maximum number of replays accepted,
  maxReplay         INTEGER (0..maxNumberOfReplays),
  -- Supported inbound blending
  supportedBlendingIn SEQUENCE OF BlendingSpec,
  -- Supported outbound blending
  supportedBlendingOut SEQUENCE OF BlendingSpec,
  -- supported galoise fields
  supportedGFSize   SEQUENCE OF (1..maxGF) OF INTEGER (1..maxGF)
}
```

Listing 12: Definition of a capability reply block

```
HeaderBlock ::= SEQUENCE {
  -- Public key of the identity representing this
  -- transmission
  identityKey       AsymmetricKey,
  -- serial identifying this block
  serial            INTEGER (0..maxSerial),
  -- number of times this block may be replayed
  -- (Tuple is identityKey, serial while
  -- UsagePeriod of block)
  maxReplays        INTEGER (0..maxNumOfReplays),
  -- subsequent Blocks are not processed before
  -- valid time.
  -- Host may reject too long retention.
  -- Recomended validity support >=1Mt.
  valid             UsagePeriod,
  -- contains the MAC-Algorithm used for signing
  signAlgorithm     MacAlgorithmSpec,
  -- contains administrative requests such as
  -- quota requests
  requests          SEQUENCE
                    (SIZE (0..maxNumOfRequests))
                    OF HeaderRequest ,
  -- Reply Block for the requests
  requestReplyBlock   RoutingCombo,
  -- padding and identitifier required to solve
  -- the cryptopuzzle
  identifier [12201] PuzzleIdentifier OPTIONAL,
  -- This is for solving crypto puzzles
  proofOfWork[12202] OCTET STRING OPTIONAL
}
```

Listing 13: Definition of a header block

Every header block contains a relative and possibly an absolute `UsagePeriod`. A receiving node calculates a headers' lifespan by intersecting an absolute lifespan and a relative lifespan. All elements of a *VortexMessage* inherit this lifespan. Therefore, payload blocks and operations as well as the routing blocks expire simultaneously within a workspace.

Furthermore, A node signals additional boundaries in the CapabilityReplyBlock (see Figure 12). with this block, a *VortexNode* may limit the storage required even further. By specifying low boundaries for the maximum simultaneously usable payload blocks in a workspace and their maximum size, we can effectively limit the size of the payload data of a single workspace. The number of simultaneously active operations is similarly limited by specifying `maxBuildOps`.

22.1.2 Life-cycle of Requests

Requests have a separate livecycle. As a Request may exist prior to a corresponding workspace, which is typically assigned to a proof of work, such requests may be subject of DoS attacks by flooding the memory of a node. All requests immediately executed have no direct memory requirements. However, requests containing a PoW cycle require to maintain the state.

While this is considered a minor issue as it is very likely that nodes will first collapse due to their network load, we can still address this issue by using a secret generator instead of a list based as outlined in section 19.3.1. By using such a generator, we minimize the impact of a very sudden increase in requests while keeping the local memory requirements to an absolute minimum.

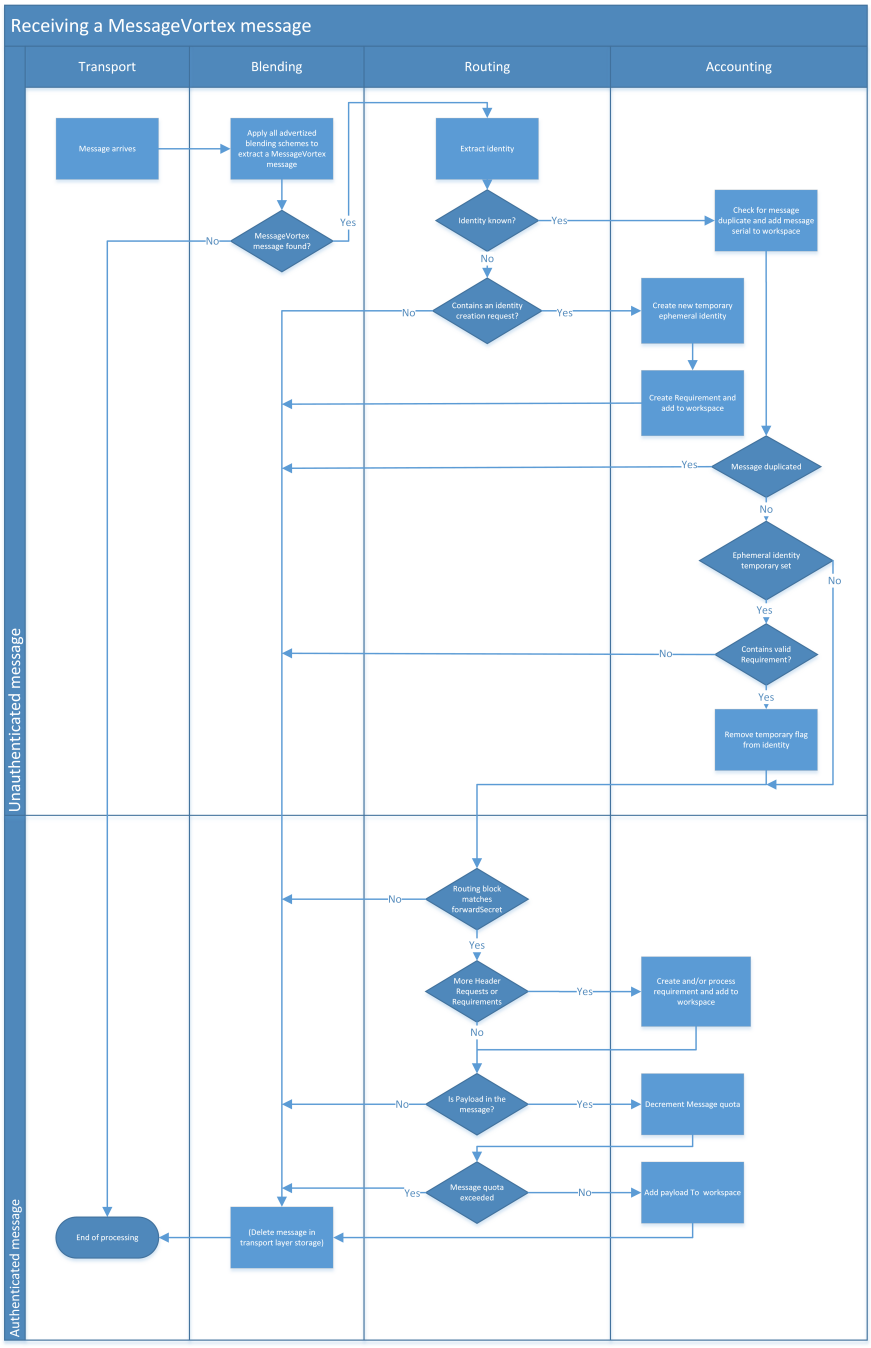


Figure 19.1: flow diagram showing processing of outgoing messages

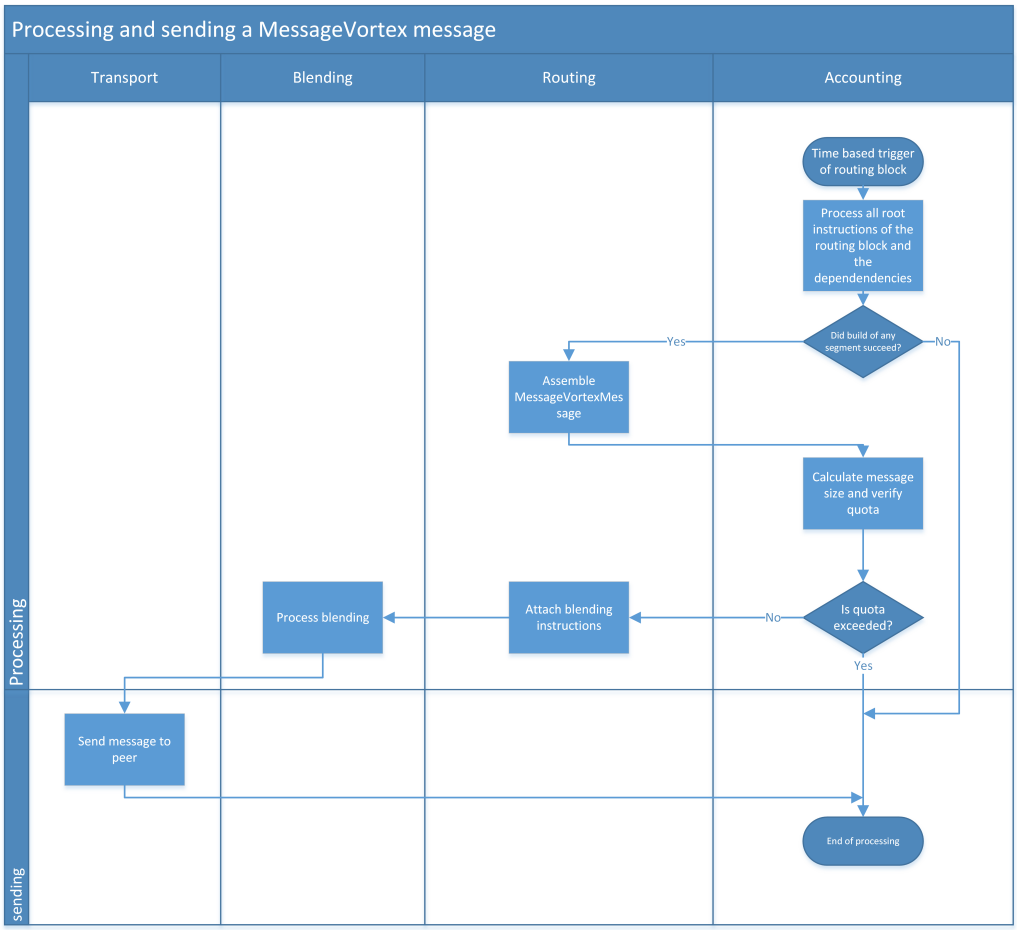


Figure 19.2: flow diagram showing processing of outgoing messages

Part VI

Operational concerns

***

Occurrences in this domain are
beyond the reach of exact
prediction because of the
variety of factors in operation,
not because of any lack of order
in nature.
Albert Einstein

23 General Concerns Regarding Operation

23.1 Hardware

We require no specialized hardware for running Vortex nodes. Instead, we designed Vortex in such a way that ordinary mobile phones may act as Vortex nodes. It is, however, recommended to have a node always connected to the Internet. A mobile phone may disconnect from time to time based on the availability of the network. For our experiments, we used a RaspberryPi Zero W. It is, however, recommended to use a faster, newer model due to the memory requirements of the proof of work algorithm.

The hardware currently requires a network interface and a fully functional JSE VM to run the reference implementation.

23.2 Addressing of Vortex Nodes

From the start, we were looking for an addressing scheme suitable for transparent addressing.

A Vortex address is built as follows:

```
localPart         = <local part of address>
domain            = <domain part of address>
email             = localPart "@" domain
keySpec           = <BASE64 encoded AsymmetricKey [DER encoded]>
smtpAlternateSpec = localPart ".." [ keySpec ] ".." domain "@localhost"
smtpUrl           = "vortexsmtp://" smtpAlternateSpec
```

To allow storage of Vortex addresses in standard messaging programs such as Outlook or Thunderbird, we introduced $smtpAlternateSpec$.

The suffix “@localhost” makes sure that any non-participating server does not route a *VortexMessage* unintentionally. The doubly dotted notation is not RFC compliant but was accepted by all tested client address books. The address is, however, not a valid SMTP address due to its double-dotted notation. We selected this representation to differentiate Vortex addresses from valid email addresses.

The main downside of vortex addresses is that they are no longer readable by a human. The main reason for this is the public key, which is required. We may abstract this further by allowing clear-text requests on the primary email address for the public key. The vortex account must then answer such requests with the valid Vortex address.

The $smtpUrl$ is representing the address in a standard way, which makes it suitable for QR codes and intent filters on Android.

The public key of an address is encoded as follows:

1. The asymmetric key is encoded as specified in the `AsymmetricKey` in ASN.1

2. The ASN.1 DER representation is then encoded using BASE64

The `keySpec` may be omitted and inserted later from an address list. The quad-dotted resulting address is illegal in a standard mail system and offers a possibility for identification. Such a key-less address may furthermore be used as a synonym for the receivers' real address as any potential receiver may send an unsolicited `HeaderRequestReplaceIdentity`.

23.3 Client

We did not create a Vortex client for sending messages. Instead, we used a standard Thunderbird email client pointing to a local SMTP and IMAP Server provided by a Vortex proxy. On the SMTP side, Vortex does encapsulate where possible mails into a Vortex message and builds an automated route to the recipient. The SMTP part of Vortex may be used to encapsulate all messages automatically with a known Vortex identity into a *VortexMessage*. On the IMAP side, it merges a local Vortex message store with the standard Email repository building a combined view.

Using Vortex like this offers us the advantages of a known client with the anonymity Vortex offers.

Using a proxy has certain downsides. At the moment, the vortex client has only a local store. Such a local store makes it impossible to handle multiple simultaneously connected clients to use Vortex. This limitation is, however, just a lack of the current implementation and not of the protocol itself. We may safely use IMAP storage for storing *VortexMessages* centrally. This statement is true as long as:

- The storage is not identifiable as such.
  This requires:
  - A non-identifiable folder/message structure
  - A storage not identifiable by access patterns
  - The stored messages do have the same strength as the transmitted messages in terms of detectability
- A secured key
  Either the host key is secured sufficiently with KDF, and a passphrase (or similar), or the host key remains off-storage.

23.3.1 Vortex Accounts

By definition, any transport layer address may represent a Vortex identity. This fact may make people believe that their current email or jabber address is suitable

as a Vortex address. This statement is technically perfectly true, but should not be done for the following reasons:

- If an address is identified as a Vortex address, it may be blocked (directly or indirectly) by an adversary. Such blocking would lead to blocking of regular email traffic as well.
- If a vortex node is malfunctioning, non-*VortexMessages* should remain unaffected. Isolation is far better if we keep non-Vortex messages in a separate account.
- If a user wants no longer to maintain its Vortex address, he may give up his Vortex transport accounts. If he had been using his normal messaging account for Vortex, he would receive mixing messages which are hard to filter even with a known host key.

23.3.2 Vortex Node Types

Depending on the type of adversary within a jurisdiction, a *VortexNode* may require different properties. In section 13.1, we defined observing and censoring adversaries. In environments with an observing adversary, the presence of a vortex node is not something that we have to keep hidden. In jurisdictions with a censoring adversary, we have to hide our nodes from the censor as their existence may be considered illegal.

23.3.2.1 Public Vortex Node

Public nodes are nodes, which advertise themselves as normal mixes. Just as all nodes, they may be an endpoint or a mix. Typically they accept all requests exactly as outlined in 20.1. As an immediate result of the publicly available information about such a node, the owner may be the target of our censoring adversary. Pressure may be opposed to close down such a node. However, since we do not need a specific account, we may safely close down one transport account and open up a different one. Such account reopenings are even possible on the same infrastructure. We are even able to notify other users of the move and remain reachable, as a user may send a `HeaderRequestIdentity` request using the old identity.

23.3.2.2 Stealth Vortex Node

This node does not answer any clear-text requests. As an immediate result, the node is only usable by other nodes knowing the public key of this node. The node is, therefore, on a known secret base only reachable. This node type may be used in environments with a censoring adversary. People may form closed

routing groups routing and anonymizing themselves. We have to state clearly at this point that putting trust into the routing nodes violates the zero trust principle. It is, however, currently the only way to outcurve a censoring adversary. Means such as using distribution lists as endpoints seemed to be of some value at first but turned out just to shift the problem of detection from the routing to the less protected transport layer.

23.3.2.3 Hidden Vortex Node

A hidden node is a special form to a stealth node. It has a predefined set of identities. Only these already known identities are processed. This behavior has certain drawbacks. An existing identity may not be changed, and new ephemeral identities may not be created. As an immediate result, traffic may become pseudonymity. To counter this effect, at least partially, we may use the same local identity for multiple senders. To remove clashes in the workspace, we may use preassigned IDs in the workspace. The sender is only one of all senders knowing the private key of an identity. The advantage of such a node is that identities have unlimited quotas on such nodes, no longer bothering about accounting and refreshing identities. Such behavior seems to be a valuable option when using bulletproof providers.

24 Routing

Routing contributes heavily to the security of *MessageVortex*. In our system, we typically have one node identity (node key). While this identity is relatively constant (but may be exchanged and notified by a `HeaderRequestReplaceIdentity` request), the involved transport nodes may be more mobile. In general, an incoming transport address does change rather infrequently (unless advertised to friends with the header request mentioned above). The sending endpoint is irrelevant in the routing, and any routing node may, apart from the protocol type, freely choose this endpoint.

While having routing capabilities is absolutely mandatory, as every repeated pattern in routing leads to the possibility of identifying a node of an anonymity system, it adds significantly to the systems' complexity.

The following sections emphasize the operational aspects of the routing. We introduce a detailed pseudo-code for creating a routing block and elaborate on the pros and cons of this implementation regarding complexity and anonymity.

24.1 Strategies for Composing Routing Blocks

We have to follow certain rules when building routing blocks. The rules are:

- Assuming an adversary has partial or full insight into a routing graph (except for the sender and the final recipient), all operations must be valid. This means that no operation may be applied and an inverse operation with different parameters (i.e., $D^{K_a}\left(E^{K_a}\left(\mathbf{X}\right)\right)$).

- No pattern is repeated within the protocol. This applies to:

  – Timing patterns in messages.
  If we define fixed patterns of how a message has to be delivered (e.g., a message has to be delivered within a certain time or a payload block does expire in a workspace within a certain amount of time) and publish these as general rules, we allow an attacker to identify such timing patterns of the net and draw precise lines which messages may possibly be involved in a message transfer. By omitting such definitions and allowing each RBB to define these values to themselves without communicating them, we make it harder to analyze the system by timing patterns.

  – Operation patterns.
  Defining operations use in a fixed pattern (e.g., first distribute a message over five independent message paths sized n), we would give clues to an adversary where in this pattern he is located and how close he is in regards to the beginning or in regards to the end. A difference in the patterns for message traffic and decoy traffic may result in the identification of decoy traffic.

  – Message patterns.
  Always communicating in the same pattern of messages (regardless of the timing), for example, always creating a full communication mesh with all parties of the anonymity set, is an identifiable property that an adversary may use to identify involved *VortexNodes*from the outside.

  – Patterns in size or content of the payloads.
  Sending always similar patterns in size or content allows an inside observer to match similar sized payloads suspecting that they might have a connection and thus breaking the anonymity generated by an intermediate, honest node. Having the same pattern in the content on two different nodes (even as “intermediate result”) is breaking all anonymization steps taken between the two workspaces as two collaborating nodes may identify this content as the same and thus conclude with certainty that they belong to the same message.

  – Applies the same patterns on decoy routes as on message routes.
  When applying different patterns on message and decoy routes, an adversary might notice such different behavior and thus exclude all in decoy traffic involved nodes from the anonymity set.

We may use several strategies depending on our anonymity needs.

Strategies may include:

- Focusing on redundancy of paths.
  In this scenario, we build routing graphs that have a minimum sized set of u independent paths expressed by the involved nodes. Such a routing graph can guarantee that a message will arrive when less than u nodes fail.

- Focusing on involved jurisdictions.
  By focusing on the jurisdiction, an RBB may decrease the likeliness of analysis. As with each jurisdiction involved in the routing o a *VortexMessage*, the likeliness increases that a non-collaborating jurisdiction is involved. By making educated guesses (e.g., that two opposing countries or organizations are unlikely to collaborate), the risk that a path may fully be analyzed from the sending node to the receiving node is less likely.

- Focusing on the speed of delivery.
  The smaller we define the time windows for routing a message from the sender to the final recipient, the simpler is analysis for an adversary as there are fewer messages involved in a possible routing (assuming that an adversary has the means to identify by some magic all *VortexMessages*). Inversely, if the speed of a message may be generally long, an adversary has to take far more messages into account.

- Focusing on the size of the anonymity set.
  The more involved nodes and transport protocols in a routing block, the more complex observation of the protocol is. By increasing the anonymity set, the likelihood of overlapping routing graphs increases significantly. Furthermore, the normal Message traffic of the transport protocol may further increase the complexity of an outside observer.

- Focusing on anonymity of the eIDs.
  By using only short term eIDs where ever possible, we increase the complexity for an adversary as we reduce the number of overlapping routing points for the same identity. While the original sending identity may remain the same, the changing eIDs make it impossible to identify anonymity groups over time.

- Focusing on the distribution of the message parts.
  A sender applying an $addRedundancy(m,n)$ operation on a message before sending is safe unless $n-m$ node in independent message paths collaborate and have full knowledge of all keys and operations (including the ones applied on the senders' node) as the resulting equation system would have any possible solution (in length and appearance) up to the size of all $n-m$ blocks.

- Focusing on diagnose-ability.
  By deploying diagnosis payload blocks on subsequent nodes instead of just leaving them in the workspace of a node, the possibility of falsifying the result of a diagnosis based on the assumption that first the message is delivered and diagnosis is made retrospectively when detecting a problem is eradicated.

The algorithm itself does not really matter as long as it guarantees the properties at the beginning of this section.

24.2 Strategies for Minimizing Impact and Maximizing Effect when Routing Foreign Messages

Keeping a single node alive can be important. If we are assuming that reception of a message and sending is done through the same transport account, it is relatively easy for an adversary to observe this. By sending to a recipient transport address, he learns that a *VortexNode* is connected to that address. Conversely, any mail coming from such an address is potentially a *VortexMessage*.

Any node may reduce the traceability by following a couple of additional rules. First of all, transport addresses for sending should be kept separate from receiving transport addresses. By doing so, an adversary needs to carry out man-in-the-middle (MitM) attacks in the respective access protocols or get direct access to the transport infrastructure to learn what transport addresses are used by the *VortexNode*. If NAT is involved in the client access, as it is the normal case when using the targeted infrastructure for a *VortexNode*, it just adds to the complexity an adversary has to solve. While this is no true gain in anonymity, it contributes heavily to the complexity an adversary has to handle. In a more advanced scenario, we would use some anonymization technology such as ToR to further hide the accessing source (*VortexNode*) from the transport infrastructure. However, the use of such technology will make access to suspicious and possibly lead to the identification of the transport account.

A supposedly compromised transport layer recipient endpoint address may be migrated using a `HeaderRequestReplaceIdentity` request as outlined in section 19.3.2. Such a request leaves no trace to the transport endpoint owner but allows any subset of known *VortexNode* to advertise the migration in a cryptographically secured way. Additionally, this request allows by omitting the new address to bind an ephemeral identity to a true transport address identifying the sender of a message. Such an ephemeral identity may be assigned with an infinite quota by the owner to spare the costs of recreating and re-authenticating the sender. If such binding of identity is done, it is absolutely vital that this identity is not used for routing but only as an endpoint. Otherwise, a malicious “friend” could draw conclusions on routing anonymity set and frequency out of such an identity.

24.2.1 Operational Aspects of MURBs

As we have interactions of any possible node with an unknown sender of a request (e.g., in the case of a new identity request), reply blocks are a necessity for the *MessageVortex* protocol.

Originally, we included the possibility of replaying replayable blocks (MURBs) for

sending error messages. Soon we figure out that such messages imply privacy issues. While the error messages have been dumped in favor of an RBB based diagnosability, we kept the possibility of MURBs to enable users to have sender/recipient anonymity.

Our MURBs are routing blocks that an owner of the block may use for a limited amount of times. Such sending may be done without having any knowledge about the recipient's identity, location, or infrastructure. A MURB is equivalent to a normal routing block except for the following properties:

- The sender is not known but the receiver of the message.
- It has a replay value of 1 or higher.
- Due to transport layer size restrictions and ephemeral quotas, the total size of the transported messages is limited.

A MURB in our term is an entirely prepared routing instruction built by the recipient of a message. The sender has only the routing blocks and the instructions to assemble the initial message. It does not know the message path except for the first message hop.

As a MURB is a routing block, it generates the same pattern on the network each time a sender uses it. To avoid statistical visibility, we need to limit the number of uses per MURB. As a maximum number of usages, the protocol is limited to 127 usages. This number should be sufficiently sized for automated messages. A minute pattern would disappear after 2 hours the latest and an hourly pattern after five days.

For a MURB to work, the RBB has to take care that all quotas required to the route are sufficiently sized. Such sizing is hard to foresee in some cases. An RBB may query these identities from time to time to make sure that they do not deplete. Wherever possible, MURBs should be dropped in favor of multiple SURBs to avoid the dangers of MURBs.

24.3 Routing Algorithms Suitable for Achieving Anonymity

In section 24.3, we elaborated on the properties of a routing block required to build an anonymizing message path.

In short, every foreseeable or logically invalid pattern may be used to identify *VortexMessages* or in transport involved nodes. This is why we cannot use a fixed pattern in routing. Instead, we use randomized routing patterns. Ordinary fixed pattern protocols, such as broadcast or DC-net based protocols, are identifiable as their communication pattern is stable (fixed set of messages between involved

nodes and foreseeable message size). Whereas the message size might be varied in such systems by adding decoy content or stuffing, such behavior depends on the secrecy of the nodes executing such operations.

24.3.1 The Routing Graph

In general, an RBB builds a routing block in three stages:

1. Create a directed multigraph (routing graph) where the nodes represent *VortexNodes*, and the edges represent the messages sent between the *VortexNodes*. The graph may contain loops. We may visualize such a routing graph traditionally. Alternatively, we found that displaying the graph as a sequence of messages (see figure 24.1) offers a better overview over the inner workings of a routing graph.
2. We then assign timing information to each edge, leaving sufficient time in between to process the incoming message on the transport layer.
3. As next step, we assign operations to all involved workspaces.

Within such a routing graph, we refer to a path between the two nodes i and j as an ordered set of edges, where an edge always starts where the previously edge ended, the first edge starts at node i, and the last edge ends at node j. A path may contain multiple times the same node, and a routing graph may contain multiple paths between two given nodes. Figure 24.2 shows all paths between nodes 0 and 1 of the graph outlined in figure 24.1. All these paths may be used to transport a message from node 0 to 1. Depending on the strategy multiple paths may be used to transport a part of a message or used as a redundancy path.

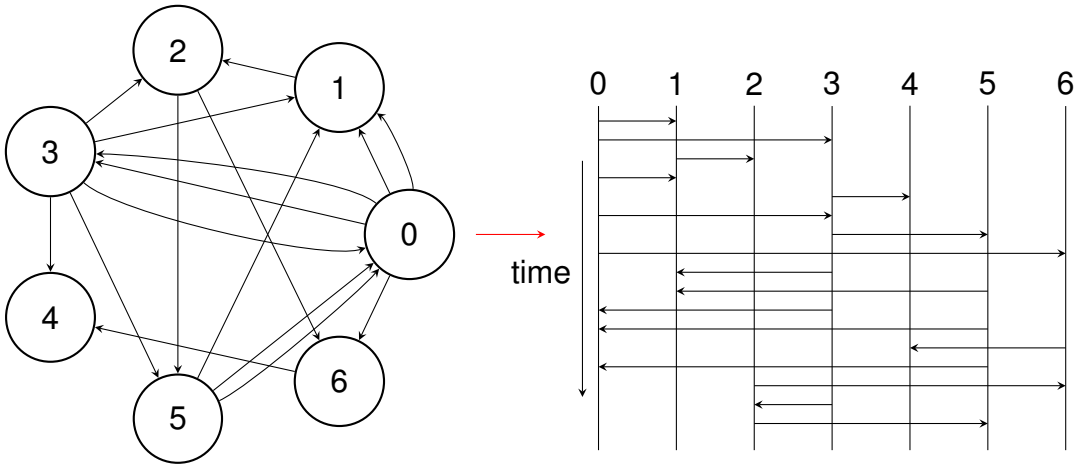


Figure 24.1: transformation of a graph into a sequence of messages

A possible routing mechanism creating such a graph and aplying routing information is described in detail in section 24.3.2.

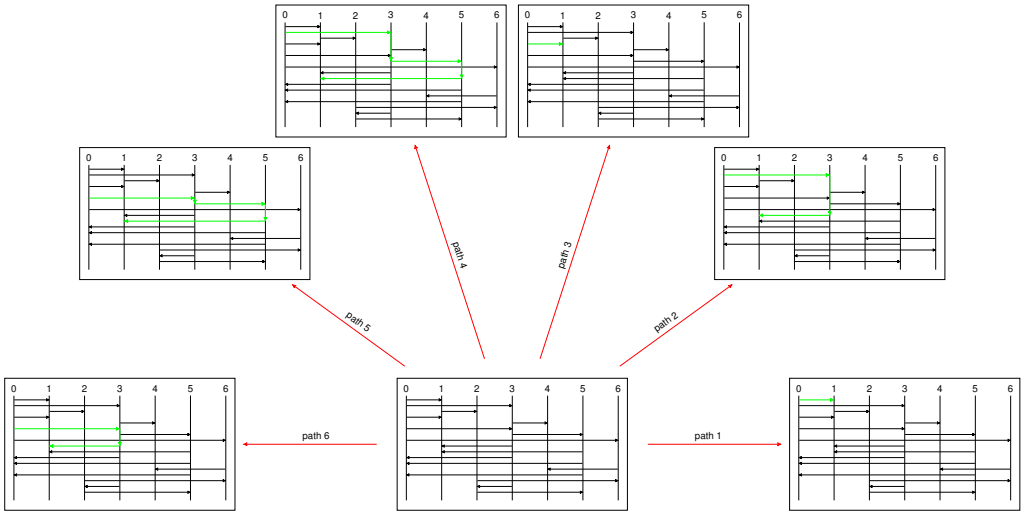


Figure 24.2: A graph containing six paths between node 0 and node 1

24.3.2 A Simple Routing Strategy

In this section, we show a simple algorithm for creating a routing graph in a non-censored environment or in an isolated node-set in a censored environment. While the algorithm is complete, we had to shorten it for this work in order to remain readable. The algorithm is not perfect as it leaks certain properties, such as the maximum possible message size.

To create a routing block, we need some basic objects as defined in algorithm 1.

Algorithm 1 Objects for building a routing block

```
▷ A routing graph
object RoutingGraph
    nodes : List<Node>
    messages : List<Message>
end object

object Message
    sourceNode : Node
    sourceId : int
    earliestTime : datetime
    latestTime : datetime
    targetNode : Node
    targetId
    operations : List<Operations>
end object

▷ The projected workspace of any eID under our control
object WorkSpace
    payloads : Map<Id,Payload>
    routingBlocks : List<RoutingBlock>
    operations : List<Operation>

    ▷ Returns an unused id with at least <numberOfSubsequentIds> unused IDs following
    abstract function getUnusedId(numberOfSubsequentIds) {...};
    ▷ Returns a random output id of an operation unused so far and marks it as used
    abstract function getRandomPayloadId() {...};
end object

▷ An object reflecting our knowledge about MessageVortex
object Universe
```

```
27:     knownNodes : MAP<NODE,WORKSPACE>

28:     ▹ Returns all the nodes of knownNodes
29:     abstract function GETALLNODES() {...};
30:     ▹ Returns a random node of knownNodes
31:     abstract function GETRANDOMNODE(list) {...};
32:     ▹ Returns the representation of the workspace of the named node
33:     abstract function GETWORKSPACE(node) {...};
34:     ▹ Adds a message to a workspace with all its content (payloads, operations)
35:     abstract function ADDMESSAGETOWORKSPACES(message) {...};
36: end object
```

To create a routing block, we first need a graph representing the message flow. Algorithm 2 shows a pseudo-code to get such a valid graph. After creating a graph, we need to assign timing and routing information. Algorithm 3 shows a possible algorithm for assigning this timing information, whereas Algorithm 4 shows a simple generator for the routing operation. The algorithm omits for simplicity allocation of workspace IDs as this is a "bookkeeping"-only problem.

To create a graph we use the function GETROUTINGGRAPH on line 1 as shown in algorithm 2. It creates an ordered set of nodes (`nodes`), whereas the first node in the set is the sender and the second node of the set is the final recipient. It then adds randomly known nodes until the anonymity set is as large as requested. Next, we assign the edges by calling function GETEDGES (Line 17). The function loops until the requested minimum number of edges are reached, and all nodes of the graph receive at least one message. On each loop, an edge is added to the graph, that is pointing from any already reached node to a random, different node.

Algorithm 2 Simple Graph for Routing Block

```
 1: function GETROUTINGGRAPH(startNode,endNode, numNodes, minEdges, universe)
 2:     ret ← new RoutingGraph()
 3:     ret.nodes ← GETNODES(startNode, endnode, numNodes, universe)
 4:     ret.messages ← GETEDGES(minEdges, ret.nodes, universe)
 5:     return ret
 6: end function

 7: function GETNODES(startNode, endnode, numberOfNodes, universe)
 8:     nodeList ← [startNode, endNode]
 9:     while len(nodeList) < numberOfNodes do
10:         randomNode ← universe.GETRANDOMNODE()()
11:         if ¬nodeList.contains(randomNode) then
12:             nodeList.append(randomNode)
13:         end if
14:     end while
15:     return nodeList
16: end function

17: function GETEDGES(minEdges, nodes, universe)
18:     edgeList ← []
19:     while len(edgeList)<minEdges or
            getReachedNodes(edgeList, listOfAllNodes[0])<len(nodes) do
20:         listOfReachedNodes ← GETREACHEDNODES(nodes, edgeList)
21:         startNode ← UNIVERSE.GETRANDOMNODE(listOfReachedNodes)
22:         endNode ← UNIVERSE.GETRANDOMNODE(listOfAllNodes − [startNode])
23:         edgeList.append(new Message(startNode, endNode))
24:     end while
25:     return edgeList
26: end function
```

```
27: function GETREACHEDNODES(edgeList,startNode)
28:     reachedNodeList ← [startNode]
29:     for all e ∈ edgeList do
30:         if ¬reachedNodeList.contains(e.targetNode) then
31:             reachedNodeList.append(e.targetNode)
32:         end if
33:     end for
34:     return reachedNodeList
35: end function
```

In algorithm 3, we assign the timing information.

We use a custom random distribution called GETRANDOMTIME(line 21). This distribution is a derived form of a Gaussian distribution and has its minimum value, maximum value, and peak value at desired spots. The squishing of the function violates some properties of the Gaussian bell curve. Due to the squishing, the left and right sides of the bell no longer have the same area. The timing information distributes in a serialized way along the timeline. Figure 24.3 shows the distribution of the implementation.

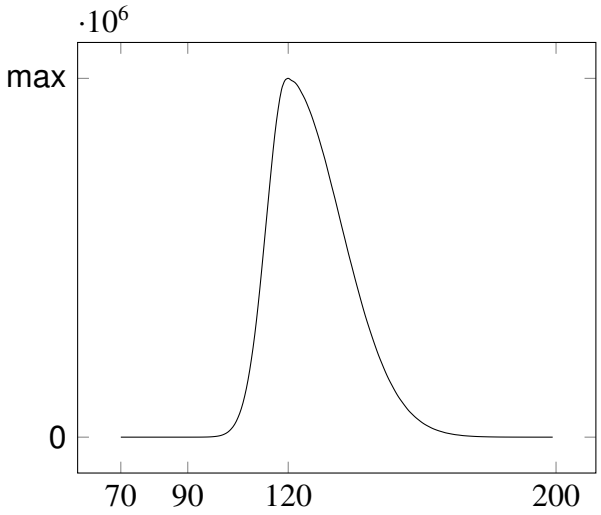


Figure 24.3: Distribution of GETRANDOMTIME(90, 120, 200) in algorithm 3

We assign the timing information by looping through our ordered set of edges. We first, calculate the earliest (earliestTime) and the maximum available time starting then (maxShare) until the message has to be sent. We calculate when the message has to be sent in relation to earliestTime (share). Finally, we generate a time when an edge may be executed earliest (minTime; line 12) and latest (maxTime; line 13).

Algorithm 3 Assign Timing Information to a Graph

```
1: function GETTIMING(edges, maxTime,minHopTime)
2:     if len(edges) × (minHopTime - 1) > maxTime then
3:         throw "maxTime too small for constraints"
4:     end if
5:     earliestTime ← 0
```

```
6:      maxRemainingTime ← maxTime − earliestTime
7:      remainingHops ← len(edges) − 1
8:      times ← []
9:      for all e ∈ edges do
10:         maxShare ← remainingTime − remainingHops × minHopTime
11:         share ← maxShare / remainingHops
12:         minTime ← GETRANDOMTIME(earliestTime, earliestTime + share, earliestTime + maxShare)
13:         maxTime ← GETRANDOMTIME(minTime, minTime + share, earliestTime + maxShare)
14:         earliestTime ← maxTime + minHopTime
15:         remainingHops ← remainingHops − 1
16:         maxRemainingTime ← maxTime − earliestTime
17:         times.append(minTime, maxTime)
18:     end for
19:     return times
20: end function

21: function GETRANDOMTIME(min, peak, max)
22:     value ← −1
23:     while value < min or value > max do
24:         value ← NEXTRADNOMGAUSSIAN( )
25:         d ← NEXTDOUBLE( )
26:         if d < (peak − min)/(max − min) then
27:             value ← peak − abs(value)×(peak−min) / 5
28:         else
29:             value ← peak + abs(value)×(max−peak) / 5
30:         end if
31:     end while
32:     return value
33: end function
```

Key of the graph itself are not the edges or nodes nor the timing but the operations applied to the graph. This part is covered by function GETROUTING in algorithm 4. We assign the operations in three steps. We first assign to redundantRoutes routes a valid message path (lines 5-15). After that we identify “unused (sub-)routes” and assign the same operations to these routes (lines 17-19).

Operations are assigned in a recursive manner. First, we identify the routes we want to assign operations. This recursive part is done by the ASSIGNROUTE(line 22-31). We first identify a payload to be transported and the chain of nodes. We call ASSIGNROUTE, which will then apply a random operation on the first node and transport the relevant payload block to the second node in the chain, mapping it there to an unused ID within the workspace. We then take the remaining path with the newly created ID in the remaining path and repeat the step, thus looping recursively through the path until we have covered the whole path.

Operations are chosen in two ways either we create an $addRedundancy$ operation of type $n-1$ of n, or we use a simple encryption step. In each case, we apply on the current node the operation, and we apply on the final node the reverse operation, thus rebuilding the message on the last node simultaneously.

Algorithm 4 Assign Routing Information to a Graph

```
1: function GETROUTING(edges, redundantRoutes, messageId)
2:     if redundantRoutes < 1 then
3:         throw "At least one route is required"
4:     end if
5:     routes ← getRoutes(edges)
```

```
    if len(routes) < redundantRoutes then
        throw "Graph has not enough redundant routes"
    end if
    ▷ Add operations to true routes
    numRoute ← 0
    while redundantRoutes > numRoute do
        currentRoute ← routes[numRoute]
        ASSIGNROUTE(currentRoute, payloadId, currentRoute[LAST], 0)
        numRoute ← numRoute + 1
    end while
    ▷ Add sensible operations to decoy routes
    for all r ∈ getUnsuedRoutes(edges) do
        ASSIGNROUTE(r, r.getRandopOperation().getUnusedIds(1), NULL, 32769)
    end for
    ADDMESSAGEMAPPING(edges)
end function

function ASSIGNROUTE(route, payloadIds, lastNode, targetIds)
    source ← route.getSourceNode()
    if payloadIds.isEmpty() then
        PayloadIds ← source.getRandopOperation().getUnusedIds(1)
        payloadSet ← ASSIGNROUTE(route[2-], targetIds.forward(), lastNode, targetIds.reverse())
    else
        targetIds ← ASSIGNOPERATION(route.getSourceNode(), payloadIds, lastNode, targetIds)
        payloadSet ← ASSIGNROUTE(route[2-], targetIds.forward(), lastNode, targetIds.reverse())
    end if
end function

function ASSIGNOPERATION(node, transportIds, reverseNode, targetIds)
    out ← node.outEdges()
    in ← node.inEdges()
    if out > 1 or extRandomInt(3) = 1 then
        ▷ assign addRedundancy
        numBlocks ← max(out+1, nextRandomInt(out+4))
        seed ← nextRandomInt(2^256 − 1)
        op ← node.addRedundancy(transportIds, numBlocks - 1, numBlocks, seed)
        if reverseNode! = NULL then
            reverseOp ← reverseNode.removeRedundancy(targetIds, op)
            newId ← op.getUnusedIds(1)
            newId.addReverseIds(reverseOp)
        end if
    else
        ▷ assign encrypt
        keySize ← (nextRandomInt(3) + 2) ∗ 64
        key ← nextRandomInt(2^keySize)
        op ← node.encrypt(transportIds, "AES", keySize, key)
        if reverseNode! = NULL then
            reverseOp ← reverseNode.decrypt(targetIds, op)
            newId ← op.getUnusedIds(1)
            newId.addReverseIds(reverseOp)
        end if
    end if
    return newIds
end function
```

The algorithm outlined in this section has a couple of downsides due to its brevity. As splitting of routes is hard to create in such a compact recursive manner, we omitted it. And for the same reason, we always used *addRedundancy* operations, which rebuilds the message out of a single block. These simplifications have some drawbacks. This algorithm never loses size (it may gain size due to padding and stuffing). Therefore, we may match similarly sized payload blocks as potentially belonging to the same message. Apart from that, the algorithm fulfills all criteria mentioned above. We apply the same operations on the decoy and true

message traffic, and we have no timing, operations, or message patterns. As soon as this algorithm is using either a split with the $split$ or $addRedundancy$ operation, this weakness disappears too.

24.4 Routing Diagnosis and Reputation Building

When all nodes are working as expected, no diagnostic is required. As we are relying on always-connected devices such as mobile phones as routers, it is likely that not all nodes are available within the required time frames. As a result, we need at least the possibility to identify malfunctioning nodes and exclude them from routing. Furthermore, active adversaries may intentionally induce bad packets to destroy message content.

MessageVortex allows a diagnosis to identify such malicious nodes. We differentiate between implicit and explicit diagnosis. When making an implicit diagnosis, we are analyzing packets that are routed from the start node over one or more other nodes back to the start nodes again. As a routing block builder knows the message content and all involved routing operations, he may calculate the payload spaces at all points throughout the message transfer and, therefore, predict the content and size of the payload blocks received. This is possible due to the fact that we defined all operations byte precise and left no room for interpretation. This applies to all parts of the operation, including padding and stuffing. If the received payload blocks differ from the expectation, at least one of the nodes involved in the transfer of the payload malfunctioned. Reputation building over time can be done by assigning to all nodes additively a small reputation value if involved working route and subtract a value if participating in a loop that malfunctioned. As malfunctioning nodes will always be in a malfunctioning loop, their reputation value will drop while working nodes will build up a score each time when participating with other working nodes.

We describe the Reputation of a node a as R_a. Node a takes part of a set closed loops I with elements I_i. The weighting w_i of a loop I_i is 1 for a successful loop and -1 for an unsuccessful loop. We then may calculate the reputation R_a as described in equation 24.1

$$R_a = \sum_i \frac{w_i}{len\,(I_i)} \tag{24.1}$$

We can make an explicit diagnosis in the case where the payload received does not match its expected value or is completely missing. We may do this by creating additional routing blocks picking up packets of the previous message in the workspaces of the suspected malfunctioning nodes. Explicit diagnosis yields a big danger. An adversary expecting diagnosis because he knows that he has cheated may fall back to an irregular behavior where the first operations are falsified, and if a second routing block arrives, the expected answers are given. This would falsify the reputation score in favor of an adversary and lower the reputation

score of any subsequent nodes. This is why we recommend not to use explicit diagnosis to identify active adversaries or calculate a reputation but only to identify nodes that are offline.

24.5 Redundancy and Distribution Strategy

The capability to distribute data and redundancy information over several nodes is one of the key features of the protocol. The *addRedundancy* operation has two purposes here. First, it allows a splitting operation where the content is not only split but distributed over all parts. While a normal *splitPayload* operation leaves the message itself intact but splits it into two parts, which may contain each meaningful, readable parts of the underlying message, *addRedundancy* distributes the message over the output blocks. The difference is not as big as it seems as the input is (with a possible exception to the sending node) not applied to the original message but to an encrypted part of the message.

Assuming that an attacker does not control the whole network of relevant messages but is in possession of the whole routing block and possesses all operations and keys to recover the original message, It is safe to say that distributing the message over multiple redundant paths is improving security. Both operations allow such behavior, but in a very different way. The operations *splitPayload* and *mergePayload* allow to create payload blocks with any size. However, when transmitting both sizes of a split, they add up to a full block size of the previously done encryption operation. So if we control both receiving nodes of the parts of the *splitPayload* operation, we may conclude that the two eIDs belong to the same real identity. This is why we always used a subsequent encryption operation after applying a *splitPayload*. This rounds both chunks again to block sizes of the encryption operation.

25 Protocol Bootstrapping

Protocol bootstrapping is especially hard in an environment with a censoring adversary. While in an environment of an observing adversary, the nodes may be public and thus queried. In an environment of a censoring adversary, any directory or possibility to query nodes inevitably leads to a possibility of harvesting *VortexNodes*.

We consider the bootstrapping problem as one of the major, unsolved problems of *MessageVortex*.

25.1 Key Distribution for Endpoints

For endpoints, we may have at least a partial solution. Sending a *VortexMessage* as an unencrypted message to the true email of a user, containing a request capability block and a `HeaderRequestReplaceIdentity` without a new $NodeSpec$ may be used to initiate a handshake between two nodes. While such behavior is cryptographically secured, the observing adversary gains as additional information that the receiving party of the message is using *MessageVortex* and learns the full address, including its key from the sending party. None of this information is confidential in an environment with an observing adversary but shows the weakness of bootstrapping the system.

25.2 Key Acquisition for Routing Nodes

Key acquisition of routing nodes in an environment of an observing adversary may be made through the $HeaderRequestNodes$ request. All these nodes distributed by such mechanisms are so-called public nodes and must be considered as untrustworthy nodes at any time.

It is interesting to have an inbound address listed as a public node due to the traffic they receive. At the same time, they are not suitable as nodes for doing communications with environments connected to a censoring adversary. Therefore, such nodes do typically not considered to increase the anonymity set. This due to the fact that such an adversary would most likely try to harvest all public nodes and blacklist them to block cross border traffic and possibly gain clues on the identity of transport endpoints of *VortexNodes* within his reach.

So, while a node in an environment with an observing adversary may use such public nodes, a *VortexNode* within reach of a censoring adversary has two choices:

- Build a trusted “own” network of trustworthy partners and exchanging keys initially by hand.
- Exit the jurisdiction on the first hop or even by using a transport layer account supposedly outside the reach of the own censoring adversary

Both options are equally bad, but the second option is easier to fulfill as currently alliances in terms of cooperations seem to be relatively stable, and only a limited amount of adversaries (e.g., “Five Eyes” or China) have the resources to record encrypted traffic for suspected later decryption.

26 Real World Problems when using *MessageVortex*

Some problems are not directly related to the *MessageVortex* protocol but must still be considered when implementing or using *MessageVortex*. The problems discovered during our experiments and possible solutions are listed in the following sections.

26.1 Size Restrictions of the Transport Layer

A transport layer may limit the size of messages transferred. We managed to create *VortexMessages* as small as $2KB$ in size. Considering the blending overhead of F5, our message is sized at least $16KB$, which is not a problem for any of the selected transport protocol. So, while a *VortexMessage* may be small, an upper size limit is possibly imposed by the transport layer. Most SMTP providers define an upper limit of $10MB$ per message. Taking into account that we use a binary transfer, which is typically BASE64 encoded, the usable transfer size is roughly $7.5MB$, as BASE64 adds roughly 25% overhead. Considering that we should not use any content bigger than 12% of the carrier message, the true transport capability of a $10MB$ message drops to $\approx 900KB$, which is disastrously small. While a single *VortexMessage*may not be bigger than the $900KB$ limit on SMTP due to this limitation, assembly in a workspace does, however, allow to transport bigger messages than the limit on the transport layer.

The size of this calculation shows the waste of the transport capacity of our system in a drastic way. If assuming that we use a high anonymity set of $k = 30$nodes and assuming that on average, each message contains half of the original message and we are exchanging 60 messages within the anonymity set, a $900KB$ message would result in $60 \times 5MB = 300MB$ cumulated transfer volume between all nodes which results in a total transfer efficiency of $\approx 0.3\%$. While such waste is not uncommon within anonymity systems (unless tuned for efficiency), it dramatically shows the level of waste.

26.2 Redundancy of the *VortexNode*

At the beginning of our work, we tried to make *VortexNodes*redundant by sharing configuration and state data over the transport media. While the idea was absolutely tempting, we discovered that any kind of such usage leads to an uncommon usage pattern of the transport account. This uncommon usage pattern allows an adversary to identify transport accounts of *VortexNodes*. We, therefore, dropped this idea.

Part VII

Analysis of MessageVortex

***

Atoms are very special: they like certain particular partners, certain particular directions, and so on. It is the job of physics to analyze why each one wants what it wants.

Richard P. Feynman

In section 13.1, we described two different kinds of adversaries. Which do require different properties for our system to be fulfilled.

An observing adversary is the less restricting one. While this adversary is observing all traffic, he is not disrupting communication. Instead, he is using all available information to collect data about all items of interest (IoI). He may do this for example by collecting inside or outside information about all message flows he may encounter. He may use this information and assign it to certain individuals or groups of individuals.

A censoring attacker is far more dangerous to our system as he does not only observe the system, but he may systematically suppress freedom of speech and all technology related to it. As he has the means and the technical know-how, he may try, apart from observing, to discover systems communicating illegally either by observation or by infiltration of systems. He may furthermore track down individuals within its reach and prosecute them. All other participants of an illegal system may be either identified and blacklisted or even attacked either by infiltrating their systems or by effectively launching DoS attacks against those systems.

In the following sections, we will analyze aspects of confidentiality, integrity,and availability for our system and highlight differences in terms of the different adversaries.

27 Identification of Possible Attack Schemes and Mitigation

In this chapter we take the attacks identified in the section 29.1 and analyze our protocol on whether it is susceptible on such attacks or not.

27.1 Static Attacks

Static attacks typically address weaknesses within a protocol design. The following attacks are typically used to attack protocols similar to our proposal.

27.1.1 Bugging and Tagging Attacks

Bugging and tagging attacks are similar in terms that both try to follow a message to its final recipient. While the goal is similar, the approaches are completely different.

We refer to a bugging attack as an attack, which discloses the recipient by forcing him to do a disclosing action. Such an action may be the lookup of an unusual DNS reccord, lookup or verification of some identifiable data (e.g., an OCSP request to verify a certificate), or downloading an external image induced by an attacker.

A tagging attack allows to follow an attribute of a message through a network and thus uncover members of a social network or even a final recipient.

27.1.2 Information Leaking Related to Information Available to Routing Nodes

Routing nodes are a vital part of any anonymity network. The easiest assumption is to trust nodes. In our case we explicitly distrust routing nodes. This means that we have to minimize the footprint of available information to such routing nodes.

27.1.3 Identification of involved *VortexNodes* or *MessageVortex* Traffic

In an environment of a censoring adversary undetectability of a *VortexNode* is crucial, as any detectability may lead in a shutdown or even repression. We will look in section 28.1 how we can identify involved messages and nodes.

27.2 Dynamic Attacks

Dynamic attacks usually involves an active adversary injecting malicious traffic. They are quite often paired with statistical approaches to discover properties of the system otherwise not available to an observer. In the following sections

27.2.1 Attacks against the *MessageVortex* system itself

An active adversary may attack the transport layer. Most of the transport layers are not able to react upon message flooding. Therefore, it is easy to attack a transport layer with a flooding attack, such as a distributed denial of service (DDoS) attack. Due to the nature of the protocol, we are unable to create additional protection on the transport layer as such modification would require a modification of the transport layer. We will analyze in section **??** the impact on the *MessageVortex* system.

We have identified the following attacks relevant for our system:

- DoS Attacks against the Transport System
- DoS by Traffic Replay
- DoS by Traffic Generation
- Attacking a single ephemeral Identity of a MessageVortex Node

- Denial of Service by Exhausting Quotas or Limits
- Attacking Sending and Receiving Identities of the MessageVortex System
- Traffic Highlighting or Traffic Analysis
- Recovery of Previously Carried Out Operations

The Vortex Message format itself is, however, crafted in such a way that only minimal effort is sufficient to get the involved parties of a transmission. The Operations $K_{msgN} = D^{K^1_{host}}(P)$ and $HEADER = D^{K_{msgN}}(H)$ are sufficient to identify message senders. Unknown Senders may be discarded without further processing. Known senders may be identified as legitimate and processed further. Known misbehaving identities and message duplicates may be discarded.

An active adversary may not follow the protocol and modify any parts of the message. The following paragraphs reflect different kinds of behavior and how they affect the messages and the system as a whole.

An adversary may not follow the blending specification. If he uses a less secure specification, an independent third party observer may follow traffic. This is not sensible as such a node may send all the knowledge to such a collaborating node directly. In the case of a target node not supporting the chosen blending method, the partial message path becomes interrupted. A possible redundancy in the path may recover the message from such a case.

Traffic replay is a common way to highlight traffic in many systems by replaying the same traffic and increase the signal to noise ratio of a system. In our case, we can use the replay of a VortexMessage block to increase the traffic to a node. After decoding the header, a MessageVortex node identifies the block as a repeated block and rejects further processing.

An adversary may replay blocks with varying content. This will not result in a DoS attack as the quota is not decreased on replayed messages (see Figure 19.1).

An adversary may first collect identities and quotas and use them later in a coordinated attack to force the node processing. The adversary may increase the impact by using large payloads and processing them in a costly manner. A possibility is to make extensive use of $addRedundancy$ or encryption operations. Furthermore, an attacker may attack the memory by distributing the message throughout the workspace to exhaust the routers' runtime memory.

As a router is free to process the operations of identity, he may discard an ephemeral identity and all associated resources at any time. Misbehaving or suspected misbehaving nodes may, therefore, be stopped. On the other hand, we are unable to prevent an adversary from allocating new identities. We may, however, work with multiple local host keys and distribute them according to the trust. A known party or someone trusted by them might receive a key different from a publicly advertised key. This identity key may be dropped at any time and

distributed to further parties again with an identity update. We may even subdivide trusted parties into several groups by updating them with different new host keys to identify misbehaving routers without knowing them.

28 Static Analysis

28.1 Analysis of the Blending and Transport Layer

The blending layer is one of the key factors when it comes to confidentiality in an environment where affected by a censoring adversary. We refer to confidentiality of the presence a *VortexNode* as detectability. Detectability of messages and systems in consequence leads to the ability of censorship. We assume that general censorshop on the transport layer (e.g., by blocking all SMTP traffic) is not an option.

In an environment of an observing adversary, confidentiality in regards to the presence of messages is not required, as we defined in those environments to be legal to use *MessageVortex*. In such environments, plain embedding may be used at any time.

28.1.1 Identifying a Vortex Message Endpoint

Depending on the blending method, a single, identifiable message is sufficient to identify a *VortexNode*. Detectability depends on various factors, such as:

- Broken internal file structure (due to plain blending)
- Uncommon high entropy in a structureless file
- Unrelated message flow (see [68])
- Non-human behaviour on the transport layer (e.g., message traffic 24x7)

If an endpoint is successfully identified, then all peering endpoints of the same protocol may be identified as well by following the message flow. This does, however, not enable an adversary to inject messages as the host key is not leaked.

Assuming a global observer as an adversary and unencrypted traffic, he might discover the originating routing layer and thus identify it as Vortex node by following traces of the transport layer. In most protocols, however, this address is spoofable and not a reliable source for the originating account.

As we specified machine communication for our messages the dead parrot problem[68] is not an issue as it only follows human communication. Thus, our system does not have to pass a touring test. Having messages sent with a non-human behavioral pattern (e.g., 24x7) is therefore not an issue either, as well as sending unrelated messages to an unstable set of endpoints.

28.1.2 Analysis of the F5 embedding Method

A routing node must embed the *VortexMessage*into a generated image. Sending the same image multiple times without any generated content will look very suspicious as the same image sent multiple times but with a different fingerprint is not normal behavior. While we may adopt message sending from open source products, it is not perfect as anyone may know what types of message are affected. In return this means that any message not heavily customized is suspicious. To make things worse, modifying the text may be relatively easy while modifying the content of the generated imagery is much harder.

From the technical point of view, the specification for the blending layer is complete. By specifying only one algorithm for steganography, we lack the possibility to switch algorithms and this makes the blending layer potentially weaker as there is no seconding algorithm such as PQt providing cryptoagility. While F5 has been available for many years, no paper has been published proving detectability of the algorithm itself. F5 was analyzed among others and showed remarkably resistant to conventional attacks. Detectability is depending on the density of embedded data, a payload of 5-10 percent is currently not deemed detectable in a real world environment[54]. Many other algorithms such as nsF5, PQt/PQe, HUGO[113], S-UNIWARD[66], MiPOD[138], or HILL[86] have beeen evaluated but offering a solid implementation is nowadays rare. An implementation java was not available.

To hide a *VortexNode* from a censoring adversary, means that we have to generate credible traffic for sending messages containing imagery roughly 10-20 times as big as the embedded payload. The carrier messages must have properties assigning it a software as the source of the message (e.g., personalized evaluation documents, status information, password recovery messages, or statistics). These messages should have constantly sized attachments as it would be typical for a process to generate messages always following the same patterns. Such size restriction for an embedding image is one of the caveats for larger messages as an adaptive image size is easily detectable by an adversary.

28.2 Analysis of Plain Embedding

It is undeniable why a file treated with plain embedding is easily identifiable as a broken or tampered file. Its use is undeniale when looking at the fact that lmost 100% of the carrier media may be used. While the information may remain parseable, its content is no longer sensible to a human and thus at least suspect. Therefore, plain embedding is not suitable for use in environments with a censoring adversary and may be seen as very weak obfuscation in environments of an observing adversary.

We wanted to know if there is a simple method to detect the modifications of such a file. While most of the analysis method requires the processing of large data sets, we tried to find apparent, non-calculation-intense test methods that were

generic. We did not take any content-based method into account as they require high calculation power. As our embedding is generic, we searched for a similar detection method. This is a weak argument but we already agreed that plain embedding is not suitable for environments with a censoring adversary.

A property of encrypted ciphertext is the high entropy. We, therefore, used the calculation of the Shannon entropy in bytes as property and tried to show the shift of entropy within the files. This detection method depended very much on the type of file used for embedding. It showed an expected behavior, that file types having in the expected area a similar entropy were not detectable by this method. However, we identified some file types to be unsuitable for plain blending due to their entropy structure.

We analyzed the files by calculating the entropy of blocks 256 bytes with a sliding window over a randomly collected set of images (e.g., the first 100 entries of a file type after searching for “mouse”, “cat”, “camel”, or “dog”). We did intentionally not filter or eliminate images. Surprisingly, we were able to tell file types apart, were able to identify files with thumbnails or an interlaced structure. We even identified certain specific patterns regarding the producer type of an image (e.g., we could differentiate between pictures scanned or taken by a camera). It was not so much surprising that we were able to identify these features, but the fact that we could see them in entropy data.

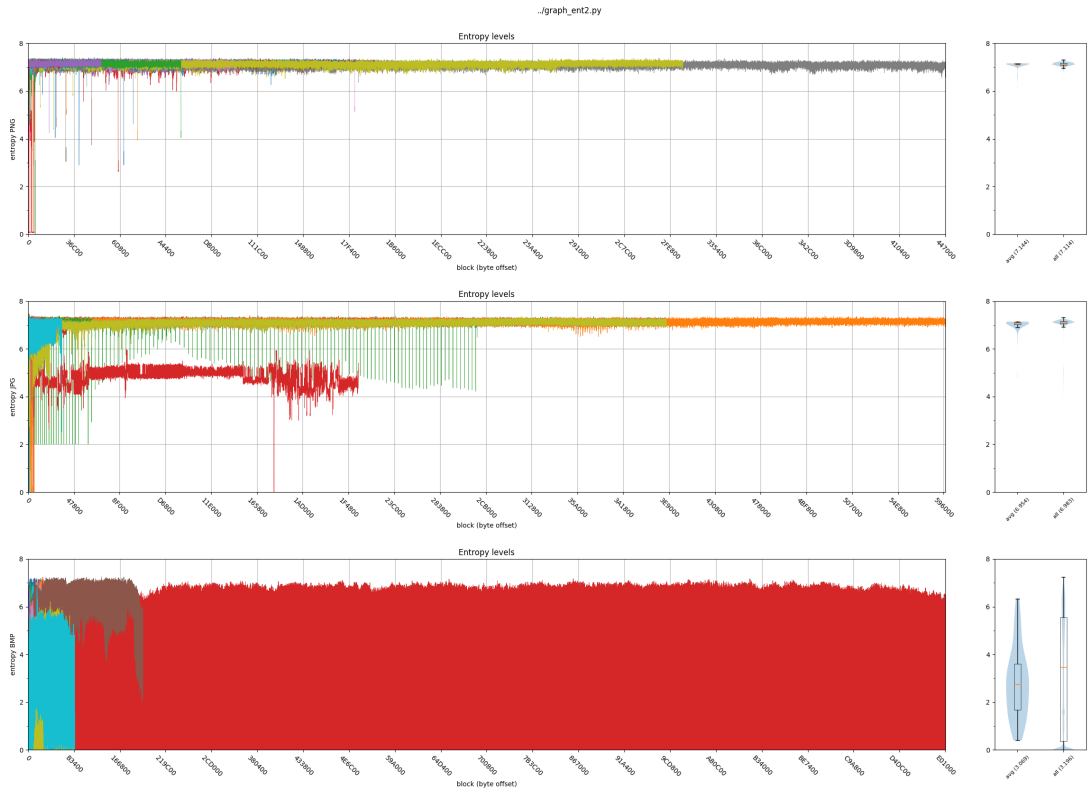


Figure 28.1: Distribution Analysis of Different, Common Graphics Formats

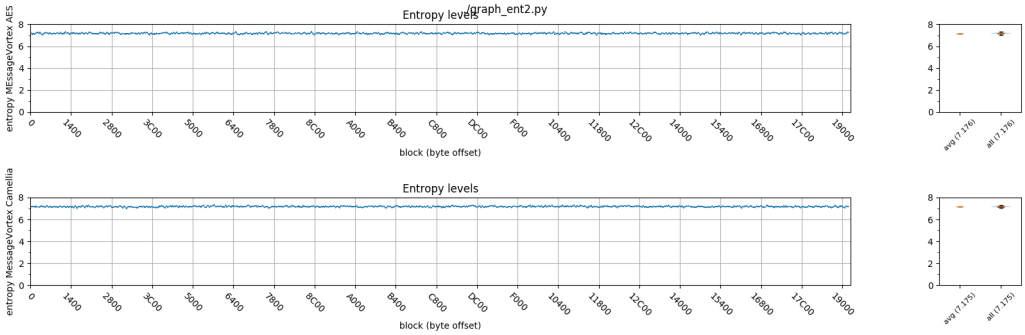


Figure 28.2: Distribution Analysis of a MessageVortex Block

We then carried out an analysis identifying the typical entropy and the inner structures. The graphs in 28.1 show a typical analysis. In that case, we looked at 100 images of each type. We graphed and analyzed their entropy and tested for the suitability of a plain embedding from an entropy poi. Table 28.1 lists the average entropy of analyzed file types and makes remarks about the suitability for plain embedding. In practice, we found that most suitable file formats have an entropy of ≈ 7.2 and an interquartile range (IQR) of 0.15 or less. Furthermore, files should have a big, uniform, the non-structured range containing these characteristics. Such a file has a suitable space for embedding. For reference, Figure 28.2 shows the distribution of typical MessageVortex blocks. We did find that the entropy must be uniformly matched in the case of plain embedding.

Criteria / Type	Avg. Entropy	IQR	Remarks
JPG	7.008	0.097	–
PNG	7.116	0.086	–
GIF	6.978	0.194	–
BMP	2.997	4.964	not suitable
PDF	6.660	0.282	Hard to embed due to a very complex inner structure but well suited
MP3	7.076	0.091	–
WAV	4.777	0.927	–
OGG	7.104	0.093	relatively easy to embedd. Hard not to break the file structure.
mpg4	n/a	n/a	good to embedd. Steganography could be applied here easily too.
zip	7.148	0.080	easy to embedd when using “password protected” archives
MVaes	7.176	0.072	Without length padding as reference encrypted with AES 256 CBC
MVcam	7.175	0.070	Without length padding as reference encrypted with Camellia 256 CBC

Table 28.1: comparison of protocols in terms of the suitability criteria as transport layer

When blending into images, BMP showed a strongly varying entropy within a file. A sampling of ten blocks at random position resulted already in detection with a false positive rate below 5%. PNG and JPG files showed to be very robust within the sample. We did not succeed in identifying the MessageVortex blending content based on entropy values. GIF images showed to be unsuitable. Archive formats such as zip files were extremely robust. We were able to embed it into a zip file and marking it (generically) as an encrypted file. This embedding was genuinely undetectable. However, such embedding may potentially lead to censorship based on blacklisting of encrypted zip files.

OGG and MP3 are suitable. However, we were able to detect the entropy dif-

ference when taking extreme dense samples. These formats may, however, be suitable for not yet standardized forms of steganography. While PDF has low entropy and a high IQR typically, some parts of the files are very well suited for embedding. Plain embedding with knowledge of the format was even possible without affecting the visual result of the file.

We could show that with an approach based on Shannon entropy, we may identify plain embedded *VortexMessages*in BMP and WAV files.

All movie formats were performing similarly to jpg and PNG. However, due to the very complex structure with scattered blocks, they seem to be unsuitable for plain embedding. They are, however, strong candidates for steganography and are being used.

28.3 Analysis of the Routing Layer

28.3.1 Analysis of the Core Operations

The core operations form a toolset for mixing messages. Under the operational restrictions outlined in section 24.3, we anlayze in the following section the operations and determine their capability for leaking information or affecting security.

28.3.1.1 Splitting and Merging

The operations $splitPAyload$ and $mergePayload$ are the trivial operations of our operations set. The operations by itself leak some information. Assuming that they were encrypted previously. A split or merge operation on its own leakes possible counterparts as the size should add up to a blocksize common in symmetric cryptography. As we outlined in section 15.2.6.3 and section 24.3 either an encryption step or an add redundancy step has to be added before a *VortexNode* may forward the block to the next layer. When doing so, we can say that the operation leaks not more than any cryptographically secure operation.

For a *VortexNode* executing the operation a split operation does not leak any additional properties. The input may be payload or not. Therefore, the output of the operation has the same properties as the input. Unless the *VortexNodes* knows the nature of the incoming payload, the output may be either decoy or true message traffic.

28.3.1.2 Encryption and Decryption Operations

All encryption steps do leak some properties. They may leak the algorithm due to the block size. The chosen parameter may be unique to the RBB. If randomly chosen, this is no longer the case. If chossen by an implementation specific

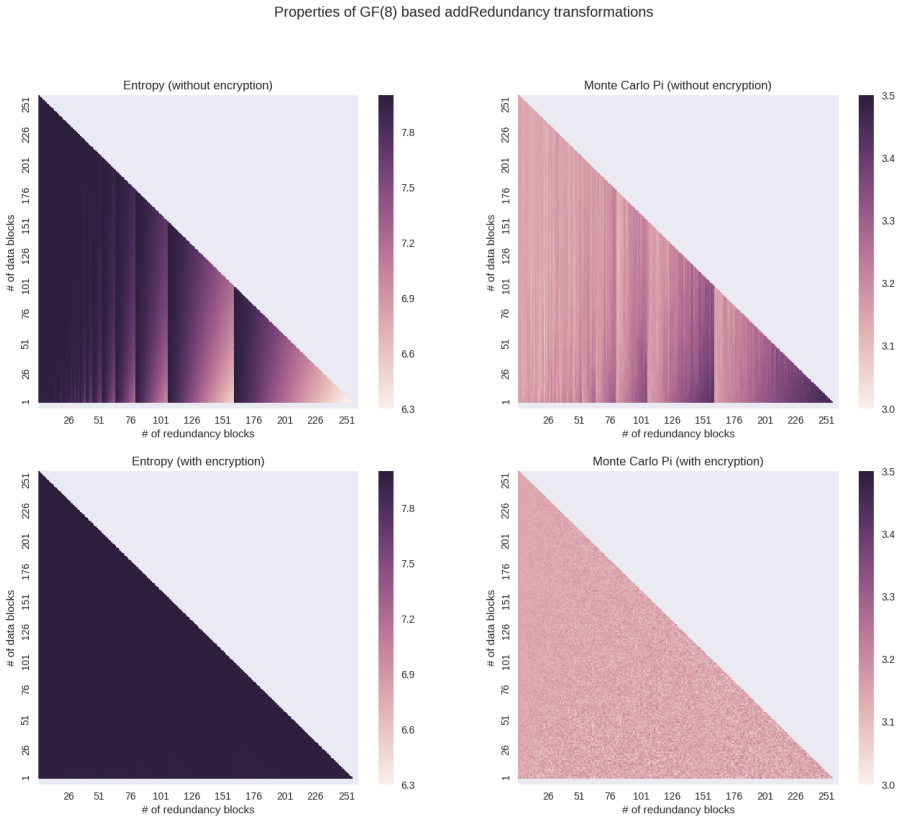


Figure 28.3: Resulting entropy of addRedundancy with and without encryption step

pattern the pattern may leak the implmentation over time. As the analysis must be done over a short period of time (the lifetime of an eID) it is up to an RBB to leak as little information as possible. We do regard the cryptographically secured content as secure.

28.3.1.3 Add and Remove Redundancy Operations

During analysis the *addRedundancy* operation showed the undesirable behavior that applying the operation lowered the entropy of the target blocks as shown in figure 28.3.

We did reconsider thus the whole operation. The choice of the Reed-Solomon (RS) operation instead of a Lagrange polynomial seemed logical. As the possibilities to recover from cheaters in an RS setting of varying contextes have been already studied in **??** and **??** and similar publications.

28.4 Senders routing layer

A sender may have some knowledge about the Routing block size and may, therefore, guess the complexity of the routing path. He is, however, unable to gain any additional information such as time of travel or number of hops until the target.

28.5 Intermediate node routing layer

An intermediate node does know all the operations applied and the immediate next hop. It does learn the routing addresses of the immediately following endpoints but is unable to use these endpoints. This is because he has no means to get the host key required to communicate.

If a routing block is repeated, a router may identify the routing block as repetition. Identifying the repetition of a block can be done by looking at the serial number of replay protection. We then may give a rough estimate of the message size by comparing the payload chunks. This estimate is, however, very rough as it is bounded by the block size of the symmetrically applied encryption.

28.6 Security of Protocol Blocks

To analyze the security of the protocol, we first go through all protocol blocks. After that, we will look at the possibilities of block recombinations and how to gain data or services based on such behavior.

Assuming plain embedding, the presence of a chain of blocks may leak an existing VortexMessage. At the moment, the protocol expects at the offset and the size of the bytes to be skipped to the next block. The encoding does not assume an end of the chain marker as such a marker would make the design identifiable. As an encoding scheme, a variable byte length has been chosen. This guarantees that any file will always result in a valid chain of blocks and thus not leak such a presence.

The entropy of the only two blocks in this stream (MPREFIX and InnerMessageBlock) is comparable as both blocks are encrypted. Both blocks are encrypted and feature a similar entropy. The blocks follow each other without any delimiter. This results in a continuous stream of data with constant properties.

To avoid repeating patters at the beginning of streams due to reused identity blocks, a MURB must provide sufficient peer keys and prefix blocks. A VortexNode may, however, refuse to process MURBS (only accept maxReplays equal to 0).

All blocks of InnerMessageBlock are protected by the peer key $E^{K_{peer}}$. The forward secrets in all blocks except the payload blocks make sure that the recombination

of blocks does not work for an adversary. To be successful, an adversary requires to know the forward secret of the next hop.

To keep the secrets of the next hop, hidden from the host assembling the message, the subsequent header and the routing block are protected by the sender key $E^{K_{sender}}$. A message assembling node is, therefore, not even capable of creating its own messages to an unknown node as the hosts' public key $E^{K^1_{host}}$ is not derivable from a message.

Therefore, a routing node is not able to assemble messages for a specific host on the base of a routed message only. A routing node does not gain any additional knowledge except for the locally executed operations, the number of messages of the ephemeral identity, the size of messages of any ephemeral identity, the sending IP of a received VortexMessage and the transport endpoint address of any receiving endpoint. The most critical information is endpoint data, as all other data is unrelated to the original message (sender recipient and size). This information becomes absolutely crucial if assuming a censoring adversary. Therefore, a sender in a jurisdiction where the use of MessageVortex is deemed illegal must use only trusted nodes within the jurisdiction and at least for the first hop outside the jurisdictional reach of an adversary.

29 Dynamic Attack Analysis

In the dynamic analysis, we reach out to an active adversary. An active adversary modifies traffic in a non-protocol conformant way, or misuses available or obtained information to disrupt messages, nodes, or the system as a whole.

29.1 Well Known Attacks

In the following sections, we emphasize on possible attacks to an anonymity preserving protocols. These attacks may be used to attack the anonymity of any entity involved in the message channel. In a later stage, we test the protocol for immunity against these classes of attacks.

29.1.1 Broken Encryption Algorithms

Encryption algorithms may become broken at any time. This either to new findings in attacking them, by more resources being available to an adversary, or by new technologies allowing new kinds of attacks. A proper protocol must be able to react to such threads promptly. This reaction should not rely on a required update of the infrastructure. Users should solely control the grade of security.

We cannot do a lot for attacks of this kind to happen. However, we might introduce a choice of algorithms, paddings, modes, and key sizes to give the user a choice

in the degree of security he wants to have.

We have introduced a way to support a set of independent cryptographic algorithms, paddings ,modes, and prngs. The support of these algorithms does not have to be uniform throughout the system, instead it is sufficient for two neighbouring nodes support the same algoithms in order to be used.

29.1.2 Attacks Targeting Anonymity

Attacks targeting users anonymity are the main focus of this work. Many pieces of information may be leaked, and the primary goal should, therefore, rely on the principles established in security.

- Prevent an attack
  Attack prevention can only be done for attacks that are already known and may not be realistic in all cases. In our protocol, we have strict boundaries defined. A node under attack should at any time of protocol usage (this excepts bandwidth depletion attacks) be able to block malicious identities. Since establishing new identities is costly for an attacker, he should always require far more resources than the defender.

- Minimize attack surface
  This part of the attack prevention is included by design in the protocol. By minimizing the information footprint we have in each operation and the disconnection between two eIDS of the same sender it is very hard to gain additional information based on statistical means.

- Redirect an attack
  Although the implementation does not do this, it is possible to handle suspected malicious *VortexNode* differently (e.g., avoid using them or only use them for decoy traffic not disclosing identities).

- Control damage
  For us, this means leaving as little information about identities or meta information as possible on untrusted infrastructures. If we leave traces (i.e., message flows, or accounting information) they should have the least possible information content and should expire within a reasonable amount of time.

- Discover an attack
  The protocol is designed in such a way that attack discovery (such as a query attack) is possible. However, we consider active attacks just as part of the regular message flow. The protocol must mitigate such attacks by design.

- Recover from an attack
  An attack does always impose a load onto a system's resources regardless of its success. It is vital that a system recovers almost immediately from an

attack and is not covered in a non-functional or only partial-functional state either temporarily or permanently.

In the following subsections, we list a couple of attack classes that have been used against systems listed in 12 or the respective academic works. We list the countermeasures which have been taken to deflect these attacks.

29.1.2.1 Probing Attacks

Identifying a node by probing and check their reaction is commonly done when fingerprinting a service. As a node is participating in a network and relaying messages probing may not be evaded. However, it may be made costly for an adversary to do systematic probing. This should be taken into account. Both currently specified transport protocol features an indefinite number of possible accounts. Since not the server but the endpoint address is behaving, node probing is more complicated than in other cases where probing of service is sufficient.

One of the problems is clear-text requests. These requests may be used on any transport layer account without previous knowledge of any host key. Thus the recommendation in table 20.1 is generally not to answer the requests. Routing nodes in jurisdictions not fearing legal repression or prosecution may reply to clear text requests, but it is usually discouraged as they allow harvesting of *VortexNodes*. A discovered *VortexNode* may leak subsequent nodes if the same account is used for receiving and sending.

One strategy to avoid would be to put high costs onto clear-text requests in such a way that a clear-text request may have a long reply time (e.g., up to one day).

A node is free to blacklist an identity in case of an early reply. This is an insufficient strategy as a big adversary may have lots of identities in stock. Requesting an unusually long key as a plain-text identity does not make sense either as these as well may be kept in stock. We may, however, force a plaintext request to have an identity block with a hash following specific rules. We may, for example, put in a requirement that the first four bytes of the hash of a header block translates to the first four characters of the routing block. At the moment, this has been rejected in the standard for practical reasons. First, as the request is unsolicited, a sender is the only one able to decide the algorithm of the hash. This would allow a requester to choose upon the complexity of the puzzle. Second, any negotiation of the cost of the request would result in the disclosure of the node as *VortexNode*, which might be unsuitable.

29.1.2.2 Hotspot Attacks

Hotspot attacks aim to isolate high traffic sites within a network. By analyzing specific properties or the general throughput locations with outstanding traffic may be identified. These messages do quite often reveal senders or recipients. Sometimes an intermediate node in an anonymizing system.

The assumption that a hotspot arrises at a specific point in our protocol is wrong, as at any point either blocks may be left out until expiry or additional traffic may be generated using an $addRedundancy$ operation.

29.1.2.3 Message Tagging and Tracing

When using an anonymization system, a message may be either fully or partially traced or even tagged. Tagging allows one to recognize a message at a later stage and map it to its predecessors. Protocols with tagable messages are not suitable for anonymization systems.

VortexMessages are not tagable. The constraint “no repeating pattern” prohibits forwarding of any block without an apropriate operation. This denies the possibility to tag a payload block. All other blocks (prefixes, header and routing block) are discarded when forwarding the message. The same applies to the carrier message which is used as transport for the blended *VortexMessage*.

Injecting a value into a payload block and following it, would imply that the evil *VortexNode*has knowledge about all subsequent operations and keys, which is equivalent to know the subsequent private keys of the *VortexNodes*. We will cover this scenario in section 29.1.2.7.

29.1.2.4 Side Channel Attacks

Side-channel attacks are numerous. Especially important to us are attacks related to either lookup in independent channels (e.g., downloading of auxiliary content of a message) or behavior related to timing patterns.

29.1.2.5 Sizing Attacks

There are two kinds of sizing attacks identified to be relevant for us. One is the possibility for matching messages with related sizes, and the other one is to relate message size to the original messages. Both attacks may be considered as a tracing attack and will be analyzed accordingly.

When matching messages in size an attack is attractive if it allows to collapse the operations of one or multiple honest *VortexNodes* between two malicious *VortexNodes*. To do so he second evil node may may match the sizes of the received payload blocks and hypothesize about which blocks are equal or it may assign the eid of the first evil node to the eID of the second node. The matching is not trivial, as...

1. The sizes are likely to have changed while transferred through the honest nodes.

2. The number of payload blocks may have changed.

3. The size may have been further obfuscated by the fact that an onionized encryption does either not add to the size (if an algorithm with the same block size is applied and no padding) or is increasing (by the block size). Obfuscation is possible as well, if we apply a $splitPayload$ or $mergePayload$ operation with a subsequent encryption (mandatory to not violate the "repeating pattern rule") or an $addRedundancy$ operation.

29.1.2.6 Bugging Attacks

Numerous attacks are available through the bugging of a protocol. In this chapter, we outline some of the possibilities and how they may be countered:

- Bugging through certificate or identity lookup:
  Almost all kinds of proof of identities, such as certificates, offer some revocation facility. While this is a perfect desirable property of these infrastructures, they offer a flaw. Since the location of this revocation information is typically embedded in the proof of identity, an evil attacker might use a falsified proof of identity with a recording revocation point.

  There are multiple possibilities to counter such an attack. The easiest one is to do no verification at all. Having no verification is, however, not desirable from the security point of view. Another possibility is only to verify trusted proof of identities. By doing so, the only attacker could be someone having access to a trusted source of proof of identities. A third possibility is to relay the request to another host either by using an anonymity structure such as Tor or by using its infrastructure. Using Tor would violate the "Zero Trust" goal. Such a measure would only conceal the source of the verification. It would not hide the fact that the message is processed. A fourth and most promising technology would be to force the sender of the certificate to include a "proof of non-revocation". Such a proof could be a timestamped and signed partial CRL. It would allow a node to verify the validity of a certificate without being forced to disclose itself by doing a verification. On the downside has to be mentioned that including proof of non-revocation involves the requirement to accept a certain amount of caching time to be accepted. This allowed caching time reduces the value of the proof as it may be expired in the meantime. It is recommended to keep the maximum cache time as low as 1d to avoid that revoked certificates may be used.

- Bugging through DNS traffic:
  A standard protocol on the Internet is DNS. Almost all network-related programs use it without thinking. Typically the use of such protocol is only a minor issue since the resolution of a lookup usually done by an ISP. In the case of a small Internet service provider (ISP), this might, however, already become a problem.

  The bugging in general attack works as follows: We include a unique DNS name to be resolved by a recipient. This can be done most easily by adding

an external resource such as an image. A recipient will process this resource and might, therefore, deliver information about the frequency of reading, or the type of client.

It must be taken into account that the transport layer will always do DNS lookups and that we may not avoid this attack completely. We may, however, minimize the possibilities of this attack.

- Bugging through external resources:
  A straightforward attack is always to include external resources into a message and wait until they are fetched. In order to avoid this kind of attack, plain text or other self-contained formats should be used when sending a message. As we may not govern the type of contained message, we can make at least recommendations concerning its structure.

29.1.2.7 Analysis by Building Interaction Graphs

Building interaction graphs is very difficult for an adversary. This has several reasons.

Looking from outside the system interaction graphs are hard to build as sending and receiving transport addresses and protocols do not match, which adds tremendously to the complexity. An outside observer may not just observe a specific SMTP server. He must track incoming messages, observe the user (typically obtaining the mail by IMAP) fetching the messages and then follow all possible connections to other infrastructures known to be supported and suspect them as outbound messages.

29.1.3 Denial of Service Attacks

29.1.3.1 Censorship

Whereas traditional censorship is widely regarded as selective information filtering and alteration, very repressive censorship can even include denial of information flows in general. Any anonymity system not offering the possibility to hide in legitimate information flows, therefore not censorship-resistant.

29.1.3.2 Denial of service

An adversary may flood the system in two ways.

- He may flood the transport layer exhausting resources of the transport system.

This is a straightforward attack. MessageVortex has no control over the existing transport protocol. Therefore, all flooding attacks on that layer are still effective. However, If an adversary attacks a node, the redundancy of a message may still be sufficient. On the other hand, flooding disrupts at least all other services using the same transport layer on that node. This result may be unacceptable for an attacker. More likely would be censorship.

- He may flood the routing layer with invalid messages.
  Identifying the messages is relatively easy for a node. Usually, it should be sufficient to decode the CPREFIX block of a message. If the CPREFIX is valid, then the header block either identifies a valid identity or processing may be aborted.

- He may flood an accounting layer with newIdentity.
  Flooding an accounting layer with identities is possible. Since the accounting layer is capable of adapting costs to a new identity, it may counter this attack by giving large puzzles to new identities. This affects all new identities and not only those flooding. If a flooding attack is carried out over a long time, a node may decide to split its identity. All recent active users get a new identity, whereas the old one opposes high costs. This would force an attacker to work in intervals and is no longer able to make a permanent DoS attack.

29.1.3.3 Credibility Attack

Another type of DoS attack is the credibility attack. While not a technical attack, it is very effective. A system not having a sufficiently big user base is offering thus a lousy level of anonymity because the anonymity set is too small or the traffic concealing message flow is insufficient.

Another way is to attack the reputation of a system in such a way that the system is no longer used. An adversary has many options to achieve such a reduction in credibility. Examples:

- Disrupt functionality of a system.
  This may be done by blocking of the messaging protocol it uses or by blocking messages. Furthermore, an adversary reduces functionality when removing known participants from the network either by law or by threatening.

- Publicly dispute the effectiveness of a system.
  Disputing the effectiveness is a very effective way to destroy a system. People are not willing to use a system which believed to be compromised if the primary goal of using the system is avoiding being observed.

- Reduce the effectiveness of a system.
  A system may be considerably loaded by an adversary to decrease the positive reception of the system. He may further use the system to send UBM to reduce the overall experience when using the system. Another way of

reducing effectiveness is to misuse the system for evil purposes such as blackmailing and making them public.

- Dispute the credibility of the system founders.
  Another way of reducing the credibility of a system is to undermine its creators. If – for example – people believe that a founders' interest was to create a honey pot (e.g., because he is working for a potential state-sponsored adversary) for personal secrets, they will not be willing to use it.

- Dispute the credibility of the infrastructure.
  If the infrastructure is known or suspected to be run by a potential adversary, people's willingness to believe in such a system is expected to be drastically reduced.

29.1.3.4 Denial of Service by Exhausting Quotas or Limits

A malicious node may try to exhaust quotas or limits. As we do trust in the sender and recipient, all other nodes do not know the forward secrets used in the message. The options for an adversary are then as follows:

- Resend a MURB (with different content) as often as possible to exhaust message and transfer quota.

- Create intentionally huge, incorrect message content to exhaust transfer quota.

29.1.4 Attacking Sending and Receiving Identities of the MessageVortex System

The most valuable goal of an adversary is breaking an entity's anonymity or monitor their traffic by the content or the metadata. In the following sections, we analyze the possibility of

29.1.4.1 Traffic Highlighting

Traffic caused by a routing block may be observed to a certain extent on a statistical base. A node may generate bad message content of exceptionally large or small nature. This might potentially highlight messages involved in message routing using no split or relative split operations as well as addRedundancy operations.

29.1.5 Recovery of Previously Carried Out Operations

It is crucial that an adversary is unable to recover parameters of a previously carried out operation. We analyzed though the protocol operations carefully to be sure not to leak any of the parameters. Some operations leak apparent data such as an encryption operation with a block cipher does typically leaks its block size. This has, however, been classified as invaluable data as the block size does not result in any information gain usable for attacking the system or narrowing down efforts. In figure **??**, we can show that the parameters are visible. We took the same 10kb block and treated it with all possible combinations of operation parameters. The image shows that there is a possibility of guessing the parameter with a high probability. For guessing the average Monte Carlo Pi and the average Shanon entropy in bits per byte were already sufficient. The results got a bit less clear when applying the same operation to random blocks while doing the analysis.

We have, however, found a flaw in the$addRedundancy$ operation. When applying this operation to an encrypted block, the entropy of the resulting block leaks some of the parameters of the operation. As a result of this finding, we added a custom padding and an additional encryption step. The repeated analysis showed that the operation does no longer leak these parameters through this channel.

29.2 Side Channel Leaking

29.2.1 Software Updates and Related Data Streams

29.2.2 Bugging in transported messages

29.3 Achieved Anonymity and Flaws

29.3.1 Measuring Anonymity

It is tough to measure anonymity, as it involves many uncontrollable factors. We may, however, control the degree of anonymity according to the number of involved parties. Assuming a sender knows the complete message path, including

all operations carried out on any untrusted node a message travels through, the anonymity is maxed to the number of involved nodes n, excluding the sender nodes. This degree of $n - 1$ may be further reduced if all well-known "routing only" or at least "routing mostly" nodes are reduced. Under these harsh assumptions, the set may be reduced to the potential set of "well known" recipients of a message.

We have to differentiate between several problems. An adversary has to identify the participants of an anonymity system. Then he has to identify members of a message or a communication anonymity set. Starting from there, he has to identify message flows and detect senders and receivers of messages within an anonymity set (which is not doable in all cases). If any adversary achieves this, we have to consider the anonymity to be broken. Depending on the degree of anonymity required, which is influenced by external factors, the participation in any or a small enough set may be sufficient to suffer consequences.

29.3.2 Attacking Routing Participants

While very hard in our case as we do not have "dedicated" anonymization infrastructure, It might be possible to identify members of the routing network. This due to flaws in the blending layer. While it is possible to scare off or block members of a routing network. It is far harder in a network where the members are mobile. Any user may change at any time the identity, including the endpoint, without losing its known peers. This unique property makes the participating entities very mobile and allows them to switch servers at any time without losing contact with peers for subsequent communication.

Routing participants may be identified either by publicly available information (e.g., published routing address) or by identifying unique properties of the protocol. Transport layer provider may then be forced to deanonymize the customer related to the account (if possible), or the relating account on the transport layer may be blocked.

To counter a possible threatening deanonymization, a MessageVortex node owner must maintain anonymity towards the transport layer provider. Nowadays, this is easily done in the XMPP protocol. The account is typically not linked to any subsequent user information, such as telephone or email. Email accounts are more restrictively regulated. Providers providing accounts without registration of phone numbers or subsequent email addresses do exist (e.g., Yandex) but are rare. In both cases, a user might be identified by its IP address. This is why concealing its IP address while connecting to the transport layer is an advisable practice. Using Tor when accessing the transport layer may suffice to do so. The anonymizing service has to be strong enough to conceal the IP. The protection of the traffic itself is not required as it is already protected.

29.3.3 Attacking Anonymity through Traffic Analysis

As traffic and decoy traffic and decoy traffic are chosen by the creator of the routing block, frequency patterns cannot be detected, unlike the router did create them. The same applies to message sizes and traffic hotspots. When reusing the same routing block, eventually message sizes or general estimates such as “bigger” or “smaller size” can be made.

For an evil routing node, even paired with a global observer, it is hard to extract any useful information. An adversary might identify all messages following through it as messages of the same true identity. As ephemeral identities are short term identities, this is of limited values. By monitoring the endpoints used by an ephemeral identity, we might calculate a “likelihood of matching” for two ephemeral identities. Luckily this is not doable without allowing a high factor of uncertainty. This matching does not improve when combining multiple ephemeral identities over time. The matching might slightly improve when trying to match ephemeral identities on different routing nodes. Making strong statements about those likelihoods is not possible as we did intentionally not define a specific behavior. We may safely say that the possibility of deanonymization is degrading if using short-lived ephemeral identities.

The knowledge a node may gain from ephemeral identities is minimal. The ephemeral identity is created by a node unknown to the receiver of the request. The only thing we know is what node was adjacent when creating the ephemeral identity. As the creation of an ephemeral identity is not linked to any other identity or ephemeral identity relationship between ephemeral identities on two nodes cannot be established. If two adjacent nodes cooperate when processing two linked ephemeral identities, no additional knowledge may be won. If two collaborating nodes have one or more non-collaborating nodes between them, they lose all linking knowledge due to the non-collaborating nodes.

Operations have been carefully crafted to leak as little information as possible. Being able to encrypt or decrypt a payload block does not leak any information. The data processed may be true message traffic or decoy as we do not know what the nature of the received message was. If an RBB avoids repeating patterns of blocks on nodes, it is not possible to link ephemeral identities of two non-adjacent nodes. Repeating patterns may arise, for example, if a block pb_1 is decrypted and re-encrypted on two nodes. In this case, both nodes may match the message as it contains the same content between the operations.

node f:

$$pb_2 = D(pb_1$$
$$pb_3 = E^{K_i}(pb_2)$$

node f+1:

.

.

node f+x:

$$pb_4 = D^{K_i}(pb_3)$$

In this example the patterns of $pb3$ and $pb_4 = pb_2$ are two patterns repeating on non-adjacent nodes. The same conclusions are even more valid for splitting operations. These two operations should be regarded as helpers for the $addRedundancy$ and $removeRedundancy$ operations. These operations may be used to generate decoy traffic or to destroy data without knowledge of doing so of the processing node. If we process a function $addRedundancy_{2of3}$, any of the output blocks contains the input payload, and any two of them may be used to recover the data. At the same time, an operation $removeRedundancy_{2of3}$ may be successful or not. The node is unable to differentiate between the two states. The padding applied and the unpadded encryption makes it impossible to judge upon success or fail of an operation.

As the communication pattern is defined by the RBB and not always the same, it is hard to judge on the security. We may, however, look at some generic examples and show that we can achieve the goals byzantine fault tolerance, privacy and unlinkability, and anonymity. Figure 29.1 shows a sending node s, a series of routing nodes n_i, j assembled to routing chains. Furthermore, we have a r for which the message is destined and a set of nodes a_k building the anonymity set. Neither the number of chains j nor the length of the chains i is relevant. A node or a sequence of nodes may be part of multiple chains. By normalizing a path into such a form, we may at least analyze some properties of the protocol. We furthermore have to keep in mind that we trust sender s and receiver r. Any possible routing block may be reduced to this scheme if knowing the exact building instructions applied by the RBB.

We have to consider the fact that two adjacent nodes collaborating may build one combined workspace executing all operations. They are, therefore, able to link all operations of these two adjacent nodes and follow all incoming and outgoing paths. We, therefore, may assume that two adjacent nodes or an uninterrupted series of collaborating nodes may be substituted by one node.

So a routing node $n_{1,}$ may not know if a *VortexMessage* received from s is the result of processing another message or the message has been injected on node s. Furthermore, if s was acting as a routing node, it successfully unlinked the

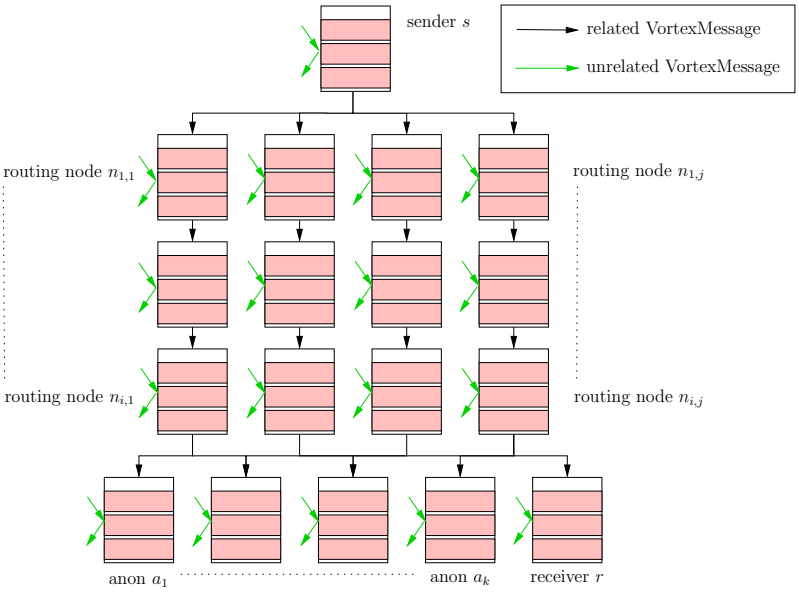


Figure 29.1: A possible path of a VortexMessage

message from any previous node. The sending node s may send a message by first employing an $addRedundancy$ operation or splitting and encrypting the message. Each path through the streams has then not enough information to rebuild the combined message. If employing an $addRedundancy$ operation, a receiver r may recover a message, if sufficient paths through the routing nodes were acting according to the protocol. Paths with misbehaving nodes may eventually be identified depending on the number of redundancy operations. Assuming that the RBB included proper padding Information for the receiver r, the receiver may identify what set of *VortexMessages* leads to the original message due to the padding applied before the RS function. So if sufficient paths, depending on the chosen operations at r, provide correct data, we may recover nodes misbehaving in our paths. If one node in a path is not collaborating with adjacent nodes in the path, the path of the *VortexMessage* becomes unlinked as previously shown with sender s. If multiple paths are used, all paths must have at least one honest node to unlink the message.

If all nodes in the anonymization set $a_1 \ldots a_k$ are honest, any preceding node may not know whether the message ends at that node or the message is just routed through an honest node. Even if some of the anonymization nodes are not honest or collaborating with an adversary, the anonymity set may be reduced in size, but the receiver is still part of the anonymity set spanning the honest anonymization nodes. So, we have shown that depending on the chosen routing block, anonymity, unlinkability, and fault tolerance against a misbehaving node may be achieved. AN RBB may furthermore send additional *VortexMessages* to suspected misbehaving nodes. If misbehavior is reproducible within an ephemeral identity, the RBB may identify it by picking up parts of the previously sent message and comparing them to an expected state. An RBB may even introduce message paths leading back to the RBB itself. Such a message path may allow observation of

the progress and success of the message delivery.

29.3.4 Attacking Anonymity through Timing Analysis

Timing is under full control of the routing block builder. No information can be derived from the timing. This is even the case if a routing block is reused. The precise timing on the network depends additionally on other factors, such as delaying through anti-UBE or anti-malware measures or delays through local delivery between multiple nodes.

29.3.5 Attacking Anonymity through Throughput Analysis

Increasing the throughput to highlight a message channel is not possible since the replay protection will block such requests. It may be possible for a limited number of times by replaying a MURB. This is one of the reasons why the usage of MURBs is discouraged unless necessary.

29.3.6 Attacking Anonymity through Routing Block Analysis

The routing block is cryptographically secure. The size of the routing block may leak an estimate about its inner complexity. It does not reveal any critical pieces of information like remaining hops to the message end or target or similar.

29.3.7 Attacking Anonymity through Header Analysis

The header contains valuable data that is cryptographically secured and only visible to the next receiver.

To an adversary not knowing the key, the size of the prefix block may leak the key size. The size of the header block itself may leak the presence of any optional blocks. Besides that, no other information is leaked to such an adversary.

To an adversary knowing the decryption key (evil routing node), the content of the header block is visible. This header block leaks all routing information for the respective node and thus the ephemeral identity. This block leaks some information of minimal value. It may leak the activity of an ephemeral identity, including frequency. This activity is, however, only matching the minimal activity of an endpoint identity as an endpoint may have multiple ephemeral identities on one node.

29.3.8 Attacking Anonymity through Payload Analysis

The payload itself does not leak any information about the message content. All content is cryptographically secured. Content may, however, leak the block size

of the applied cipher.

29.3.9 Attacking Anonymity through Bugging

Bugging is one of the most pressing problems. The protocol has been carefully crafted not to allow any bugging. The use of MIME messages in the final message, however, enables the bugging of the message itself. A bugged message content may breach receiver anonymity to the sender of the message.

29.3.10 Attacking Anonymity through Replay Analysis

Due to the replay protection, no traffic may be generated or multiplied except for the traffic sent by the attacking node. As this information is already known to the node, there is no value in doing so.

29.3.11 Diagnosability of traffic

29.3.11.1 Hijacking of Header and Routing Blocks

An attacker might try to recombine a header block of the third party with a routing block crafted to get the workspace content of a different node. To protect against this scenario, every routing block and its corresponding header block has a shared value called forwardSecret. As the content of a hijacked header block is not known, he is unable to guess the forward secret within the block.

It is not possible to brute-force the value due to the replay protection. More precisely, the probability of hijacking a single identity block is $\frac{1}{2^{32}}$. Hijacking such a block allows onetime access to the working space and is visible to the owner due to the manipulated quotas. Failing an attack will result in deleting the ephemeral identity, and a new, unlinked ephemeral identity will be created.

29.3.11.2 Partial Implicit Routing Diagnosis

We can create data that is routed back to or through the original sending node. This traffic is well defined and may be used to certify that the loop processing the message is working as expected. By combining the messages and sending intermediate results through multiple paths, it is even possible to extract the sub status of some loops and combine the result within transfer into a single message.

As a special case, a sender may use implicit routing diagnostic to diagnose the full route. A sender may do this by taking specific excerpts of the received message at the recipients' node and route these blocks back from the recipient to the sender.

29.3.11.3 Partial Explicit Routing Diagnosis

If a message fails to deliver according to implicitly routing diagnosis, additional messages may be sent to pick up the content of the workspace of ephemeral identities throughout the path. These messages are due to the only binding to the ephemeral identity, not distinguishable from the original messages. Assuming that a node always behaves either according to or not according to the rules of the system, a node may be identified by capturing built blocks with known content.

If a node is identified as a misbehaving node, it may be excluded from subsequent routing requests or reduced in its reliability or trustability ratings. A node may calculate such scores locally to build a more reliable network over time, avoiding misbehaving or non-conformant nodes. This does not violate our zero-trust philosophy as the scoring is made locally and relies on our observations.

30 Analysis of the effectiveness of Attack Schemes

31 Analysis of the Degree of Anonymization in Comparison to other Systems

32 Recommendations on Using the Vortex Protocol

The following sections list recommendations using the VortexProtocol. It is a summary of the previous sections.

32.1 Reuse of Routing blocks

Routing blocks should not be reused. The reuse of a routing block may leak some limited information to an adversary node such as approximate message size or message frequency of an unknown tupel using this network.

32.2 Use of Ephemeral Identities

Ephemeral identities should be used for a minimal number of messages. Using multiple identities with overlapping lifespans is considered a good practice. Using

different ephemeral identities for the same message is acceptable and can be a good practice as long as operations do not leak the linking between those two identities.

Special care must be taken if using overlapping ephemeral identities across nodes. While ephemeral identities may be completely unlinked on a single node, the linking between multiple nodes may leave a trace from one identity to the next. It is advisable to recreate on a regular base all ephemeral identities from scratch. This guarantees an unlinking from previous ephemeral identities.

32.3 Recommendations on Operations applied on Nodes

All operations, carried out on a single node, have to be crafted in such a way that no information whether the operation is a decoy or a real message is leaked. Otherwise, it becomes possible to narrow down the message flow.

Encryption operations should be either strictly encrypting or strictly decrypting. At no point in the path, a previously applied encryption on an untrusted node should be removed as removal might lead to linking to the previous inverse operation.

Similarly, there are rules for adding and removing redundancy information. As these operations serve as decoy traffic generators, great care needs to be taken not to leak this information. We emphasize here again that it is possible to add redundancy information on one node, encrypt one or multiple blocks once, or multiple blocks on a second node, and then remove the redundancy information again from the new set. This will lead to a payload data block than the original. However, this does not qualify the block as decoy traffic. The process may be reversed on the final recipient. Such an operation is, however, mathematically very demanding if the same operation is used for redundancy at the same time as multiple possible tuples need to be tried if one node has failed.

Whenever possible, the reappearance of a payload block in a single encoding it should be avoided or limited to an absolute minimum as such an occurrence allows linking of two ephemeral identities.

32.4 Reuse of Keys, IVs or Routing patterns

An RBB should avoid reuse of any keys, IVs, routing patterns, or PRNG seeds along its routing path of untrusted nodes. Reusing such values would allow an attacker to match ephemeral identities to a single identity. While this is minimal risk and may be ignored in some cases, an RBB should avoid it as it may leak information to collaborating nodes.

32.5 Recommendations on Choosing involved Nodes

Involved nodes should be trustworthy but not necessarily trusted. A message should always include a set of known recipients. It is regarded as a good practice to use a minimal fixed anonymity set of known recipients as routers. Doing so does not leak any information unless always the same pattern of operations is applied (see 32.1).

32.6 Message content

Although it is possible to embed any content into a Vortex message, great care should be taken as the content may allow disclosing a reader's identity or location. For this reason, only self-contained messages should be used (such as plain text messages).

Allowing a user to use more complex representations such as MIME offers many possibilities for the bugging of the content. A client displaying such messages should always handle them with great care. Taping messages by downloading external images or verifying the validity by OCSP or even doing a reverse lookup on an IP address may leak valuable information.

32.6.1 Splitting of message content

Message content should be split and distributed among routing nodes. Splitting should, however, not be done excessively to avoid failure due to too many failing nodes. It furthermore makes diagnostics complicated.

32.7 Routing

32.7.1 Redundancy

Redundancy is a valuable feature of the protocol. It allows unsuspicious decoy generation and to compensate message path disruption. A routing block should always be crafted in such a way that redundancy is aligned with the complexity of the routing block and the importance of a message to avoid an adversary controlling all nodes except for the sender's and receiver's one.

32.7.2 Operation Considerations

Operations should be kept easy, but at the same time, guarantee anonymity. The following recommendations are kept to an absolute minimum in order not to create any identifiable behavior.

A payload block should always have a single representation only once when traveling through routing nodes. A recurring pattern would allow an evil router to identify and thus match an ephemeral identity of one router to an ephemeral identity of another router even if there are multiple routes in between. So, when applying encryption only operations between routing nodes, the encryption should be onionized. A clear onionizing routing pattern (only showing encryption steps on a single chunk) is OK. A pattern such as removing encryption and then reapply different encryption is not.

32.7.3 Anonymity

Anonymity is greatly dependent on the quality of the routing block and the chosen anonymity set for a single message and a communication tuple over time.

32.7.3.1 Size of the Anonymity Set

The requirement for an anonymity set is dependent on jurisdictional restrictions. In some of the more restrictive countries, no one can be held guilty for an action that may not be credibly assigned to him alone. In other jurisdictions, it is possible to be held liable for actions just because of an identified membership in a group. This makes it essential that message traffic and the crafting of the blending is under the sole control of the sender. He needs to create an anonymity-set sufficiently large and spanning enough jurisdictions to create sufficient anonymity for his situation.

33 To be placed (TBP)

$$
\begin{aligned}
VortexMessage =& \langle \mathbf{MP}^{K^{-1}_{hostN}}, \mathbf{CP}^{K^{-1}_{hostN}}, \mathbf{H}^{K_{senderN}}, E^{K^{-1}_{senderN}}\left(H\left(\mathbf{HEADER}\right)\right) \\
& \left[\mathbf{R}^{K_{senderN}}\right], [\mathbf{PL}] * \rangle^{K_{peerN}} \rangle && (33.1)\\
\mathbf{MP}^{K^{-1}_{hostN}} =& E^{K^{-1}_{hostN}}\left(\mathbf{PREFIX}\langle K_{peerN}\rangle\right) && (33.2)\\
\mathbf{CP}^{K^{-1}_{hostN}} =& E^{K^{-1}_{hostN}}\left(\mathbf{CPREFIX}\langle K_{senderN}\rangle\right) && (33.3)\\
\mathbf{H}^{K_{senderN}} =& E^{K_{senderN}}\left(\mathbf{HEADER}\right) && (33.4)\\
\mathbf{HEADER} =& \langle K^{1}_{senderN}, serial, maxReplays, validity, [requests, requestRoutingBlock], \\
& [puzzleIdentifier, proofOfWork] \rangle && (33.5)\\
\mathbf{R}^{K_{senderN}} =& E^{K_{senderN}}\left(\mathbf{ROUTING}\right) && (33.6)\\
\mathbf{ROUTING} =& \langle [\mathbf{ROUTINGCOMBO}] *, forwardSecret, replyBlock \rangle && (33.7)\\
\mathbf{ROUTINGCOMBO} =& \langle processIntervall, K_{peerN+1}, recipient, \mathbf{nextCP}, \mathbf{nextMP}, \\
& \mathbf{nextHEADER}, \mathbf{nextROUTING}, assemblyInstructions, id \rangle \\
& && (33.8)\\
\mathbf{PL} =& \langle \text{payload octets} \rangle * && (33.9)\\
& && (33.10)
\end{aligned}
$$

Figure 33.1: Mathematical representation of a *VortexMessage*

Part VIII

Discussion on Results

***

Limit your inputs to only those that support a certain kind of self-destructive behavior, and you can be cheered with enthusiasm as you drive yourself off a cliff.

Adam-Troy Castro

34 Measuring up to the Requirenets

In this section we analyze the level of achievment in respect to the requirements defined in section13.2.1. We will go through each requirement and discuss the level of achievement. In the case of a failure, we highlight the reason for the failure and elaborate on the consequences of the current flaws.

35 Achieved level of anonymity

36 Weaknesses of the protocol

37 Further and Missing Research

The current blending layer is simple in its inner working. It creates context-less messages based on an easily recognizable scheme. An unsuspecting observer may have the impression that this is just a way of communicating, but censor may, by observing the message flow easily and conclude that these messages are not written by a human. Such detection could lead to censorship of the respective routing node and thus disrupt the message flow. It is easy to recover from such censorship by advertising a new identity to known peer partners. To minimize the effects of censorship, an improvement in this area would help.

To be undetectable, all work done by the blending layer has to be indistinguishable from regular human communication. This applies not only to the message steganographic embedding of the message but to the message content as well. This is very much similar to the problems of chatterbots these days. Assuming that a blending layer is only communicating with other nodes correctly embedding messages, we have a chatterbot problem. It is reduced as the chatterbot must only reply credibly and undetectable to generated messages of other chatterbots. If assuming that a blending layer replies to any non-Vortex nodes, the problem boils down to a Turing test, as stated in [155]. As we defined that an adversary has enormous but limited resources, this blending is, however, sufficient if it is done “good enough”. What criteria would apply here is a topic for further research. Applying any research to this topic would require to add a more precise adversary model.

The currently applied choice of transport layer protocol is a snapshot of current Internet traffic. While done with great care, it must be adapted to the changing communication habits of humanity. Identifying new or depreciated communication protocols and blending schemes would be another field of research.

A comprehensive survey of the newest trends and techniques in steganography is another topic to be covered. It would allow identifying new candidates for blending techniques. Especially interesting are steganography algorithms covering movie file formats.

This is especially hard since true evidence of in-depth protocol usage seems to be completely missing. While we were able to gather much data which is collected by simple routers (such as bandwidth), credible figures about client and content usage seemed to be completely missing or of very poor quality.

Anonymity has effects on the behavior of humans. We have found that although there is some research in this field (such as [117]), the evidence is very weak. Although the possibility of anonymity is undisputed among so-called free countries, the downsides (e.g., misuse for criminal acts) of anonymity are apparent. More research in this field is required. On the other hand, a lack of awareness for anonymity, especially in “non-free” jurisdiction, has been observed, which would be another relevant field of research.

Part IX

Appendix

***

Limit your inputs to only those that support a certain kind of self-destructive behavior, and you can be cheered with enthusiasm as you drive yourself off a cliff.

Adam-Troy Castro

A The RFC draft document

Workgroup: Internet Engineering Task Force
Internet-Draft: draft-gwerder-messagevortexmain-05
Published: 12 September 2020
Intended Status: Experimental
Expires: 16 March 2021
Author: M. Gwerder
FHNW

MessageVortex Protocol

Abstract

The MessageVortex (referred to as Vortex) protocol achieves different degrees of anonymity, including sender, receiver, and third-party anonymity, by specifying messages embedded within existing transfer protocols, such as SMTP or XMPP, sent via peer nodes to one or more recipients.

The protocol outperforms others by decoupling the transport from the final transmitter and receiver. No trust is placed into any infrastructure except for that of the sending and receiving parties of the message. The creator of the routing block (Routing block builder;RBB) has full control over the message flow. Routing nodes gain no non-obvious knowledge about the messages even when collaborating. While third-party anonymity is always achieved, the protocol also allows for either sender or receiver anonymity.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 16 March 2021.

Copyright Notice

Table of Contents

1. Introduction

Anonymisation is hard to achieve. Most previous attempts relied on either trust in a dedicated infrastructure or a specialized networking protocol.

Instead of defining a transport layer, Vortex piggybacks on other transport protocols. A blending layer embeds MessageVortex messages (VortexMessage) into ordinary messages of the respective transport protocol. This layer picks up the messages, passes them to a routing layer, which applies local operations to the messages, and resends the new message chunks to the next recipients.

A processing node learns as little as possible from the message or the network utilized. The operations have been designed to be sensible in any context. The 'onionized' structure of the protocol makes it impossible to follow the trace of a message without having control over the processing node.

MessageVortex is a protocol which allows sending and receiving messages by using a routing block instead of a destination address. With this approach, the sender has full control over all parameters of the message flow.

A message is split and reassembled during transmission. Chunks of the message may carry redundant information to avoid service interruptions during transit. Decoy and message traffic are not differentiable as the nature of the addRedundancy operation allows each generated portion to be either message or decoy. Therefore, any routing node is unable to distinguish between message and decoy traffic.

After processing, a potential receiver node knows if the message is destined for it (by creating a chunk with ID 0) or other nodes. Due to missing keys, no other node may perform this processing.

This RFC begins with general terminology (see Section 2) followed by an overview of the process (see Section 3). The subsequent sections describe the details of the protocol.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.2. Protocol Specification

Appendix A specifies all relevant parts of the protocol in ASN.1 (see [CCITT.X680.2002] and [CCITT.X208.1988]). The blocks are DER encoded, if not otherwise specified.

1.3. Number Specification

All numbers within this document are, if not suffixed, decimal numbers. Numbers suffixed with a small letter 'h' followed by two hexadecimal digits are octets written in hexadecimal. For example, a blank ASCII character (' ') is written as 20h and a capital 'K' in ASCII as 4Bh.

2. Entities Overview

The following entities used in this document are defined below.

2.1. Node

The term 'node' describes any computer system connected to other nodes, which support the MessageVortex Protocol. A 'node address' is typically an email address, an XMPP address or other transport protocol identity supporting the MessageVortex protocol. Any address SHOULD include a public part of an 'identity key' to allow messages to transmit safely. One or more addresses MAY belong to the same node.

2.1.1. Blocks

A 'block' represents an ASN.1 sequence in a transmitted message. We embed messages in the transport protocol, and these messages may be of any size.

2.1.2. NodeSpec

A nodeSpec block, as specified in Section a.6, expresses an addressable node in a unified format. The nodeSpec contains a reference to the routing protocol, the routing address within this protocol, and the keys required for addressing the node. This RFC specifies transport layers for XMPP and SMTP. Additional transport layers will require an extension to this RFC.

2.1.2.1. NodeSpec for SMTP nodes

An alternative address representation is defined that allows a standard email client to address a Vortex node. A node SHOULD support the smtpAlternateSpec (its specification is noted in ABNF as in [RFC5234]). For applications with QR code support, an implementation SHOULD use the smtpUrl representation.

```
    localPart         = <local part of address>
domain            = <domain part of address>
email             = localPart "@" domain
keySpec           = <BASE64 encoded AsymmetricKey [DER encoded]>
smtpAlternateSpec = localPart ".." keySpec ".." domain "@localhost"
smtpUrl           = "vortexsmtp://" smtpAlternateSpec
```

This representation does not support quoted local part SMTP addresses.

2.1.2.2. NodeSpec for XMPP nodes

Typically, a node specification follows the ASN.1 block NodeSpec. For support of XMPP clients, an implementation SHOULD support the jidAlternateSpec (its specification is noted in ABNF as in [RFC5234]).

```
    localPart         = <local part of address>
domain            = <domain part of address>
resourcePart      = <resource part of the address>
jid               = localPart "@" domain [ "/" resourcePart ]
keySpec           = <BASE64 encoded AsymmetricKey [DER encoded]>;
jidAlternateSpec  = localPart ".." keySpec ".."
                    domain "@localhost" [ "/" resourcePart ]
jidUrl            = "vortexxmpp://" jidAlternateSpec
```

2.2. Peer Partners

This document refers to two or more message sending or receiving entities as peer partners. One partner sends a message, and all others receive one or more messages. Peer partners are message specific, and each partner always connects directly to a node.

2.3. Encryption keys

Several keys are required for a Vortex message. For identities and ephemeral identities (see below), we use asymmetric keys, while symmetric keys are used for message encryption.

2.3.1. Identity Keys

Every participant of the network includes an asymmetric key, which SHOULD be either an EC key with a minimum length of 384 bits or an RSA key with a minimum length of 2048 bits.

The public key must be known by all parties writing to or through the node.

2.3.2. Peer Key

Peer keys are symmetrical keys transmitted with a Vortex message and are always known to the node sending the message, the node receiving the message, and the creator of the routing block.

A peer key is included in the Vortex message as well as the building instructions for subsequent Vortex messages (see RoutingCombo in Appendix A).

2.3.3. Sender Key

The sender key is a symmetrical key protecting the identity and routing block of a Vortex message. It is encrypted with the receiving peer key and prefixed to the identity block. This key further decouples the identity and processing information from the previous key.

A sender key is known to only one peer of a Vortex message and the creator of the routing block.

2.4. Vortex Message

The term 'Vortex message' represents a single transmission between two routing layers. A message adapted to the transport layer by the blending layer is called a 'blended Vortex message' (see Section 3).

A complete Vortex message contains the following items:

- The peer key, which is encrypted with the host key of the node and stored in a prefixBlock, protects the inner Vortex message (innerMessageBlock).
- The sender key, also encrypted with the host key of the node, protects the identity and routing block.
- The identity block, protected by the sender key, contains information about the ephemeral identity of the sender, replay protection information, header requests (optional), and a requirement reply (optional).
- The routing block, protected by the sender key, contains information on how subsequent messages are processed, assembled, and blended.
- The payload block, protected by the peer key, contains payload chunks for processing.

2.5. Message

A message is content to be transmitted from a single sender to a recipient. The sender uses a routing block either built itself or provided by the receiver to perform the transmission. While a message may be anonymous, there are different degrees of anonymity as described by the following.

- If the sender of a message is not known to anyone else except the sender, then this degree is referred to as 'sender anonymity.'
- If the receiver of a message is not known to anyone else except the receiver, then the degree is 'receiver anonymity.'
- If an attacker is unable to determine the content, original sender, and final receiver, then the degree is considered 'third-party anonymity.'
- If a sender or a receiver may be determined as one of a set of <k> entities, then it is referred to as k-anonymity[KAnon].

A message is always MIME encoded as specified in [RFC2045].

2.6. Key and MAC specifications and usage

MessageVortex uses a unique encoding for keys. This encoding is designed to be small and flexible while maintaining a specific base structure.

The following key structures are available:

- SymmetricKey
- AsymmetricKey

MAC does not require a complete structure containing specs and values, and only a MacAlgorithmSpec is available. The following sections outline the constraints for specifying parameters of these structures where a node MUST NOT specify any parameter more than once.

If a crypto mode is specified requiring an IV, then a node MUST provide the IV when specifying the key.

2.6.1. Asymmetric Keys

Nodes use asymmetric keys for identifying peer nodes (i.e., identities) and encrypting symmetric keys (for subsequent de-/encryption of the payload or blocks). All asymmetric keys MUST contain a key type specifying a strictly-normed key. Also, they MUST contain a public part of the key encoded as an X.509 container and a private key specified in PKCS#8 wherever possible.

RSA and EC keys MUST contain a keySize parameter. All asymmetric keys SHOULD contain a padding parameter, and a node SHOULD assume PKCS#1 if no padding is specified.

NTRU specification MUST provide the parameters "n", "p", and "q".

2.6.2. Symmetric Keys

Nodes use symmetric keys for encrypting payloads and control blocks. These symmetric keys MUST contain a key type specifying a key, which MUST be in an encoded form.

A node MUST provide a keySize parameter if the key (or, equivalently, the block) size is not standardized or encoded in the name. All symmetric key specifications MUST contain a mode and padding parameter. A node MAY list multiple padding or mode parameters in a ReplyCapability block to offer the recipient a free choice.

2.7. Transport Address

The term 'transport address' represents the token required to address the next immediate node on the transport layer. An email transport layer would have SMTP addresses, such as 'vortex@example.com,' as the transport address.

2.8. Identity

2.8.1. Peer Identity

The peer identity may contain the following information of a peer partner:

- A transport address (always) and the public key of this identity, given there is no recipient anonymity.
- A routing block, which may be used to contact the sender. If striving for recipient anonymity, then this block is required.
- The private key, which is only known by the owner of the identity.

2.8.2. Ephemeral Identity

Ephemeral identities are temporary identities created on a single node. These identities MUST NOT relate to another identity on any other node so that they allow bookkeeping for a node. Each ephemeral identity has a workspace assigned, and may also have the following items assigned.

- An asymmetric key pair to represent the identity.
- A validity time of the identity.

2.8.3. Official Identity

An official identity may have the following items assigned.

- Routing blocks used to reply to the node.
- A list of assigned ephemeral identities on all other nodes and their projected quotas.
- A list of known nodes with the respective node identity.

2.9. Workspace

Every official or ephemeral identity has a workspace, which consists of the following elements.

- Zero or more routing blocks to be processed.
- Slots for a payload block sequentially numbered. Every slot:
  - MUST contain a numerical ID identifying the slot.
  - MAY contain payload content.
  - If a block contains a payload, then it MUST contain a validity period.

2.10. Multi-use Reply Blocks

'Multi-use reply blocks' (MURB) are a special type routing block sent to a receiver of a message or request. A sender may use such a block one or several times to reply to the sender linked to the ephemeral identity, and it is possible to achieve sender anonymity using MURBs.

A vortex node MAY deny the use of MURBs by indicating a maxReplay equal to zero when sending a ReplyCapability block. An unobservable node SHOULD deny the use of MURBs.

2.11. Protocol Version

This Document describes the version 1 of the protocol. The message PrefixBlock contains an optional version indicator. If absent protocol version 1 should be assumed.

3. Layer Overview

The protocol is designed in four layers as shown in Figure 1.

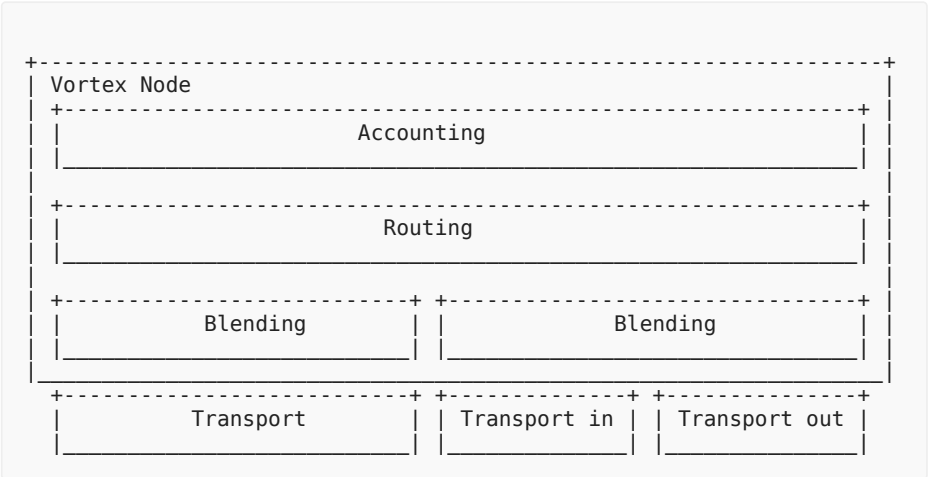


Figure 1: Layer overview

Every participating node MUST implement the layer's blending, routing, and accounting. There MUST be at least one incoming and one outgoing transport layer available to a node. All blending layers SHOULD connect to the respective transport layers for sending and receiving packets.

3.1. Transport Layer

The transport layer transfers the blended Vortex messages to the next vortex node and stores it until the next blending layer picks up the message.

The transport layer infrastructure SHOULD NOT be specific to anonymous communication and should contain significant portions of non-Vortex traffic.

3.2. Blending Layer

The blending layer embeds blended Vortex Message into the transport layer data stream and extracts the packets from the transport layer.

3.3. Routing Layer

The routing layer expands the information contained in MessageVortex packets, processes them, and passes generated packets to the respective blending layer.

3.4. Accounting Layer

The accounting layer tracks all ephemeral identities authorized to use a MessageVortex node and verifies the available quotas to an ephemeral identity.

4. Vortex Message

4.1. Overview

Figure 2 shows a Vortex message. The enclosed sections denote encrypted blocks, and the three or four-letter abbreviations denote the key required for decryption. The abbreviation k_h stands for the asymmetric host key, and sk_p is the symmetric peer key. The receiving node obtains this key by decrypting MPREFIX with its host key k_h. Then, sk_s is the symmetric sender key. When decrypting the MPREFIX block, the node obtains this key. The sender key protects the header and routing blocks by guaranteeing the node assembling the message does not know about upcoming identities, operations, and requests. The peer key protects the message, including its structure, from third-party observers.

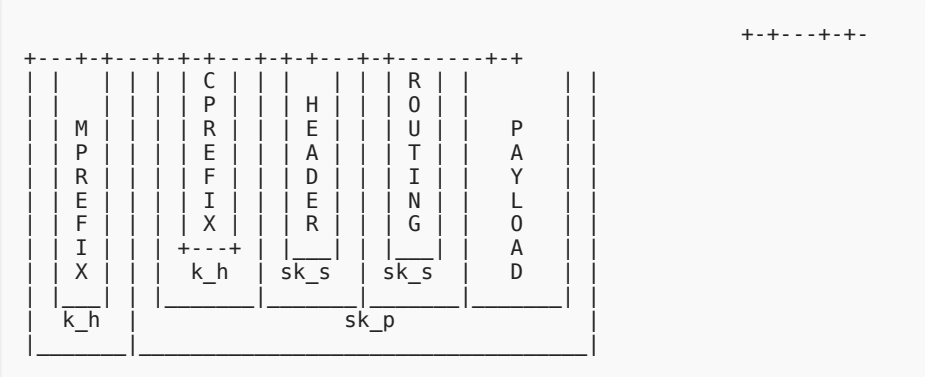


Figure 2: Vortex message overview

4.2. Message Prefix Block (MPREFIX)

The PrefixBlock contains a symmetrical key as defined in Section a.1 and is encrypted using the host key of the receiving peer host. The symmetric key utilized MUST be from the set advertised by a CapabilitiesReplyBlock (see Section 7.2.6). A node MAY

choose any parameters omitted in the CapabilitiesReplyBlock freely unless stated otherwise in Section 7.2.6. A node SHOULD avoid sending unencrypted PrefixBlocks, and a prefix block MUST contain the same forward-secret as the other prefix as well as the routing and header blocks. A host MAY reply to a message with an unencrypted message block, but any reply to a message SHOULD be encrypted.

The sender MUST choose a key which may be encrypted with the host key in the respective PrefixBlock using the padding advertised by the CapabilitiesReplyBlock.

4.3. Inner Message Block

A node MUST always encrypt an InnerMessageBlock with the symmetric key of the PrefixBlock to hide the inner structure of the message. The InnerMessageBlock SHOULD always accommodate four or more payload chunks.

4.3.1. Control Prefix Block

Control prefix (CPREFIX) and MPREFIX blocks share the same structure and logic as well as containing the sender key sk_s. If an MPREFIX block is unencrypted, a node MAY omit the CPREFIX block. An omitted CPREFIX block results in unencrypted control blocks (e.g., the HeaderBlock and RoutingBlock).

4.3.2. Control Blocks

The control blocks of the HeaderBlock and a RoutingBlock contain the core information to process the payload.

4.3.2.1. Header Block

The header block (see HeaderBlock in Appendix A) contains the following information.

- It MUST contain the local ephemeral identity of the routing block builder.
- It MAY contain header requests.
- It MAY contain the solution to a PuzzleRequired block previously opposed in a header request.

The list of header requests MAY be one of the following.

- Empty.
- Contain a single identity create request (HeaderRequestIdentity).

- Contain a single increase quota request.

If a header block violates these rules, then a node MUST NOT reply to any header request. The payload and routing blocks SHOULD still be added to the workspace and processed if the message quota is not exceeded.

4.3.2.2. Routing Block

The routing block (see RoutingBlock in Appendix A) contains the following information.

- It MUST contain a serial number uniquely identifying the routing block of this user. The serial number MUST be unique during the lifetime of the routing block.
- It MUST contain the same forward secret as the two prefix blocks and the header block.
- It MAY contain assembly and processing instructions for subsequent messages.
- It MAY contain a reply block for messages assigned to the owner of the identity.

4.3.3. Payload Block

Each InnerMessageBlock with routing information SHOULD contain at least four PayloadChunks.

5. General notes

The MessageVortex protocol is a modular protocol that allows the use of different encryption algorithms. For its operation, a Vortex node SHOULD always support at least two distinct types of algorithms, paddings or modes such that they rely on two mathematical problems.

5.1. Supported Symmetric Ciphers

A node MUST support the following symmetric ciphers.

- AES128 (see [FIPS-AES] for AES implementation details).
- AES256.
- CAMELLIA128 (see [RFC3657] Chapter 3 for Camellia implementation details).
- CAMELLIA256.

A node SHOULD support any standardized key larger than the smallest key size.

A node MAY support Twofish ciphers (see [TWOFISH]).

5.2. Supported Asymmetric Ciphers

A node MUST support the following asymmetric ciphers.

- RSA with key sizes greater or equal to 2048 ([RFC8017]).
- ECC with named curves secp384r1, sect409k1 or secp521r1 (see [SEC1]).

5.3. Supported MACs

A node MUST support the following Message Authentication Codes (MAC).

- SHA3-256 (see [ISO-10118-3] for SHA implementation details).
- RipeMD160 (see [ISO-10118-3] for RIPEMD implementation details).

A node SHOULD support the following MACs.

- SHA3-512.
- RipeMD256.
- RipeMD512.

5.4. Supported Paddings

A node MUST support the following paddings specified in [RFC8017].

- PKCS1 (see [RFC8017]).
- PKCS7 (see [RFC5958]).

5.5. Supported Modes

A node MUST support the following modes.

- CBC (see [RFC1423]) such that the utilized IV must be of equal length as the key.
- EAX (see [EAX]).
- GCM (see [RFC5288]).
- NONE (only used in special cases, see Section 11).

A node SHOULD NOT use the following modes.

- NONE (except as stated when using the addRedundancy function).

- ECB.

A node SHOULD support the following modes.

- CTR ([RFC3686]).
- CCM ([RFC3610]).
- OCB ([RFC7253]).
- OFB ([MODES]).

6. Blending

Each node supports a fixed set of blending capabilities, which may be different for incoming and outgoing messages.

The following sections describe the blending mechanism. There are currently two blending layers specified with one for the Simple Mail Transfer Protocol (SMTP, see [RFC5321]) and the second for the Extensible Messaging and Presence Protocol (XMPP, see [RFC6120]). All nodes MUST at least support "encoding=plain:0,256".

6.1. Blending in Attachments

There are two types of blending supported when using attachments.

- Plain binary encoding with offset (PLAIN).
- Embedding with F5 in an image (F5).

A node MUST support PLAIN blending for reasons of interoperability whereas a node MAY support blending using F5.

A routing block builder (RBB) MUST take care about sizing restrictions of the transport layer when composing routing blocks

6.1.1. PLAIN embedding into attachments

A blending layer embeds a VortexMessage in a carrier file with an offset for PLAIN blending. For replacing a file start, a node MUST use the offset 0. The routing node MUST choose the payload file for the message, and SHOULD use a credible payload type (e.g., MIME type) with high entropy. Furthermore, it SHOULD prefix a valid header structure to avoid easy detection of the Vortex message. Finally, a routing node SHOULD use a valid footer, if any, to a payload file to improve blending.

The blended Vortex message is embedded in one or more message chunks, each starting with a chnk header. The chunk header consists two unsigned integers of variable length. The integer starts with the LSB, and if bit 7 is set, then there is another byte following. There cannot be more than four bytes where the last, fourth byte is always 8 bit. The three preceding bytes have a payload of seven bits each, which results in a maximum number of 2^29 bits. The first of the extracted numbers (modulo remaining document bytes starting from the first and including byte of the chunk header) reflect the number of bytes in the chunk after the chunk header. The second contains the number of bytes (again modulo remaing document bytes) to be skipped after the current chunk to reach the next chunk. There exists no "last chunk" indicator. And a gap or chunk may surpass the end of the file.

```
position:00h   02h   04h   06h   08h ... 400h  402h  404h  406h
408h  40Ah
value:   01 02 03 04 05 06 07 08 09  ... 01 05 0A 0B 0C 0D 0E 0F f0
03 12 13

Embedding: "(plain:1024)"

Result:  0A 13 (+ 494 omitted bytes; then skip 12 bytes to next
chunk)
```

A node SHOULD offer at least one PLAIN blending method and MAY offer multiple offsets for incoming Vortex messages.

A plain blending is specified as the following.

```
plainEncoding = "("plain:" <numberOfBytesOfOffset>
                [ "," <numberOfBytesOfOffset> ]* ")"
```

6.1.2. F5 embedding into attachments

For F5, a blending layer embeds a Vortex message into a jpeg file according to [F5]. The password for blending may be public, and a routing node MAY advertise multiple passwords. The use of F5 adds approximately tenfold transfer volume to the message. A routing block building node SHOULD only use F5 blending where appropriate.

A blending in F5 is specified as the following.

```
f5Encoding = "(F5:" <passwordString> [ "," <PasswordString> ]* ")"
```

Commas and backslashes in passwords MUST be escaped with a backslash whereas closing brackets are treated as normal password characters unless they are the final character of the encoding specification string.

6.2. Blending into an SMTP layer

Email messages with content MUST be encoded with Multipurpose Internet Mail Extensions (MIME) as specified in [RFC2045]. All nodes MUST support BASE64 encoding and MUST test all sections of a MIME message for the presence of a VortexMessage.

A vortex message is present if a block containing the peer key at the known offset of any MIME part decodes correctly.

A node SHOULD support SMTP blending for sending and receiving. For sending SMTP, the specification in [RFC5321] must be used. TLS layers MUST always be applied when obtaining messages using POP3 (as specified in [RFC1939] and [RFC2595]) or IMAP (as specified in [RFC3501]). Any SMTP connection MUST employ a TLS encryption when passing credentials.

6.3. Blending into an XMPP layer

For interoperability, an implementation SHOULD provide XMPP blending.

Blending into XMPP traffic is performed using the [XEP-0231] extension of the XMPP protocol.

PLAIN and F5 blending are acceptable for this transport layer.

7. Routing

7.1. Vortex Message Processing

7.1.1. Processing of incoming Vortex Messages

An incoming message is considered initially unauthenticated. A node should consider a VortexMessage as authenticated as soon as the ephemeral identity is known and is not temporary.

For an unauthenticated message, the following rules apply.

- A node MUST ignore all Routing blocks.
- A node MUST ignore all Payload blocks.
- A node SHOULD accept identity creation requests in unauthenticated messages.
- A node MUST ignore all other header requests except identity creation requests.
- A node MUST ignore all identity creation requests belonging to an existing identity.

A message is considered authenticated as soon as the identity used in the header block is known and not temporary. A node MUST NOT treat a message as authenticated if the specified maximum number of replays is reached. For authenticated messages, the following rules apply.

- A node MUST ignore identity creation requests.
- A node MUST replace the current reply block with the reply block provided in the routing block (if any). The node MUST keep the reply block if none is provided.
- A node SHOULD process all header requests.
- A node SHOULD add all routing blocks to the workspace.
- A node SHOULD add all payload blocks to the workspace.

A routing node MUST decrement the message quota by one if a received message is authenticated, valid, and contains at least one payload block. If a message is identified as duplicate according to the reply protection, then a node MUST NOT decrement the message quota.

The message processing works according pseudo-code shown below.

```
function incomming_message(VortexMessage blendedMessage) {
  try{
    msg = unblend( blendedMessage );
    if( not msg ) {
      // Abort processing
      throw exception( "no embedded message found" )
    } else {
      hdr = get_header( msg )
      if( not known_identity( hdr.identity ) {
        if( get_requests( hdr ) contains HeaderRequestIdentity ) {
          create_new_identity( hdr ).set_temporary( true )
          send_message( create_requirement( hdr )  )
        } else {
          // Abort processing
          throw exception( "identity unknown" )
        }
      } else {
        if( is_duplicate_or_replayed( msg ) ) {
          // Abort processing
          throw exception "duplicate or replayed message" )
        } else {
          if( get_accounting( hdr.identity ).is_temporary() ) {
            if( not verify_requirement( hdr.identity, msg ) ) {
              get_accounting( hdr.identity ).set_temporary( false )
            }
          }
          if( get_accounting( hdr ).is_temporary() ) {
            throw exception( "no processing on temporary identity" )
          }

          // Message authenticated

get_accounting( hdr.identity ).register_for_replay_protection( msg )
          if( not verify_mtching_forward_secrets( msg ) ) {
            throw exception( "forward secret missmatch" )
          }
          if( contains_payload( msg ) ) {

if( get_accounting( hdr.identity ).decrement_message_quota() ) {
              while index,nextPayloadBlock =
get_next_payload_block( msg ) {
                add_workspace( header.identity, index,
nextPayloadBlock )
              }
              while nextRoutingBlock =
get_next_routing_block( msg ) {
                add_workspace( hdr.identity,
add_routing( nextRoutingBlock ) )
              }
              process_reserved_mapping_space( msg )
              while nextRequirement = get_next_requirement( hdr ) {
                add_workspace( hdr.identity, nextRequirement )
              }
```

```
            } else {
              throw exception( "Message quota exceeded" )
            }
          }
        }
      }
    } catch( exception e ) {
      // Message processing failed
      throw e;
    }
  }
```

7.1.2. Processing of Routing Blocks in the Workspace

A routing workspace consists of the following items.

- The identity linked to, which determines the lifetime of the workspace.
- The linked routing combos (RoutingCombo).
- A payload chunk space with the following multiple subspaces available:
  - ID 0 represents a message to be embedded (when reading) or a message to be extracted to the user (when written).
  - ID 1 to ID maxPayloadBlocks represent the payload chunk slots in the target message.
  - All blocks between ID maxPayloadBlocks + 1 to ID 32766 belong to a temporary routing block-specific space.
  - ID 32767 MUST be used to signal a solicited reply block.
  - All blocks between ID 32768 to ID 65535 belong to a shared space available to all operations of the identity.

The accounting layer typically triggers processing and represents either a cleanup action or a routing event. A cleanup event deletes the following information from all workspaces.

- All processed routing combos.
- All routing combos with expired usagePeriod.
- All payload chunks exceeding the maxProcess time.
- All expired objects.
- All expired puzzles.

- All expired identities.
- All expired replay protections.

Note that maxProcessTime reflects the number of seconds since the arrival of the last octet of the message at the transport layer facility. A node SHOULD NOT take additional processing time (e.g., for anti-UBE or anti-virus) into account.

The accounting layer triggers routing events occurring at least the minProcessTime after the last octet of the message arrived at the routing layer. A node SHOULD choose the latest possible moment at which the peer node receives the last octet of the assembled message before the maxProcessTime is reached. The calculation of this last point in time where a message may be set SHOULD always assume that the target node is working. A sending node SHOULD choose the time within these bounds randomly. An accounting layer MAY trigger multiple routing combos in bulk to further obfuscate the identity of a single transport message.

First, the processing node escapes the payload chunk at ID 0 if needed (e.g., a non-special block is starting with a backslash). Next, it executes all processing instructions of the routing combo in the specified sequence. If an instruction fails, then the block at the target ID of the operation remains unchanged. The routing layer proceeds with the subsequent processing instructions by ignoring the error. For a detailed description of the operations, see Section 7.4. If a node succeeds in building at least one payload chunk, then a VortexMessage is composed and passed to the blending layer.

7.1.3. Processing of Outgoing Vortex Messages

The blending layer MUST compose a transport layer message according to the specification provided in the routing combo. It SHOULD choose any decoy message or steganographic carrier in such a way that the dead parrot syndrome, as specified in [DeadParrot], is avoided.

7.2. Header Requests

Header requests are control requests for the anonymization system. Messages with requests or replies only MUST NOT affect any quota.

7.2.1. Request New Ephemeral Identity

Requesting a new ephemeral identity is performed by sending a message containing a header block with the new identity and an identity creation request (HeaderRequestIdentity) to a node. The node MAY send an error block (see Section 7.3.1) if it rejects the request.

If a node accepts an identity creation request, then it MUST send a reply. A node accepting a request without a requirement MUST send back a special block containing "no error". A node accepting a request under the precondition of a requirement to be fulfilled MUST send a special block containing a requirement block.

A node SHOULD NOT reply to any clear-text requests if the node does not want to disclose its identity as a Vortex node officially. A node MUST reply with an error block if a valid identity is used for the request.

7.2.2. Request Message Quota

Any valid ephemeral identity may request an increase of the current message quota to a specific value at any time. The request MUST include a reply block in the header and may contain other parts. If a requested value is lower than the current quota, then the node SHOULD NOT refuse the quota request and SHOULD send a "no error" status.

A node SHOULD reply to a HeaderRequestIncreaseMessageQuota request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message or a "no error" status message.

7.2.3. Request Increase of Message Quota

A node may request to increase the current message quota by sending a HeaderRequestIncreaseMessageQuota request to the routing node. The value specified within the node is the new quota. HeaderRequestIncreaseMessageQuota requests MUST include a reply block, and a node SHOULD NOT use a previously sent MURB to reply.

If the requested quota is higher than the current quota, then the node SHOULD send a "no error" reply. If the requested quota is not accepted, then the node SHOULD send a requestedQuotaOutOfBand reply.

A node accepting the request MUST send a RequirementBlock or a "no error block."

7.2.4. Request Transfer Quota

Any valid ephemeral identity may request to increase the current transfer quota to a specific value at any time. The request MUST include a reply block in the header and may contain other parts. If a requested value is lower than the current quota, then the node SHOULD NOT refuse the quota request and SHOULD send a "no error" status.

A node SHOULD reply to a HeaderRequestIncreaseTransferQuota request (see Appendix A) of a valid ephemeral identity. The reply MUST include a requirement, an error message or a "no error" status message.

7.2.5. Query Quota

Any valid ephemeral identity may request the current message and transfer quota. The request MUST include a reply block in the header and may contain other parts.

A node MUST reply to a HeaderRequestQueryQuota request (see Appendix A), which MUST include the current message quota and the current message transfer quota. The reply to this request MUST NOT include a requirement.

7.2.6. Request Capabilities

Any node MAY request the capabilities of another node, which include all information necessary to create a parseable VortexMessage. Any node SHOULD reply to any encrypted HeaderRequestCapability.

A node SHOULD NOT reply to clear-text requests if the node does not want to disclose its identity as a Vortex node officially. A node MUST reply if a valid identity is used for the request, and it MAY reply to unknown identities.

7.2.7. Request Nodes

A node may ask another node for a list of routing node addresses and keys, which may be used to bootstrap a new node and add routing nodes to increase the anonymization of a node. The receiving node of such a request SHOULD reply with a requirement (e.g., RequirementPuzzleRequired).

A node MAY reply to a HeaderRequest request (see Appendix A) of a valid ephemeral identity, and the reply MUST include a requirement, an error message or a "no error" status message. A node MUST NOT reply to an unknown identity, and SHOULD always reply with the same result set to the same identity.

7.2.8. Request Identity Replace

This request type allows a receiving node to replace an existing identity with the identity provided in the message, and is required if an adversary manages to deny the usage of a node (e.g., by deleting the corresponding transport account). Any sending node may recover from such an attack by sending a valid authenticated message to another identity to provide the new transport and key details.

A node SHOULD reply to such a request from a valid known identity, and the reply MUST include an error message or a "no error" status message.

7.2.9. Request Upgrade

This request type allows a node to request a new version of the software in an anonymous, unliked manor. The identifier MUST identify the software product uniquely. The version MUST reflect the version tag of the currently installed version or a similarly usable tag.

7.3. Special Blocks

Special blocks are payload messages that reflect messages from one node to another and are not visible to the user. A special block starts with the character sequence '\special' (or 5Ch 73h 70h 65h 63h 69h 61h 6Ch) followed by a DER encoded special block (SpecialBlock). Any non-special message decoding to ID 0 in a workspace starting with this character sequence MUST escape all backslashes within the payload chunk with an additional backslash.

7.3.1. Error Block

An error block may be sent as a reply contained in the payload section. The error block is embedded in a special block and sent with any provided reply block. Error messages SHOULD contain the serial number of the offending header block and MAY contain human-readable text providing additional messages about the error.

7.3.2. Requirement Block

If a node is receiving a requirement block, then it MUST assume that the request block is accepted, is not yet processed, and is to be processed if it meets the contained requirement. A node MUST process a request as soon as the requirement is fulfilled, and MUST resend the request as soon as it meets the requirement.

A node MAY reject a request, accept a request without a requirement, accept a request upon payment (RequirementPaymentRequired), or accept a request upon solving a proof of work puzzle (RequirementPuzzleRequired).

7.3.2.1. Puzzle Requirement

If a node requests a puzzle, then it MUST send a RequirementPuzzleRequired block. The puzzle requirement is solved if the node receiving the puzzle is replying with a header block that contains the puzzle block, and the hash of the encoded block begins with the bit sequence mentioned in the puzzle within the period specified in the field 'valid.'

A node solving a puzzle requires sending a VortexMessage to the requesting node, which MUST contain a header block that includes the puzzle block and MUST have a MAC fingerprint starting with the bit sequence as specified in the challenge. The receiving node calculates the MAC from the unencrypted DER encoded HeaderBlock with the algorithm specified by the node. The sending node may achieve the requirement by adding a proofOfWork field to the HeaderBlock containing any content fulfilling the criteria. The sending node SHOULD keep the proofOfWork field as short as possible.

7.3.2.2. Payment Requirement

If a node requests a payment, then it MUST send a RequirementPaymentRequired block. As soon as the requested fee is paid and confirmed, the requesting node MUST send a "no error" status message. The usage period 'valid' describes the period during which the payment may be carried out. A node MUST accept the payment if occurring within the 'valid' period but confirmed later. A node SHOULD return all unsolicited payments to the sending address.

7.3.2.3. Upgrade

If a node requests an upgrade a ReplyUpgrade block MAY be sent. The block must contain the identifier and version of the most recent software version. The blob MAY contain the software if there is a newer one available.

7.4. Routing Operations

Routing operations are contained in a routing block and processed upon arrival of a message or when compiling a new message. All operations are reversible, and no operation is available for generating decoy traffic, which may be used through encryption of an unpadded block or the addRedundancy operation.

All payload chunk blocks inherit the validity time from the message routing combos as arrival time + max(maxProcessTime).

When applying an operation to a source block, the resulting target block inherits the expiration of the source block. When multiple expiration times exist, the one furthest in the future is applied to the target block. If the operation fails, then the target expiration remains unchanged.

7.4.1. Mapping Operation

The straightforward mapping operation is used in inOperations of a routing block to map the routing block's specific blocks to a permanent workspace.

7.4.2. Split and Merge Operations

The split and merge operations allow splitting and recombining message chunks. A node MUST adhere to the following constraints.

- The operation must be applied at an absolute (measuring in bytes) or relative (measured as a float value in the range 0>value>100) position.
- All calculations must be performed according to IEEE 754 [IEEE754] and in 64-bit precision.
- If a relative value is a non-integer result, then a floor operation (i.e., cutting off all non-integer parts) determines the number of bytes.
- If an absolute value is negative, then the size represents the number of bytes counted from the end of the message chunk.
- If an absolute value is greater than the number of bytes in a block, then all bytes are mapped to the respective target block, and the other target block becomes a zero byte-sized block.

An operation MUST fail if relative values are equal to, or less than, zero. An operation MUST fail if a relative value is equal to, or greater than, 100. All floating-point operations must be performed according to [IEEE754] and in 64-bit precision.

7.4.3. Encrypt and Decrypt Operations

Encryption and decryption are executed according to the standards mentioned above. An encryption operation encrypts a block symmetrically and places the result in the target block. The parameters MUST contain IV, padding, and cipher modes. An encryption operation without a valid parameter set MUST fail.

7.4.4. Add and Remove Redundancy Operations

The addRedundancy and removeRedundancy operations are core to the protocol. They may be used to split messages and distribute message content across multiple routing nodes. The operation is separated into three steps.

1. Pad the input block to a multiple of the key block size in the resulting output blocks.
2. Apply a Vandermonde matrix with the given sizes.
3. Encrypt each resulting block with a separate key.

The following sections describe the order of the operations within an addRedundancy operation. For a removeRedundancy operation, invert the functions and order. If the removeRedundancy has more than the required blocks to recover the information, then it should take only the required number beginning from the smallest. If a seed and PRNG are provided, then the removeRedundancy operation MAY test any combination until recovery is successful.

7.4.4.1. Padding Operation

Padding is done in multiple steps. First, we calculate the padding value p. We then concatenate the padding value p as 32 bit little-endian unint with the message and fill the remaining bytes required with the seeded PRNG.

A processing node calculates the final length of all payload blocks, including redundancy. This is done by in three steps followed by the calculation of the padding value p.

1. i=len(<input block>) [calculate the size of the input block]
2. e=lcm(<Blocksize of output encyrption in # bytes>,<# of output blocks>) [Calculate Minimum size of the output block]

3. l=roof((i+4+C2)/e)*e [Calculate the final length of the padded stream suitable for the subsequent operations. C2 is a constant which is either provided by the RBB or 0 if not specified.]
4. p=i+(C1*l(mod (roof((2^32-1-i)/l)*))) [Calculate padding value p. C1 is a positive integer constant and MUST be provided by the RBB to maintain diagnosability.]

The remainder of the input block, up to length L, is padded with random data. A routing block builder should specify the value of the $randomInteger$. If not specified the routing node may choose a random positive integer value. A routing block builder SHOULD specify a PRNG and a seed used for this padding. If GF(16) is applied, then all numbers are treated as little-endian representations. Only GF(8) and GF(16) are allowed fields.

The length of 0 is a valid length

This padding guarantees that each resulting block matches the block size of the subsequent encryption operation and does not require further padding.

For padding removal, the padding p at the start is first removed as a little-endian integer. Second, the length of the output block is calculated by applying <output block size in bytes>=p (mod <input block size in bytes>-4)

7.4.4.2. Apply Matrix

Next, the input block is organized in a data matrix D of dimensions (inrows, incols) where incols=(<number of data blocks>-<number of redundancy blocks>) and inrows=L/(<number of data blocks>-<number of redundancy blocks>). The input block data is first distributed in this matrix across, and then down.

Next, the data matrix D is multiplied by a Vandermonde matrix V with its number of rows equal to the incols calculated and columns equal to the <number of data blocks>. The content of the matrix is formed by v(i,j)=pow(i,j), where i reflects the row number starting at 0, and j reflects the column number starting at 0. The calculations described must be carried out in the GF noted in the respective operation to be successful. The completed operation results in matrix A.

7.4.4.3. Encrypt Target Block

Each row vector of A is a new data block encrypted with the corresponding encryption key noted in the keys of the addRedundancyOperation. If there are not enough keys available, then the keys used for encryption are reused from the beginning after the

final key is used. A routing block builder SHOULD provide enough keys so that all target blocks may be encrypted with a unique key. All encryptions SHOULD NOT use padding.

7.5. Processing of Vortex Messages

The accounting layer triggers processing according to the information contained in a routing block in the workspace. All operations MUST be executed in the sequence provided in the routing block, and any failing operation must leave the result block unmodified.

All workspace blocks resulting in IDs of 1 to maxPayloadBlock are then added to the message and passed to the blending layer with appropriate instructions.

8. Accounting

8.1. Accounting Operations

The accounting layer has two types of operations.

- Time-based (e.g., cleanup jobs and initiation of routing).
- Routing triggered (e.g., updating quotas, authorizing operations, and pickup of incoming messages).

Implementations MUST provide sufficient locking mechanisms to guarantee the integrity of accounting information and the workspace at any time.

8.1.1. Time-Based Garbage Collection

The accounting layer SHOULD keep a list of expiration times. As soon as an entry (e.g., payload block or identity) expires, the respective structure should be removed from the workspace. An implementation MAY choose to remove expired items periodically or when encountering them during normal operation.

8.1.2. Time-Based Routing Initiation

The accounting layer MAY keep a list of when a routing block is activated. For improved privacy, the accounting layer should use a slotted model where, whenever possible, multiple routing blocks are handled in the same period, and the requests to the blending layers are mixed between the transactions.

8.1.3. Routing Based Quota Updates

A node MUST update quotas on the respective operations. For example, a node MUST decrease the message quota before processing routing blocks in the workspace and after the processing of header requests.

8.1.4. Routing Based Authorization

The transfer quota MUST be checked and decreased by the number of data bytes in the payload chunks after an outgoing message is processed and fully assembled. The message quota MUST be decreased by one on each routing block triggering the assembly of an outgoing message.

8.1.5. Ephemeral Identity Creation

Any packet may request the creation of an ephemeral identity. A node SHOULD NOT accept such a request without a costly requirement since the request includes a lifetime of the ephemeral identity. The costs for creating the ephemeral identity SHOULD increase if a longer lifetime is requested.

9. Acknowledgments

Thanks go to my family who supported me with patience and countless hours as well as to Mark Zeman for his feedback challenging my thoughts and peace.

10. IANA Considerations

This memo includes no request to IANA.

Additional encryption algorithms, paddings, modes, blending layers or puzzles MUST be added by writing an extension to this or a subsequent RFC. For testing purposes, IDs above 1,000,000 should be used.

11. Security Considerations

The MessageVortex protocol should be understood as a toolset instead of a fixed product. Depending on the usage of the toolset, anonymity and security are affected. For a detailed analysis, see [MVAnalysis].

The primary goals for security within this protocol rely on the following focus areas.

- Confidentiality
- Integrity
- Availability
- Anonymity
  - Third-party anonymity
  - Sender anonymity
  - Receiver anonymity

These aspects are affected by the usage of the protocol, and the following sections provide additional information on how they impact the primary goals.

The Vortex protocol does not rely on any encryption of the transport layer since Vortex messages are already encrypted. Also, confidentiality is not affected by the protection mechanisms of the transport layer.

If a transport layer supports encryption, then a Vortex node SHOULD use it to improve the privacy of the message.

Anonymity is affected by the inner workings of the blending layer in many ways. A Vortex message cannot be read by anyone except the peer nodes and routing block builder. The presence of a Vortex node message may be detected through the typical high entropy of an encrypted file, broken structures of a carrier file, a meaningless content of a carrier file or the contextless communication of the transport layer with its peer partner. A blending layer SHOULD minimize the possibility of simply detection by minimizing these effects.

A blending layer SHOULD use carrier files with high compression or encryption. Carrier files SHOULD NOT have inner structures such that the payload is comparable to valid content. To achieve undetectability by a human reviewer, a routing block builder should use F5 instead of PLAIN blending. This approach, however, increases the protocol overhead by approximately tenfold.

The two layers of 'routing' and 'accounting' have the deepest insight into a Vortex message's inner working. Each knows the immediate peer sender and the peer recipients of all payload chunks. As decoy traffic is generated by combining chunks and applying redundancy calculations, a node can never know if a malfunction (e.g.,

during a recovery calculation) was intended. Therefore, a node is unable to distinguish a failed transaction from a terminated transaction as well as content from decoy traffic.

A routing block builder SHOULD follow the following rules not to compromise a Vortex message's anonymity.

- All operations applied SHOULD be credibly involved in a message transfer.
- A sufficient subset of the result of an addRedundancy operation should always be sent to peers to allow recovery of the data built.
- The anonymity set of a message should be sufficiently large to avoid legal prosecution of all jurisdictional entities involved, even if a certain amount of the anonymity set cooperates with an adversary.
- Encryption and decryption SHOULD follow normal usage whenever possible by avoiding the encryption of a block on a node with one key and decrypting it with a different key on the same or adjacent node.
- Traffic peaks SHOULD be uniformly distributed within the entire anonymity set.
- A routing block SHOULD be used for a limited number of messages. If used as a message block for the node, then it should be used only once. A block builder SHOULD use the HeaderRequestReplaceIdentity block to update the reply to routing blocks regularly. Implementers should always remember that the same routing block is identifiable by its structure.

An active adversary cannot use blocks from other routing block builders. While the adversary may falsify the result by injecting an incorrect message chunk or not sending a message, such message disruptions may be detected by intentionally routing information to the routing block builder (RBB) node. If the Vortex message does not carry the information expected, then the node may safely assume that one of the involved nodes is misbehaving. A block building node MAY calculate reputation for involved nodes over time and MAY build redundancy paths into a routing block to withstand such malicious nodes.

Receiver anonymity is at risk if the handling of the message header and content is not done with care. An attacker might send a bugged message (e.g., with a DKIM or DMARC header) to deanonymize a recipient. Careful attention is required when handling anything other than local references when processing, verifying, or rendering a message.

12. References

12.1. Normative References

[CCITT.X208.1988] International Telephone and Telegraph Consultative Committee, "Specification of Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.208, November 1998.

[CCITT.X680.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", November 2002.

[EAX] Bellare, M., Rogaway, P., and D. Wagner, "The EAX mode of operation", 2011.

[F5] Westfeld, A., "F5 - A Steganographic Algorithm - High Capacity Despite Better Steganalysis", 24 October 2001.

[FIPS-AES] Federal Information Processing Standard (FIPS), "Specification for the ADVANCED ENCRYPTION STANDARD (AES)", November 2011.

[IEEE754] IEEE, "754-2008 - IEEE Standard for Floating-Point Arithmetic", 29 August 2008.

[ISO-10118-3] International Organization for Standardization, "ISO/IEC 10118-3:2004 -- Information technology -- Security techniques -- Hash-functions -- Part 3: Dedicated hash-functions", March 2004.

[MODES] National Institute for Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", December 2001.

[RFC1423] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, DOI 10.17487/RFC1423, February 1993 , <https://www.rfc-editor.org/info/rfc1423>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997 , <https://www.rfc-editor.org/info/rfc2119>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003 , <https://www.rfc-editor.org/info/rfc3610>.

[RFC3657] Moriai, S. and A. Kato, "Use of the Camellia Encryption Algorithm in Cryptographic Message Syntax (CMS)", RFC 3657, DOI 10.17487/RFC3657, January 2004 , <https://www.rfc-editor.org/info/rfc3657>.

[RFC3686] Housley, R., "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, DOI 10.17487/RFC3686, January 2004 , <https://www.rfc-editor.org/info/rfc3686>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008 , <https://www.rfc-editor.org/info/rfc5234>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008 , <https://www.rfc-editor.org/info/rfc5288>.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, DOI 10.17487/RFC5958, August 2010 , <https://www.rfc-editor.org/info/rfc5958>.

[RFC7253] Krovetz, T. and P. Rogaway, "The OCB Authenticated-Encryption Algorithm", RFC 7253, DOI 10.17487/RFC7253, May 2014 , <https://www.rfc-editor.org/info/rfc7253>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016 , <https://www.rfc-editor.org/info/rfc8017>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", 21 May 2009.

[TWOFISH] Schneier, B., "The Twofish Encryptions Algorithm: A 128-Bit Block Cipher, 1st Edition", March 1999.

[XEP-0231] Peter, S.A. and P. Simerda, "XEP-0231: Bits of Binary", 3 September 2008 , <https://xmpp.org/extensions/xep-0231.html>.

12.2. Informative References

[DeadParrot] Houmansadr, A., Burbaker, C., and V. Shmatikov, "The Parrot is Dead: Observing Unobservable Network Communications", 2013 , <https://people.cs.umass.edu/~amir/papers/parrot.pdf>.

[KAnon] Ahn, L., Bortz, A., and N.J. Hopper, "k-Anonymous Message Transmission", 2003.

[MVAnalysis] Gwerder, M., "MessageVortex", 2018 , <https://messagevortex.net/devel/messageVortex.pdf>.

[RFC1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, DOI 10.17487/RFC1939, May 1996 , <https://www.rfc-editor.org/info/rfc1939>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996 , <https://www.rfc-editor.org/info/rfc2045>.

[RFC2595] Newman, C., "Using TLS with IMAP, POP3 and ACAP", RFC 2595, DOI 10.17487/RFC2595, June 1999 , <https://www.rfc-editor.org/info/rfc2595>.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003 , <https://www.rfc-editor.org/info/rfc3501>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008 , <https://www.rfc-editor.org/info/rfc5321>.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120, March 2011 , <https://www.rfc-editor.org/info/rfc6120>.

Appendix A. The ASN.1 schema for Vortex messages

The following sections contain the ASN.1 modules specifying the MessageVortex Protocol.

A.1. The Main MessageVortex Blocks

```
MessageVortex-Schema DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS PrefixBlock, InnerMessageBlock, RoutingBlock,
          maxWorkspaceID;
  IMPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec, CipherSpec
                FROM MessageVortex-Ciphers
          HeaderRequest
                FROM MessageVortex-Requests
          PayloadOperation, MapBlockOperation
                FROM MessageVortex-Operations

          UsagePeriod, BlendingSpec
                FROM MessageVortex-Helpers;

  --****************************************************************
  -- Constant definitions
  --****************************************************************
  -- maximum serial number
  maxSerial                 INTEGER ::= 4294967295
  -- maximum number of administrative requests
  maxNumberOfRequests       INTEGER ::= 8
  -- maximum number of seconds which the message might be delayed
  -- in the local queue (starting from startOffset)
  maxDurationOfProcessing   INTEGER ::= 86400
  -- maximum id of an operation
  minWorkspaceID            INTEGER ::= 32768
  -- maximum number of routing blocks in a message
  maxRoutingBlocks          INTEGER ::= 127
  -- maximum number a block may be replayed
  maxNumberOfReplays        INTEGER ::= 127
  -- maximum number of payload chunks in a message
  maxPayloadBlocks          INTEGER ::= 127
  -- maximum number of seconds a proof of non revocation may be old
  maxTimeCachedProof        INTEGER ::= 86400
  -- The maximum ID of the workspace
  maxWorkspaceId            INTEGER ::= 65535
  -- The maximum number of assembly instructions per combo
  maxAssemblyInstructions   INTEGER ::= 255


  --****************************************************************
  -- Types
  --****************************************************************
  PuzzleIdentifier ::= OCTET STRING ( SIZE(0..32) )
  ChainSecret      ::= OCTET STRING  (SIZE (16..64))

  --****************************************************************
  -- Block Definitions
  --****************************************************************
  PrefixBlock ::= SEQUENCE {
    version           [0] INTEGER OPTIONAL
    forwardsecret   [1] ChainSecret,
    key             [2] SymmetricKey,
  }
```

```
InnerMessageBlock ::= SEQUENCE {
  padding   OCTET STRING,
  prefix    CHOICE {
      plain           [11011] PrefixBlock,
      -- contains prefix encrypted with receivers
      -- public key
      encrypted       [11012] OCTET STRING
  },
  header  CHOICE  {
    -- debug/internal use only
    plain     [11021] HeaderBlock,
    -- contains encrypted identity block
    encyrpted [11022] OCTET STRING
  },
  -- contains signature of Identity [as stored in
  -- HeaderBlock; signed unencrypted HeaderBlock without
  -- Tag]
  identitySignature OCTET STRING,
  -- contains routing information (next hop) for the
  -- payloads
  routing     [11001] CHOICE {
    plain     [11031] RoutingBlock,
    -- contains encrypted routing block
    encyrpted [11032] OCTET STRING
  },
  -- contains the actual payload
  payload     SEQUENCE (SIZE (0..maxPayloadBlocks))
              OF OCTET STRING
}

HeaderBlock ::= SEQUENCE {
  -- Public key of the identity representing this
  -- transmission
  identityKey         AsymmetricKey,
  -- serial identifying this block
  serial              INTEGER (0..maxSerial),
  -- number of times this block may be replayed
  -- (Tuple is identityKey, serial while UsagePeriod
  -- of block)
  maxReplays          INTEGER (0..maxNumberOfReplays),
  -- subsequent Blocks are not processed before
  -- valid time.
  -- Host may reject too long retention. Recomended
  -- validity support >=1Mt.
  valid               UsagePeriod,
  -- represents the chained secret which has to be
  -- found in subsequent blocks prevents reassembly
  -- attack
  forwardSecret       ChainSecret,
  --  contains the MAC-Algorithm used for signing
  signAlgorithm       MacAlgorithmSpec,
  -- contains administrative requests such as quota
  -- requests
  requests            SEQUENCE
```

```
                        (SIZE (0..maxNumberOfRequests))
                        OF HeaderRequest ,
  -- Reply Block for the requests
  requestReplyBlock   RoutingCombo,
  -- padding and identitifier required to solve the
  -- cryptopuzzle
  identifier  [12201] PuzzleIdentifier OPTIONAL,
  -- This is for solving crypto puzzles
  proofOfWork [12202] OCTET STRING OPTIONAL
}

RoutingBlock ::= SEQUENCE {
  -- contains the routingCombos
  routing    [331] SEQUENCE (SIZE (0..maxRoutingBlocks))
                           OF RoutingCombo,
  -- contains the secret of the header block
  forwardSecret     ChainSecret,
  -- contains the mapping operations to map payloads
  -- to the workspace
  mappings   [332] SEQUENCE
                  (SIZE (0..maxPayloadBlocks))
                  OF MapBlockOperation,
  -- contains a routing block which may be used when
  -- sending error messages back to the quota owner
  -- this routing block may be cached for future use
  replyBlock [332]   SEQUENCE {
       murb           RoutingCombo,
       maxReplay      INTEGER,
       validity       UsagePeriod
  } OPTIONAL
}

RoutingCombo ::= SEQUENCE {
  -- contains the period when the payload should be
  -- processed.
  -- Router might refuse too long queue retention
  -- Recommended support for retention >=1h
  minProcessTime INTEGER (0..maxDurationOfProcessing),
  maxProcessTime INTEGER (0..maxDurationOfProcessing),
  -- The message key to encrypt the message
  peerKey        [401] SEQUENCE (1..maxNumberOfReplays)
                       SymmetricKey OPTIONAL,
  -- contains the next recipient
  recipient      [402] BlendingSpec,
  -- PrefixBlock encrypted with message key
  mPrefix        [403] SEQUENCE (1..maxNumberOfReplays)
                       OF OCTET STRING OPTIONAL,
  -- PrefixBlock encrypted with sender key
  cPrefix        [404] OCTET STRING OPTIONAL,
  -- HeaderBlock encrypted with sender key
  header         [405] OCTET STRING OPTIONAL,
  -- RoutingBlock encrypted with sender key
  routing        [406] OCTET STRING OPTIONAL,
  -- contains information for building messages (when
  -- used as MURB)
```

```
    -- ID 0 denotes original/local message
    -- ID 1-maxPayloadBlocks denotes target message
    -- 32768-maxWorkspaceId shared workspace for all
    -- blocks of this identity)
    assembly       [407] SEQUENCE
                   (SIZE (0..maxAssemblyInstructions))
                   OF PayloadOperation,
    -- optional for storage of the arrival time
    validity       [408] UsagePeriod OPTIONAL,
  }

END
```

A.2. The MessageVortex Ciphers Structures

```
MessageVortex-Ciphers DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec,
          MacAlgorithm, CipherSpec, PRNGType;

  CipherSpec ::= SEQUENCE {
      asymmetric  [16001] AsymmetricAlgorithmSpec OPTIONAL,
      symmetric   [16002] SymmetricAlgorithmSpec OPTIONAL,
      mac         [16003] MacAlgorithmSpec OPTIONAL,
      cipherUsage [16004] CipherUsage
  }

  CipherUsage ::= ENUMERATED {
    sign          (200),
    encrypt       (210)
  }

  SymmetricAlgorithmSpec  ::=  SEQUENCE  {
    algorithm     [16101]SymmetricAlgorithm,
    -- if ommited: pkcs7
    padding       [16102]CipherPadding OPTIONAL,
    -- if ommited: cbc
    mode          [16103]CipherMode OPTIONAL,
    parameter     [16104]AlgorithmParameters OPTIONAL
  }

  AsymmetricAlgorithmSpec  ::=  SEQUENCE  {
    algorithm     AsymmetricAlgorithm,
    -- if ommited: pkcs1
    padding       [16102]CipherPadding OPTIONAL,
    parameter     AlgorithmParameters OPTIONAL
  }

  SymmetricKey ::= SEQUENCE {
    keyType       SymmetricAlgorithm,
    parameter     AlgorithmParameters,
    key           OCTET STRING (SIZE(16..512))
  }

  AsymmetricKey ::= SEQUENCE {
    keyType       AsymmetricAlgorithm,
    -- private key encoded as PKCS#8/PrivateKeyInfo
    publicKey     [2] OCTET STRING,
    -- private key encoded as X.509/SubjectPublicKeyInfo
    privateKey    [3] OCTET STRING OPTIONAL
  }

  SymmetricKey ::= SEQUENCE {
    keyType       SymmetricAlgorithm,
    parameter     AlgorithmParameters,
    key           OCTET STRING (SIZE(16..512))
  }

  AsymmetricKey ::= SEQUENCE {
```

```
    keyType       AsymmetricAlgorithm,
    -- private key encoded as PKCS#8/PrivateKeyInfo
    publicKey     [2] OCTET STRING,
    -- private key encoded as X.509/SubjectPublicKeyInfo
    privateKey    [3] OCTET STRING OPTIONAL
  }

  SymmetricAlgorithm ::= ENUMERATED {
    aes128        (1000),  -- required
    aes192        (1001),  -- optional support
    aes256        (1002),  -- required
    camellia128   (1100),  -- required
    camellia192   (1101),  -- optional support
    camellia256   (1102),  -- required
    twofish128    (1200),  -- optional support
    twofish192    (1201),  -- optional support
    twofish256    (1202)   -- optional support
  }

  AsymmetricAlgorithm ::= ENUMERATED {
    rsa           (2000),
    dsa           (2100),
    ec            (2200),
    ntru          (2300)
  }
  ECCurveType ::= ENUMERATED{
    secp384r1     (2500),
    sect409k1     (2501),
    secp521r1     (2502)
  }
  AlgorithmParameters ::= SEQUENCE {
    keySize       [9000] INTEGER (0..65535) OPTIONAL,
    curveType     [9001] ECCurveType   OPTIONAL,
    iv            [9002] OCTET STRING  OPTIONAL,
    nonce         [9003] OCTET STRING  OPTIONAL,
    mode          [9004] CipherMode    OPTIONAL,
    padding       [9005] CipherPadding OPTIONAL,
    n             [9010] INTEGER       OPTIONAL,
    p             [9011] INTEGER       OPTIONAL,
    q             [9012] INTEGER       OPTIONAL,
    k             [9013] INTEGER       OPTIONAL,
    t             [9014] INTEGER       OPTIONAL
  }

  CipherMode ::= ENUMERATED {
    -- ECB is a really bad choice. Do not use unless really
    -- necessary and you are sure that the content is already
    -- encrypted
    ecb           (10000), -- optional support
    cbc           (10001), -- required
    eax           (10002), -- optional support
    ctr           (10003), -- required
    ccm           (10004), -- optional support
    gcm           (10005), -- optional support
    ocb           (10006), -- optional support
```

```
    ofb           (10007), -- optional support
    none          (10100)  -- required
  }

  CipherPadding ::= ENUMERATED {
    none           (10200), -- required
    pkcs1          (10201), -- required
    rsaesOaep      (10202), -- optional support
    oaepSha256Mgf1 (10202), -- optional support
    pkcs7          (10301), -- required
    ap             (10221)  -- required
  }

  MacAlgorithm ::= ENUMERATED {
    sha3-256      (3000), -- required
    sha3-384      (3001), -- optional support
    sha3-512      (3002), -- required
    ripemd160     (3100), -- optional support
    ripemd256     (3101), -- required
    ripemd320     (3102)  -- optional support
  }

  MacAlgorithmSpec  ::=  SEQUENCE  {
    algorithm     MacAlgorithm,
    parameter     AlgorithmParameters
  }

  PRNGAlgorithmSpec ::= SEQUENCE {
    type          PRNGType,
    seed          OCTET STRING
  }

  PRNGType ::= ENUMERATED {
    mrg32k3a      (10300), -- required
    blumMicali    (10301)  -- required
  }

END
```

A.3. The MessageVortex Request Structures

```
MessageVortex-Requests DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS HeaderRequest;
  IMPORTS RequirementBlock
                FROM MessageVortex-Requirements
          UsagePeriod, NodeSpec
                FROM MessageVortex-Helpers;

  HeaderRequest ::= CHOICE {
    identity      [0] HeaderRequestIdentity,
    capabilities  [1] HeaderRequestCapability,
    messageQuota  [2] HeaderRequestIncreaseMessageQuota,
    transferQuota [3] HeaderRequestIncreaseTransferQuota,
    quotaQuery    [4] HeaderRequestQuota,
    nodeQuery     [5] HeaderRequestNodes,
    replace       [6] HeaderRequestReplaceIdentity
  }

  HeaderRequestIdentity ::= SEQUENCE {
    period UsagePeriod
  }

  HeaderRequestReplaceIdentity ::= SEQUENCE {
    replace           SEQUENCE {
        old            NodeSpec,
        new            NodeSpec OPTIONAL
    },
    identitySignature OCTET STRING
  }

  HeaderRequestQuota ::= SEQUENCE {
  }

  HeaderRequestNodes ::= SEQUENCE {
    numberOfNodes   INTEGER (0..255)
  }

  HeaderRequestIncreaseMessageQuota ::= SEQUENCE {
    messages INTEGER (0..4294967295)
  }

  HeaderRequestIncreaseTransferQuota ::= SEQUENCE {
    size     INTEGER (0..4294967295)
  }

  HeaderRequestCapability ::= SEQUENCE {
    period UsagePeriod
  }

  HeaderRequestUpgrade ::= SEQUENCE {
    version     OCTET STRING,
    identifier  OCTET STRING
  }
```

```
END
```

A.4. The MessageVortex Replies Structures

```
MessageVortex-Replies DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS SpecialBlock;
  IMPORTS BlendingSpec, NodeSpec
                FROM MessageVortex-Helpers
          RequirementBlock
                FROM MessageVortex-Requirements
          CipherSpec, PRNGType, MacAlgorithm
                FROM MessageVortex-Ciphers
          maxGFSize
                FROM MessageVortex-Operations
          maxNumberOfReplays
                FROM MessageVortex-Schema;

  SpecialBlock ::= CHOICE {
    capabilities  [1]  ReplyCapability,
    requirement   [2]  SEQUENCE (SIZE (1..127))
                       OF RequirementBlock,
    quota         [4]  ReplyCurrentQuota,
    nodes         [5]  ReplyNodes,
    status        [99] StatusBlock
  }

  StatusBlock ::= SEQUENCE {
    code            StatusCode
  }

  StatusCode ::= ENUMERATED {

    -- System messages
    ok                          (2000),
    quotaStatus                 (2101),
    puzzleRequired              (2201),

    -- protocol usage failures
    transferQuotaExceeded       (3001),
    messageQuotaExceeded        (3002),
    requestedQuotaOutOfBand     (3003),
    identityUnknown             (3101),
    messageChunkMissing         (3201),
    messageLifeExpired          (3202),
    puzzleUnknown               (3301),

    -- capability errors
    macAlgorithmUnknown         (3801),
    symmetricAlgorithmUnknown   (3802),
    asymmetricAlgorithmUnknown  (3803),
    prngAlgorithmUnknown        (3804),
    missingParameters           (3820),
    badParameters               (3821),

    -- Mayor host specific errors
    hostError                   (5001)
  }
```

```
  ReplyNodes ::= SEQUENCE {
    node   SEQUENCE (SIZE (1..5))
           OF NodeSpec
  }

  ReplyCapability ::= SEQUENCE {
    -- supported ciphers
    cipher            SEQUENCE (SIZE (2..256)) OF CipherSpec,
    -- supported mac algorithms
    mac               SEQUENCE (SIZE (2..256)) OF MacAlgorithm,
    -- supported PRNGs
    prng              SEQUENCE (SIZE (2..256)) OF PRNGType,
    -- maximum number of bytes to be transferred (outgoing bytes in
vortex message without blending)
    maxTransferQuota  INTEGER (0..4294967295),
    -- maximum number of messages to process for this identity
    maxMessageQuota   INTEGER (0..4294967295),
    -- maximum simultaneously tracked header serials
    maxHeaderSerials  INTEGER (0..4294967295),
    -- maximum simultaneously valid build operations in workspace
    maxBuildOps       INTEGER (0..4294967295),
    -- maximum payload size
    maxPayloadSize    INTEGER (0..4294967295),
    -- maximum active payloads (without intermediate products)
    maxActivePayloads INTEGER (0..4294967295),
    -- maximum header lifespan in seconds
    maxHeaderLive     INTEGER (0..4294967295),
    -- maximum number of replays accepted,
    maxReplay         INTEGER (0..maxNumberOfReplays),
    -- Supported inbound blending
    supportedBlendingIn SEQUENCE OF BlendingSpec,
    -- Supported outbound blending
    supportedBlendingOut SEQUENCE OF BlendingSpec,
    -- supported galoise fields
    supportedGFSize   SEQUENCE OF (1..maxGF) OF INTEGER (1..maxGF)
  }

  ReplyCurrentQuota ::= SEQUENCE {
    messages INTEGER (0..4294967295),
    size     INTEGER (0..4294967295)
  }

  ReplyUpgrade ::= SEQUENCE {
    -- The offered version
    version    [0] OCTET STRING,
    -- The offered identitfier
    identifier [1] OCTET STRING,
    -- The archive or blob containing the software
    blob       [2] OPTIONAL OCTET STRING
  }

END
```

A.5. The MessageVortex Requirements Structures

```
MessageVortex-Requirements DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS RequirementBlock;
  IMPORTS MacAlgorithmSpec
                FROM MessageVortex-Ciphers
          UsagePeriod, UsagePeriod
                FROM MessageVortex-Helpers;

  RequirementBlock ::= CHOICE {
    puzzle  [1] RequirementPuzzleRequired,
    payment [2] RequirementPaymentRequired
  }

  RequirementPuzzleRequired ::= SEQUENCE {
    -- bit sequence at beginning of hash from
        -- the encrypted identity block
    challenge     BIT STRING,
    mac           MacAlgorithmSpec,
    valid         UsagePeriod,
    identifier    INTEGER (0..4294967295)
  }

  RequirementPaymentRequired ::= SEQUENCE {
    account       OCTET STRING,
    ammount       REAL,
    currency      Currency
  }

  Currency ::= ENUMERATED {
    btc      (8001),
    eth      (8002),
    zec      (8003)
  }

END
```

A.6. The MessageVortex Helpers Structures

```
MessageVortex-Helpers DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS UsagePeriod, BlendingSpec, NodeSpec;
  IMPORTS AsymmetricKey, SymmetricKey
               FROM MessageVortex-Ciphers;

  -- the maximum number of embeddable parameters
  maxNumberOfParameter      INTEGER ::= 127

  UsagePeriod ::= CHOICE {
    absolute [2] AbsoluteUsagePeriod OPTIONAL,
    relative [3] RelativeUsagePeriod OPTIONAL
  }

  AbsoluteUsagePeriod ::= SEQUENCE {
    notBefore       [0]     GeneralizedTime OPTIONAL,
    notAfter        [1]     GeneralizedTime OPTIONAL
  }

  RelativeUsagePeriod ::= SEQUENCE {
    notBefore       [0]     INTEGER OPTIONAL,
    notAfter        [1]     INTEGER OPTIONAL
  }

  -- contains a node spec of a routing point
  -- At the moment either smtp:<email> or xmpp:<jabber>
  BlendingSpec ::= SEQUENCE {
    target            [1] NodeSpec,
    blendingType      [2] IA5String,
    parameter         [3] SEQUENCE
                          ( SIZE (0..maxNumberOfParameter) )
                          OF BlendingParameter
  }

  BlendingParameter ::= CHOICE {
    offset          [1] INTEGER,
    symmetricKey    [2] SymmetricKey,
    asymmetricKey   [3] AsymmetricKey,
    passphrase      [4] OCTET STRING
  }

  NodeSpec ::= SEQUENCE {
    transportProtocol [1] Protocol,
    recipientAddress  [2] IA5String,
    recipientKey      [3] AsymmetricKey OPTIONAL
  }

  Protocol ::= ENUMERATED {
    smtp  (100),
    xmmp  (110)
  }

END
```

A.7. The MessageVortex Additional Structures

```
-- States reflected:
--   Tuple()=Val() [vallidity; allowed operations] {Store}
--
-- - Tuple(identity)=Val(messageQuota,transferQuota,
--   sequence of Routingblocks for Error Message
--   Routing) [validity; Requested at creation; may be
--   extended upon request] {identityStore}
-- - Tuple(Identity,Serial)=maxReplays ['valid' from
--   Identity Block; from First Identity Block; may only
--   be reduced] {IdentityReplayStore}

MessageVortex-NonProtocolBlocks DEFINITIONS
                                EXPLICIT TAGS ::=
BEGIN
  IMPORTS PrefixBlock, InnerMessageBlock, RoutingBlock,
          maxWorkspaceID
                FROM MessageVortex-Schema
          UsagePeriod, NodeSpec, BlendingSpec
                FROM MessageVortex-Helpers
          AsymmetricKey
                FROM MessageVortex-Ciphers
          RequirementBlock
                FROM MessageVortex-Requirements;

  -- maximum size of transfer quota in bytes of an identity
  maxTransferQuota          INTEGER ::= 4294967295
  -- maximum # of messages quota in messages of an identity
  maxMessageQuota           INTEGER ::= 4294967295

  -- do not use these blocks for protocol encoding
  -- (internal only)
  VortexMessage ::= SEQUENCE {
    prefix        CHOICE {
        plain           [10011] PrefixBlock,
        -- contains prefix encrypted with receivers public
        -- key
        encrypted       [10012] OCTET STRING
    },
    innerMessage  CHOICE {
        plain     [10021] InnerMessageBlock,
        -- contains inner message encrypted with Symmetric
        -- key from prefix
        encrypted [10022] OCTET STRING
    }
  }

  MemoryPayloadChunk ::= SEQUENCE {
    id                        INTEGER (0..maxWorkspaceID),
    payload                   [100] OCTET STRING,
    validity                  UsagePeriod
  }

  IdentityStore ::= SEQUENCE {
    identities SEQUENCE (SIZE (0..4294967295))
```

```
                OF IdentityStoreBlock
  }

  IdentityStoreBlock ::= SEQUENCE {
    valid               UsagePeriod,
    messageQuota        INTEGER (0..maxMessageQuota),
    transferQuota       INTEGER (0..maxTransferQuota),
    -- if omitted this is a node identity
    identity            [1001] AsymmetricKey OPTIONAL,
    -- if ommited own identity key
    nodeAddress         [1002] NodeSpec      OPTIONAL,
    -- Contains the identity of the owning node;
    -- May be ommited if local node
    nodeKey             [1003] SEQUENCE OF AsymmetricKey
                               OPTIONAL,
    routingBlocks       [1004] SEQUENCE OF RoutingBlock
                               OPTIONAL,
    replayStore         [1005] IdentityReplayStore,
    requirement         [1006] RequirementBlock OPTIONAL
  }

  IdentityReplayStore ::= SEQUENCE {
    replays   SEQUENCE (SIZE (0..4294967295))
              OF IdentityReplayBlock
  }

  IdentityReplayBlock ::= SEQUENCE {
    identity            AsymmetricKey,
    valid               UsagePeriod,
    replaysRemaining  INTEGER (0..4294967295)
  }

END
```

Appendix B. Changelog

Version #	Date	Changes
0	11-2018	Initial version
1	02-2019	Removed term block. Added more precise spec about blending. Change in spec for XMPP blending (from XEP-234 to XEP-231). Restructured ASN.1.

Version #	Date	Changes
2	03-2019	Language and consistency improvments. Added example for chunnked plain embedding. Added pseudocode for incomming message processing. Improved wording of hashes in ASN.1.
3	09-2019	Removed LaTeX notation in padding.
4	03-2020	Added spec for Software update using MV. Minor language improvments.
5	09-2020	Reinserted lost ASN.1 specs (unintentinally lost in last two versions). Added changelog. Modified padding to improve credibility of bad values.

Table 1: changes in versions

Author's Address

University of Applied Sciences and Arts
Northwestern Switzerland
Martin Gwerder
Bahnhofstrasse 5
CH-5210 Windisch
Switzerland
Phone: +41 56 202 76 81
Email: rfc@messagevortex.net

B Glossary

adversary In this work, we are referring to an adverser to any entity opposing to the privacy of a message. For a more throughout definition refer to section 13.1

anonymity We refer to the term anonymity as defined in [114]. “Anonymity of a subject means that the subject is not identifiable within a set of subjects, the anonymity set.”[1]

Sender Anonymity The anonymity set is the set of all possible subjects. For actors, the anonymity set consists of the subjects who might cause an action. For actees, the anonymity set consists of the subjects which might be acted upon. Therefore, a sender may be anonymous (sender anonymity) only within a set of potential senders, his/her sender anonymity set, which itself may be a subset of all subjects worldwide who may send a message from time to time.

Receiver Anonymity The same for the recipient means that a recipient may be anonymous (recipient anonymity) only within a set of potential recipients, his/her recipient anonymity set. Both anonymity sets may be disjoint, be the same, or they may overlap. The anonymity sets may vary over time.

agent An agent is a single component of a service provided to a user or other services.

carrier message A transport layer message containg an embedded *VortexMessage*. In an ideal implementation a carrier message is not identifiable as a carrier of a *VortexMessage*.

decoy traffic Any data transported between routers that have no relevance to the message at the final destination and are not needed for the flow of the message.

eID An ephemeral identity (eID) is a unique user of a vortex node characterized by its public key. This user is created with a VortexMessage and has only a limited lifetime. After expiry all informations related to this identity are deleted.

EWS Exchange Web Services (EWS) are a Microsoft proprietary protocol to access exchange services from a client. It may be regarded as an alternative to IMAPv4. This is however incomplete as EWS offers additional features such as User Configuration, Delegate Management or Unified Messaging.

identity A tuple of a routable address and a public key. This tuple is a long-living tuple but may be exchanged from time to time. An Identity is always assigned to a nod, but one node may have multiple identities.

jurisdiction A geographical area where a set of legal rules created by a single actor or a group of actors apply, which contains executive capabilities (e.g., police, army, or secret service) to enforce this set of legal rules. Most of these legal rules are based on their specific physical location (e.g., German law is limited to the jurisdiction of Germany). Some jurisdictions may over-arch multiple separated geographical locations (e.g., laws of the European Union) or specific to some handpicked countries (e.g., International Covenant on Civil and Political

[1] footnotes omitted in quote

Rights). Due to there overlapping nature multiple jurisdictions may have contradictory rules applying for the same event.

IMAP IMAP (currently IMAPv4) is a typical protocol to be used between a Client MRA and a Remote MDA. It has been specified in its current version in [32]. The protocol is capable of fully maintaining a server-based message store. This includes the capability of adding, modifying, and deleting messages and folders of a mailstore. It does not include however sending mails to other destinations outside the server-based store.

ID A numerical identification reflecting a single payload chunk in a workspace of an eID.

IoI The Item of Intrest (IoI) are defined in [114] and refer to any subject action or entity which is of interest to a potential adversary.

LMTP The Local Mail Transfer Protocol is defined in [105]. This RFC defines a protocol similar to SMTP for local mail senders. This protocol allows a sender to have no mail queue at all and thus simplifies the client implementation.

local mail store A Local Mail Store offers a persistent store on a local non-volatile memory in which messages are being stored. A store may be flat or structured (e.g., supports folders). A local mail store may be an authoritative store for mails or a "cache only" copy. It is typically not a queue.

MDA An MDA provides uniform access to a local message store.

Remote MDA A Remote MDA is typically supporting a specific access protocol to access the data stored within a local message store.

Local MDA A Local MDA is typically giving local applications access to a server store. This may be done thru an API, a named socket or similar mechanisms.

message The "real content" to be transferred from the sender to the recipient. Please note the difference compared to a *VortexMessage*. We refer to the encoded form of a *VortexMessage*, which may or may not contain parts of the original message always as *VortexMessage*.

MessageVortex The protocol described in this document.

MRA A Mail Receiving Agent is an agent, which receives emails from another agent. Depending on the used protocol two subtypes of MRAs are available.

Client MRA A client MRA picks up emails in the server mail storage from a remote MDA. Client MRAs usually connect through a standard protocol that was designed for client access. Examples for such protocols are POP or IMAP.

Server MRA Unlike a Client MRA, a server MRA listens passively for incoming connections and forwards received messages to an MTA for delivery and routing. A typical protocol supported by a server MRA is SMTP

MS-OXCMAPIHTTP Microsofts Messaging Application Programming Interface (MAPI) Extensions for HTTP specifies the Messaging Application Programming Interface (MAPI) Extensions for HTTP in [97], which enable a client to access

personal messaging and directory data on a server by sending HTTP requests and receiving responses returned on the same HTTP connection. This protocol extends HTTP and HTTPS.

MSA A Mail Sending Agent. This agent sends emails to a Server MRA.

MTA A Mail Transfer Agent. This transfer agent routes emails between other components. Typically an MTA receives emails from an MRA and forwards them to an MDA or MSA. The main task of an MTA is to provide reliable queues and solid track of all emails as long as they are not forwarded to another MTA or local storage.

MTS A Mail Transfer Service. This is a set of agents which provide the functionality to send and receive messages and forward them to a local or remote store.

MSS A Mail Storage Service. This is a set of agents providing a reliable store for local mail accounts. It also provides Interfacing which enables clients to access the users' mail.

MUA A Mail User Agent. This user-agent reads emails from local storage and allows a user to read existing emails, create and modify emails.

MURB A multi use reply block. This tzpe of routing block is provided by a sender to give a node the possibilitz to route back answers without the knowledge of a location of the sender. In contrast to a SURB a MURB may be used mutiple times. The number of times is regulated by the $maxReplay$ field. Furthermore a MURB must provide multiple peer keys for all routing steps to avoid repeating patterns of key blocks. This structure makes a MURB much bigger than a SURB.

operation A function transforming the content of a payload block. MessageVortex supports four categories of operations. Relevant for the service are $addRedundancy$/$removeRedundancy$, $encrypt$/$decrypt$, and $split$/$merge$. Additionally for operation there is a $mapping$ operation allowing to map the payloads of a message into the payload space or vice-versa.

payload Any data transported between routers regardless of the meaningfulness or relevance to the VortexMessage.

privacy From the Oxford English Dictionary: "

1. The state or condition of being withdrawn from the society of others, or from the public interest; seclusion. The state or condition of being alone, undisturbed, or free from public attention, as a matter of choice or right; freedom from interference or intrusion.
2. Private or retired place; private apartments; places of retreat.
3. Absence or avoidance of publicity or display; a condition approaching to secrecy or concealment. Keeping of a secret.
4. A private matter, a secret; private or personal matters or relations; The private parts.

5. Intimacy, confidential relations.

6. The state of being privy to some act.

"[146]
In this work, privacy is related to definition two. Mails should be able to be handled as a virtual private place where no one knows who is talking to whom and about what or how frequent (except for directly involved people).

pseudonymity As Pseudonymity we take the definition as specified in [114].

> A pseudonym is an identifier of a subject other than one of the subject's real names. The subject which the pseudonym refers to is the holder of the pseudonym. A subject is pseudonymous if a pseudonym is used as an identifier instead of one of its real names.[2]

POP POP (currently in version 3) is a typical protocol to be used between a Client MRA and a Remote MDA. Unlike IMAP, it is not able to maintain a mail store. Its sole purpose is to fetch and delete emails in a server-based store. Modifying Mails or even handling a complex folder structure is not doable with POP

recipient The user or process destined to receive the message in the end.

router Any *VortexNode*which is processing messages. Please note that all VortexNodes are routers.

routing block A block in the VortexMessage containing all the instructions for processing the current message. It may contain, furthermore, addotional routing blocks to compose subsequent messages. The routing block is protected by the sender key K_{sender}.

routing graph A graphical representation of a routing block. A routing graph is a directed multigraph with *VortexNodes* as nodes and *VortexMessages* as edges. For further details see section 24.3.1.

RBB A routing block builder (RBB) is a VortexNode assembling the operations and hops for a message. If the RBB is not equal to the sender of the message the receiver may be anonymous to the sender.

sender The user or process originally composing the message. we refer as the sender both the human creator or initiator of a message, as well as the process assembling and preparing the message.

server admin We do regard a server admin as a person with high privileges and profound technical knowledge of a server and its associated technology. A Server Admin may have access to one or multiple servers of the same kind.

service A service is an endpoint on a server providing the functionality to a client. This service may consist of several Agents (agent).

SMTP SMTP is the most commonly used protocol for sending emails across the Internet. In its current version it has been specified in [73].

[2] footnotes omitted in quote

storage A store to keep data. It is assumed to be temporary or persistent.

SURB A single use reply block. This type of routing block is provided by a sender to give a node the possibility to route back answers without the knowledge of a location of the sender. A SURB may only be used once subsequent uses of the block are not possible. The lifetime of a SURB is typically limited to minutes or hours.

UBM We use the term Unsolicited Bulk Message as a term for any mass message being received by a user without prior explicit consent. A less formal term for such a message in email terminology is spam or junk mail.

undetectability As undetectability we take the definition as specified in [114].

> Undetectability of an item of interest (IOI) from an attacker's perspective means that the attacker cannot sufficiently distinguish whether it exists or not.[3]

unlikability We refer to the term unlinkability as defined in [114]. "Unlinkability of two or more items of interest (IOIs, e.g., subjects, messages, actions, ...) from an attacker's perspective means that within the system (comprising these and possibly other items), the attacker cannot sufficiently distinguish whether these IOIs are related or not.

unobservability As unobservability we take the definition as specified in [114].

> Unobservability of an item of interest (IOI) means
>
> - undetectability of the IOI against all subjects uninvolved in it and
> - anonymity of the subject(s) involved in the IOI even against the other subject(s) involved in that IOI.

As mentioned in this paper, unobservability raises the bar of required attributes again ($\Rightarrow$ reads "implies"):

$$\begin{aligned} censorship\ resistance &\Rightarrow unobservability \\ unobserability &\Rightarrow undetectability \\ unobserability &\Rightarrow anonymity \end{aligned}$$

user Any entity operating a *VortexNode*.

VortexMessage The encoded message passed from one VortexNode to another one. The VortexMessage is typically considered before any embedding takes place.

VortexNode A hardware node running the MessageVortex specific software. These nodes typically run on always-connected, user-run devices such as mobile phones or tablets.

[3] footnotes omitted in quote

workspace A storage uniquely allocated for a specific eID. Within this workspace, we find all received payloads referred by an ID, all routing blocks to be processed, and all unexpired operations.

XMPP The Extensible Messaging and Presence Protocol (XMPP)[129, 130] was formerly also known as Jabber protocol. It is an extensible instant messenger protocol widely adopted in chat clients.

zero trust Zero trust is not a truly researched model in systems engineering. It is, however, widely adopted. We refer in this work to the zero trust model when denying the trust in any infrastructure not directly controlled by the sending or receiving entity. This distrust extends especially but not exclusively to the network transporting the message, the nodes storing and forwarding messages, the backup taken from any system except the client machines of the sending and receiving parties, and software, hardware, and operators of all systems not explicitly trusted. As explicitly trusted in our model, we do regard the user sending a message (and his immediate hardware used for sending the message), and the users receiving the messages. Trust in between the receiving parties (if more than one) of a message is not necessarily given.

B Bibliography

[1] *754-2008 - IEEE Standard for Floating-Point Arithmetic - Redline*. Aug. 2008. DOI: 10.1109/IEEESTD.2008.5976968. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=5976966 (cit. on p. 116).

[2] Spencer Ackerman. *NSA warned to rein in surveillance as agency reveals even greater scope*. Newspaper. June 2013. URL: https://www.theguardian.com/world/2013/jul/17/nsa-surveillance-house-hearing (cit. on p. 4).

[3] Luis von Ahn, Andrew Bortz, and Nicholas J. Hopper. "k-Anonymous Message Transmission". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*. Ed. by Vijay Atluri and Peng Liu. ACM Press, Oct. 2003, pp. 122–130. DOI: 10.1145/948109.948128. URL: http://www.abortz.com/papers/k-anon.pdf (cit. on pp. 14, 57).

[4] Herve Aiache, Cedric Tavernier, and Corinne Sieux. "Reed-Solomon codes and multi-path strategies to improve privacy performance over ad hoc networks". In: *2008 3rd International Symposium on Wireless Pervasive Computing*. IEEE. 2008, pp. 430–435. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4556244 (cit. on p. 36).

[5] Oriol Amat, John Blake, and Jack Dowds. *THE ETHICS OF CREATIVE ACCOUNTING*. Journal of Economic Literature classification. Dec. 1999. DOI: 10.1007/BF02639318. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.198.7724&rep=rep1&type=pdf (cit. on p. 5).

[6] *AMQP v1.0*. AMQP.org, Oct. 2011. URL: http://www.amqp.org/confluence/display/AMQP/AMQP+Specification (cit. on p. 102).

[7] Banks Andrew and Gupta Rahul. *MQTT*. en. OASIS, Apr. 2014. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.pdf (cit. on p. 101).

[8] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. "PIR with compressed queries and amortized query processing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 962–979. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8418648 (cit. on p. 50).

[9] Sebastian Angel and Srinath Setty. "Unobservable communication over fully untrusted infrastructure". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 551–569. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-angel.pdf (cit. on p. 50).

[10] James Ball. *NSA's Prism surveillance program: how it works and what it can do*. Newspaper. June 2013. URL: https://www.theguardian.com/world/2013/jun/08/nsa-prism-server-collection-facebook-google (cit. on pp. 3, 4).

[11] Marco Valerio Barbera, Vasileios P. Kemerlis, Vasilis Pappas, and Angelos Keromytis. "CellFlood: Attacking Tor Onion Routers on the Cheap". In: *Proceedings of ESORICS 2013*. Sept. 2013. URL: http://www.cs.columbia.edu/~vpk/papers/cellflood.esorics13.pdf (cit. on p. 42).

[12] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. “Low-Resource Routing Attacks Against Tor”. In: *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2007)*. Washington, DC, USA, Oct. 2007. DOI: 10.1145/1314333.1314336. URL: http://systems.cs.colorado.edu/~bauerk/papers/wpes25-bauer.pdf (cit. on p. 42).

[13] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/rfc7540. URL: https://rfc-editor.org/rfc/rfc7540.txt (cit. on p. 100).

[14] Alex Biryukov, Ivan Pustogarov, and Ralf Philipp Weinmann. “TorScan: Tracing Long-lived Connections and Differential Scanning Attacks”. In: *Proceedings of the European Symposium Research Computer Security - ESORICS'12*. Springer, Sept. 2012. URL: http://freehaven.net/anonbib/papers/torscan-esorics2012.pdf (cit. on p. 42).

[15] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. “Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. May 2013. URL: http://www.ieee-security.org/TC/SP2013/papers/4977a080.pdf (cit. on p. 42).

[16] Manuel Blum and Silvio Micali. “How to generate cryptographically strong sequences of pseudorandom bits”. In: *SIAM journal on Computing* 13.4 (1984), pp. 850–864. DOI: 10.1137/0213053. URL: http://dx.doi.org/10.1137/0213053 (cit. on p. 97).

[17] Floor Boon, Steven Derix, and Huib Modderkolk. *Document Snowden: Nederland al sinds 1946 doelwit van NSA*. Newspaper. Nov. 2013. URL: https://www.nrc.nl/nieuws/2013/11/23/nederland-sinds-1946-doelwit-van-nsa-a1429490 (cit. on pp. 3, 4).

[18] Carsten Bormann, Klaus Hartke, and Zach Shelby. *RFC7252: The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/rfc7252. URL: https://rfc-editor.org/rfc/rfc7252.txt (cit. on p. 102).

[19] S. Bradner. *RFC2119 Key words for use in RFCs to Indicate Requirement Levels*. IETF, 1997. URL: http://tools.ietf.org/pdf/rfc2119.pdf (cit. on p. 121).

[20] *Has F5 Really Been Broken*. Department of Computing, University of Surrey, Guildford GU2 7XH, England. IET, 2009. DOI: 10.1049/ic.2009.0245. URL: https://pdfs.semanticscholar.org/4d6c/d9d7e3a419ea74a4a363a36fcc674e89ecc7.pdf (cit. on pp. 25, 75, 112).

[21] BSI. *Migration zu Post-Quanten-Kryptografie*. Bundesamt für Sicherheit in der Informationstechnik. Mar. 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Krypto/Post-Quanten-Kryptografie.pdf?__blob=publicationFile&v=2 (cit. on pp. 62, 66).

[22] Travis Burtrum. *XEP-0418: DNS Queries over XMPP (DoX)*. Mar. 2019. URL: https://xmpp.org/extensions/xep-0418.pdf (cit. on p. 26).

[23] *Campaign Monitor*. 2012. URL: http://www.campaignmonitor.com/resources/will-it-work/email-clients/ (cit. on p. 34).

[24] David Chaum. “Untraceable Electronic Mail, Return, Addresses, and Digital Pseudonyms”. In: *Communications of the ACM* (1981). URL: http://www.cs.utexas.edu/~shmat/courses/cs395t_fall04/chaum81.pdf (cit. on pp. 15, 36, 41, 42).

[25] David Chaum. “The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability”. In: *Journal of Cryptology* 1 (1988), pp. 65–75. URL: http://www.cs.ucsb.edu/~ravenben/classes/595n-s07/papers/dcnet-jcrypt88.pdf (cit. on pp. 39, 48).

[26] David Chaum, Claude Crépeau, and Ivan Damgard. “Multiparty unconditionally secure protocols”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 11–19. URL: http://crypto.cs.mcgill.ca/~crepeau/PDF/ASPUBLISHED/CCD88A.pdf (cit. on p. 66).

[27] Chen Chen and Adrian Perrig. “Phi: Path-hidden lightweight anonymity protocol at network layer”. In: *Proceedings on Privacy Enhancing Technologies* 2017.1 (2017), pp. 100–117. DOI: 10.1515/popets-2017-0007. URL: https://www.degruyter.com/downloadpdf/j/popets.2017.2017.issue-1/popets-2017-0007/popets-2017-0007.pdf (cit. on p. 47).

[28] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. “Freenet: A Distributed Anonymous Information Storage and Retrieval System”. In: *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. July 2000, pp. 46–66. URL: https://freenetproject.org/ (cit. on pp. 39, 48).

[29] *Commercial National Security Algorithm Suite*. URL: https://apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm (cit. on p. 19).

[30] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. “Riposte: An anonymous messaging system handling millions of users”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 321–338. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7163034 (cit. on p. 49).

[31] Henry Corrigan-Gibbs and Bryan Ford. “Dissent: Accountable Anonymous Group Messaging”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. Chicago, Illinois, USA: ACM, 2010, pp. 340–350. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866346. URL: http://doi.acm.org/10.1145/1866307.1866346 (cit. on pp. 40, 48).

[32] M. Crispin. *RFC3501 INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. IETF, 2003. URL: http://tools.ietf.org/pdf/rfc3501.pdf (cit. on p. A66).

[33] George Danezis, Claudia Diaz, Emilia Käsper, and Carmela Troncoso. “The Wisdom of Crowds: Attacks and Optimal Constructions”. In: *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 2009), Saint-Malo, France*. Ed. by Michael Backes and Peng Ning. Vol. 5789. Lecture Notes in Computer Science. Springer, Sept. 2009, pp. 406–423. ISBN: 978-3-642-04443-4. URL: http://homes.esat.kuleuven.be/~ekasper/papers/crowds.pdf (cit. on p. 39).

[34] Norman Danner, Sam DeFabbia-Kane, Danny Krizanc, and Marc Liberatore. “Effectiveness and detection of denial of service attacks in Tor”. In: *Transactions on Information and System Security* 15.3 (2012), 11:1–11:25. DOI: 10.1145/2382448.2382449. URL: http://arxiv.org/pdf/1110.5395v3.pdf (cit. on p. 42).

[35] Roger Dingledine and Nick Mathewson. *Tor Protocol Specification*. URL: https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt (cit. on p. 42).

[36] ISO DIS. “8824: Specification of Abstract Syntax Notation One (ASN. I)”. In: *Basic Encoding Rules* (2015) (cit. on p. 90).

[37] Viktor Dukhovni and Wes Hardaker. *RFC7672: SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS)*. RFC 7672. Oct. 2015. DOI: 10.17487/RFC7672. URL: https://rfc-editor.org/rfc/rfc7672.txt (cit. on p. 123).

[38] Lisa M. Dusseault. *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. RFC 4918. June 2007. DOI: 10.17487/RFC4918. URL: https://rfc-editor.org/rfc/rfc4918.txt (cit. on p. 100).

[39] Morris Dworkin. *Recommendation for block cipher modes of operation. methods and techniques*. Tech. rep. DTIC Document, 2001 (cit. on pp. 21, 93).

[40] André Egners, Dominic Gatzen, Andriy Panchenko, and Ulrike Meyer. “Introducing SOR: SSH-based onion routing”. In: *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*. IEEE. IEEE, Mar. 2012, pp. 280–286. DOI: 10.1109/waina.2012.89. URL: https://www.researchgate.net/profile/Andre_Egners/publication/237007773_Introducing_SOR_SSH-based_onion_routing/links/548805e90cf2ef34478ed724/Introducing-SOR-SSH-based-onion-routing.pdf (cit. on p. 46).

[41] M. Elkins. *RFC2015 MIME Security with Pretty Good Privacy (PGP)*. IETF, 1996. URL: http://tools.ietf.org/pdf/rfc2015.pdf (cit. on pp. 4, 68).

[42] *Email Client Market Share*. 2014. URL: http://emailclientmarketshare.com/ (cit. on p. 34).

[43] Nathan Evans, Roger Dingledine, and Christian Grothoff. “A Practical Congestion Attack on Tor Using Long Paths”. In: *Proceedings of the 18th USENIX Security Symposium*. Aug. 2009. URL: http://freehaven.net/anonbib/papers/congestion-longpaths.pdf (cit. on p. 42).

[44] *Fact Sheet Suite B Cryptography*. 2008. URL: http://www.nsa.gov/ia/industry/crypto_suite_b.cfm (cit. on p. 19).

[45] C Feather. *Network News Transfer Protocol (NNTP)*. IETF. Oct. 2006. URL: https://www.rfc-editor.org/rfc/pdfrfc/rfc3977.txt.pdf (cit. on p. 99).

[46] Hannes Federrath. “Das AN.ON-System: Starke Anonymität und Unbeobachtbarkeit im Internet”. In: *Anonymität im Internet*. Springer, 2003, pp. 172–178 (cit. on p. 45).

[47] Paul Feldman. “A practical scheme for non-interactive verifiable secret sharing”. In: *Foundations of Computer Science, 1987., 28th Annual Symposium on*. IEEE. 1987, pp. 427–438. URL: https://www.cs.umd.edu/~gasarch/TOPICS/secretsharing/feldmanVSS.pdf (cit. on p. 19).

[48] Jessica Fidrich, Miroslav Goljan, and Dorin Hogea. “Steganalysis of JPEG Images: Breaking the F5 Algorithm”. In: *International Workshop on Information Hiding*. Springer. 2002, pp. 310–323. URL: http://www.ws.binghamton.edu/fridrich/research/f5.pdf (cit. on p. 111).

[49] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. *OpenPGP Message Format*. RFC 4880. Nov. 2007. DOI: 10.17487/RFC4880. URL: https://rfc-editor.org/rfc/rfc4880.txt (cit. on p. 35).

[50] N. Freed and N. Borenstein. *RFC2045 Multipurpose Internet Mail Extensions; (MIME) Part One: Format of Internet Message Bodies*. IETF, 1996. URL: http://tools.ietf.org/pdf/rfc2045.pdf (cit. on pp. 4, 103).

[51] N. Freed and N. Borenstein. *RFC2046 Multipurpose Internet Mail Extensions; (MIME) Part Two: Media Types*. IETF, 1996. URL: http://tools.ietf.org/pdf/rfc2046.pdf (cit. on p. 103).

[52] Michael J. Freedman and Robert Morris. “Tarzan: A Peer-to-Peer Anonymizing Network Layer”. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*. Washington, DC, Nov. 2002. URL: http://pdos.lcs.mit.edu/tarzan/docs/tarzan-ccs02.pdf (cit. on p. 47).

[53] *FREEDOM ON THE NET 2018*. Oct. 2018 (cit. on p. 4).

[54] Jessica Fridrich, Tomav Pevny, and Jan Kodovsky. “Statistically undetectable jpeg steganography: dead ends challenges, and opportunities”. In: *Proceedings of the 9th workshop on Multimedia & security*. 2007, pp. 3–14. URL: https://dl.acm.org/doi/pdf/10.1145/1288869.1288872?casa_token=XajiDUEdBf4AAAAA:Bv1tEz785TUYLzJaCM_T-1suexnqfktng90KxZeindGFUow_rrBe5TDOe0CfuiDkc4-127FJa6F- (cit. on pp. 112, 157).

[55] Simson Garfinkel. *PGP: Pretty Good Privacy*. Encryption for everyone. O’Reilly/International Thomson Verlag, 1996. ISBN: 3-930673-30-4 (cit. on p. 68).

[56] R. Gellens and J. Klensin. *RFC4409 Message Submission for Mail*. IETF, 2006. URL: http://tools.ietf.org/pdf/rfc4409.pdf (cit. on p. 33).

[57] Sharad Goel, Mark Robson, Milo Polte, and Emin Gun Sirer. *Herbivore: A Scalable and Efficient Protocol for Anonymous Communication*. Tech. rep. 2003-1890. Ithaca, NY: Cornell University, Feb. 2003. URL: http://www.cs.cornell.edu/People/egs/papers/herbivore-tr.pdf (cit. on pp. 40, 48).

[58] Philippe Golle and Ari Juels. “Dining Cryptographers Revisited”. In: *Proceedings of Eurocrypt 2004*. May 2004. URL: http://crypto.stanford.edu/~pgolle/papers/nim.pdf (cit. on p. 40).

[59] Andy Greenberg. *Leaked NSA Doc Says It Can Collect And Keep Your Encrypted Data As Long As It Takes To Crack It*. June 2013. URL: https://www.forbes.com/sites/andygreenberg/2013/06/20/leaked-nsa-doc-says-it-can-collect-and-keep-your-encrypted-data-as-long-as-it-takes-to-crack-it/#5edf34edb07d (cit. on pp. 3, 4).

[60] Ceki Gülcü and Gene Tsudik. “Mixing E-mail With Babel”. In: *Proceedings of the Network and Distributed Security Symposium - NDSS ’96*. IEEE, Feb. 1996, pp. 2–16. URL: http://citeseer.nj.nec.com/2254.html (cit. on p. 41).

[61] Shai Halevi and Phillip Rogaway. “A Tweakable Enciphering Mode”. In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 482–499. ISBN: 978-3-540-45146-4. DOI: 10.1007/978-3-540-45146-4_28 (cit. on pp. 22, 95).

[62] Michael Herrmann and Christian Grothoff. “Privacy Implications of Performance-Based Peer Selection by Onion Routers: A Real-World Case Study using I2P”. In: *Proceedings of the 11th Privacy Enhancing Technologies Symposium (PETS 2011)*. Waterloo, Canada, July 2011. URL: http://freehaven.net/anonbib/papers/pets2011/p9-herrmann.pdf (cit. on p. 44).

[63] P. Hoffman and P. McManus. *DNS Queries over HTTPS (DoH)*. IETF. Oct. 2018. URL: https://tools.ietf.org/html/rfc8484 (cit. on p. 26).

[64] Paul E. Hoffman and Jakob Schlyter. *RFC6698: The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698. Aug. 2012. DOI: 10.17487/RFC6698. URL: https://rfc-editor.org/rfc/rfc6698.txt (cit. on p. 123).

[65] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. “NTRU: A ring-based public key cryptosystem”. In: *International Algorithmic Number Theory Symposium*. More information about recent research at https://ntru.com. Springer. 1998, pp. 267–288. URL: https://assets.onboardsecurity.com/static/downloads/NTRU/resources/ANTS97.pdf (cit. on p. 18).

[66] Vojtěch Holub, Jessica Fridrich, and Tomáš Denemark. “Universal distortion function for steganography in an arbitrary domain”. In: *EURASIP Journal on Information Security* 2014.1 (2014), p. 1 (cit. on p. 157).

[67] Morteza Darvish Morshedi Hosseini and Mojtaba Mahdavi. “Modification in spatial, extraction from transform: A new approach for JPEG steganography”. In: *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*. IEEE. 2015, pp. 134–140 (cit. on p. 112).

[68] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. “The Parrot is Dead: Observing Unobservable Network Communications”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. May 2013. URL: http://www.cs.utexas.edu/~amir/papers/parrot.pdf (cit. on pp. 16, 74, 75, 156).

[69] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. “Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries”. In: *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS 2013)*. Nov. 2013. URL: http://www.ohmygodel.com/publications/usersrouted-ccs13.pdf (cit. on p. 43).

[70] Justin Karneges and Peter Saint-Andre. *XEP-0047: In-band bytestreams*. 2003. URL: https://xmpp.org/extensions/xep-0047.html (cit. on p. 107).

[71] S. Kaur, S. Bansal, and R. K. Bansal. “Steganography and classification of image steganography techniques”. In: *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*. Mar. 2014, pp. 870–875. DOI: 10.1109/IndiaCom.2014.6828087. URL: https://ieeexplore.ieee.org/abstract/document/6828087 (cit. on p. 25).

[72] Juhoon Kim, Fabian Schneider, Bernhard Ager, and Anja Feldmann. “Today’s usenet usage: NNTP traffic characterization”. In: *2010 INFOCOM IEEE Conference on Computer Communications Workshops*. IEEE. 2010, pp. 1–6. DOI: 10.1109/INFCOMW.2010.5466665. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5466665 (cit. on p. 99).

[73] J. Klensin. *RFC5321 Simple Mail Transfer Protocol*. IETF, 2008. URL: http://tools.ietf.org/pdf/rfc5321.pdf (cit. on pp. 31, 103, 122, A68).

[74] Neal Koblitz, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. 2004. URL: http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf (cit. on p. 18).

[75] Jan Kodovsky, Tomas Pevny, and Jessica Fridrich. “Modern steganalysis can detect YASS”. In: *Media Forensics and Security II*. Vol. 7541. International Society for Optics and Photonics. 2010, p. 754102. URL: http://ia.binghamton.edu/publication/FridrichPDF/yass_attack.pdf (cit. on p. 25).

[76] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. “Atom: Scalable anonymity resistant to traffic analysis”. In: *CoRR abs/1612.07841* (2016). URL: https://www.researchgate.net/profile/Bryan_Ford/publication/311900860_Atom_Scalable_Anonymity_Resistant_to_Traffic_Analysis/links/58c918ddaca2723ab18138c5/Atom-Scalable-Anonymity-Resistant-to-Traffic-Analysis.pdf (cit. on p. 46).

[77] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. “Riffle: An efficient communication system with strong anonymity”. In: *Proceedings on Privacy Enhancing Technologies* 2016.2 (2016), pp. 115–134. URL: https://content.sciendo.com/downloadpdf/journals/popets/2016/2/article-p115.pdf (cit. on p. 46).

[78] Butler W. Lampson. *A Note on the Confinement Problem*. 1973. URL: http://faculty.kfupm.edu.sa/COE/mimam/Papers/73%20A%20Note%20on%20the%20Confinement%20Problem.pdf (cit. on p. 25).

[79] David Lazar, Yossi Gilad, and Nickolai Zeldovich. “Karaoke: Distributed private messaging immune to passive traffic analysis”. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 711–725. URL: https://www.usenix.org/system/files/osdi18-lazar.pdf (cit. on p. 50).

[80] David Lazar and Nickolai Zeldovich. “Alpenhorn: Bootstrapping secure communication without leaking metadata”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 571–586. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-lazar.pdf (cit. on p. 51).

[81] Arjen K. Lenstra. *key length – contribution to the handbook of information security*. 2004. URL: https://infoscience.epfl.ch/record/164539/files/NPDF-32.pdf (cit. on pp. 19, 20).

[82] Moritz Leuenberger and Josi Meier. *Vorkommnisse im EJPD Bericht der Parlamentarischen Untersuchungskommission(PUK)*. Bundesblatt 1989-55. Nov. 1989. URL: https://www.parlament.ch/centers/documents/de/ed-berichte-puk-ejpd.pdf (cit. on p. 3).

[83] Brian Neil Levine and Clay Shields. "Hordes — A Multicast Based Protocol for Anonymity". In: *Journal of Computer Security* 10.3 (2002), pp. 213–240. URL: http://prisms.cs.umass.edu/brian/pubs/brian.hordes.jcs01.pdf (cit. on p. 48).

[84] Peter H. Lewis. *Behind an Internet Message Service's Closure*. New York Times. Sept. 1996. URL: https://www.nytimes.com/1996/09/06/business/behind-an-internet-message-service-s-close.html (cit. on pp. 41, 60).

[85] Bin Li, Jiwu Huang, and Yun Qing Shi. "Steganalysis of YASS". In: *IEEE Transactions on Information Forensics and Security* 4.3 (2009), pp. 369–382. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5153278 (cit. on p. 25).

[86] Bin Li, Ming Wang, Jiwu Huang, and Xiaolong Li. "A new cost function for spatial image steganography". In: *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2014, pp. 4206–4210 (cit. on p. 157).

[87] Helger Lipmaa, Phillip Rogaway, and David Wagner. "CTR-mode encryption". In: *First NIST Workshop on Modes of Operation*. 2000 (cit. on pp. 21, 94).

[88] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. *XEP-0166: Jingle*. 2009. URL: https://xmpp.org/extensions/xep-0166.html (cit. on p. 107).

[89] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. "l-diversity: Privacy beyond k-anonymity". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1.1 (2007), p. 3. URL: http://www.cs.cornell.edu/~vmuthu/research/ldiversity.pdf (cit. on p. 14).

[90] George Marsaglia et al. "Xorshift rngs". In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6 (cit. on p. 97).

[91] Luther Martin. "XTS: A Mode of AES for Encrypting Hard Disks". In: *IEEE Security & Privacy Magazine* 8.3 (May 2010), pp. 68–69. DOI: 10.1109/msp.2010.111. URL: https://ieeexplore.ieee.org/iel5/8013/5470945/05470958.pdf (cit. on pp. 22, 95).

[92] Larry Masinter. *RFC2397 The" data" URL scheme*. 1998. URL: https://www.rfc-editor.org/info/rfc2397 (cit. on p. 107).

[93] Mitsuru Matsui, S Moriai, and J Nakajima. "RFC3713: A Description of the Camellia Encryption Algorithm". In: (2004) (cit. on p. 17).

[94] Robert J. McEliece and Dilip V. Sarwate. "On sharing secrets and Reed-Solomon codes". In: *Communications of the ACM* 24.9 (1981), pp. 583–584. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.2829&rep=rep1&type=pdf (cit. on p. 81).

[95] David McGrew and John Viega. “The Galois/counter mode of operation (GCM)”. In: *Submission to NIST. http://csrc. nist. gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec. pdf* (2004) (cit. on pp. 22, 94).

[96] David A McGrew and John Viega. “The security and performance of the Galois/-Counter Mode (GCM) of operation”. In: *International Conference on Cryptology in India*. Springer. 2004, pp. 343–355 (cit. on pp. 22, 94).

[97] *Messaging Application Programming Interface (MAPI) Extensions for HTTP*. Microsoft Corporation, 2018. URL: https://interoperability.blob.core.windows.net/files/MS-OXCMAPIHTTP/[MS-OXCMAPIHTTP].pdf (cit. on p. A66).

[98] Victor S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology — CRYPTO ’85 Proceedings*. Ed. by Hugh C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426. ISBN: 978-3-540-39799-1. DOI: 10.1007/3-540-39799-X_31. URL: http://dx.doi.org/10.1007/3-540-39799-X_31 (cit. on p. 18).

[99] Kazuhiko Minematsu, Stefan Lucks, Hiraku Morita, and Tetsu Iwata. “Attacks and security proofs of EAX-prime”. In: *International Workshop on Fast Software Encryption*. Springer. 2013, pp. 327–347. URL: http://eprint.iacr.org/2012/018.pdf (cit. on pp. 21, 93).

[100] Alan Mislove, Gaurav Oberoi, Ansley Post, Charles Reis, Peter Druschel, and Dan S Wallach. “AP3: Cooperative, decentralized anonymous communication”. In: *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM. 2004, p. 30. URL: http://www-dev.ccs.neu.edu/home/amislove/publications/AP3-SIGOPSEW.pdf (cit. on p. 45).

[101] Prateek Mittal and Nikita Borisov. “Information Leaks in Structured Peer-to-peer Anonymous Communication Systems”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*. Ed. by Paul Syverson, Somesh Jha, and Xiaolan Zhang. Alexandria, Virginia, USA: ACM Press, Oct. 2008, pp. 267–278. URL: http://www.hatswitch.org/~nikita/papers/information-leak.pdf (cit. on pp. 45, 47).

[102] Jerey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. *RFC2616: Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/rfc2616. URL: https://rfc-editor.org/rfc/rfc2616.txt (cit. on p. 100).

[103] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. “Skypemorph: Protocol obfuscation for tor bridges”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. ACM Press, 2012, pp. 97–108. DOI: 10.1145/2382196.2382210. URL: http://www.cypherpunks.ca/~iang/pubs/skypemorph-ccs.pdf (cit. on p. 64).

[104] J. P. Muñoz-Gea, J. Malgosa-Sanahuja, P. Manzanares-Lopez, J. C. Sanchez-Aarnoutse, and J. Garcia-Haro. “A Low-Variance Random-Walk Procedure to Provide Anonymity in Overlay Networks”. In: *Computer Security - ESORICS 2008*. Springer Berlin Heidelberg, 2008, pp. 238–250. ISBN: 978-3-540-88313-5. DOI: 10.1007/978-3-540-88313-5_16 (cit. on p. 39).

[105] John G. Myers. *Local Mail Transfer Protocol*. RFC 2033. Oct. 1996. DOI: 10.17487/RFC2033. URL: https://rfc-editor.org/rfc/rfc2033.txt (cit. on p. A66).

[106] Arjun Nambiar and Matthew Wright. "Salsa: A Structured Approach to Large-Scale Anonymity". In: *Proceedings of CCS 2006*. Nov. 2006. URL: http://ranger.uta.edu/~mwright/papers/salsa-ccs06.pdf (cit. on p. 47).

[107] Alan Nicholson, Jan-Dirk Schmöcker, Michael Bell, and Yasunori Iida. "Assessing transport reliability: malevolence and user knowledge". In: *The network reliability of transport: Proceedings of the 1st international symposium on transportation network reliability (INSTR)*. Emerald Group Publishing Limited. 2003, pp. 1–22. DOI: 10.1108/9781786359544-001. URL: https://www.emerald.com/insight/content/doi/10.1108/9781786359544-001/full/html (cit. on p. 62).

[108] Suresh Venkatasubramanian Ninghui Li Tiancheng Li. "t-Closeness: Privacy Beyond k-Anonymity and l-Diversity". In: (). URL: http://www.cs.purdue.edu/homes/li83/papers/icde_closeness.pdf (cit. on p. 15).

[109] NSA. *XKeyscore presentation from 2008*. Web and several newspapers (e.g., guardian). Three slides have been redacted as they contained suposedly specific NSA operations. July 2013. URL: https://www.theguardian.com/world/interactive/2013/jul/31/nsa-xkeyscore-program-full-presentation (cit. on pp. 3, 4).

[110] Lasse Øverlier and Paul Syverson. "Locating Hidden Servers". In: *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE CS, May 2006. URL: http://tor-svn.freehaven.net/anonbib/cache/hs-attack06.pdf (cit. on p. 42).

[111] Adrian Perrig, Pawel Szalachowski, Raphael M Reischuk, and Laurent Chuat. *SCION: a secure Internet architecture*. Springer, 2017. URL: https://www.scion-architecture.net/pdf/SCION-book.pdf (cit. on p. 47).

[112] Saint-Andre Peter and Kaes Craig. *XEP-0013: Flexible Offline Message Retrieval*. 2005. URL: http://xmpp.org/extensions/xep-0013.html (cit. on p. 103).

[113] Tomav Pevny, Tomav Filler, and Patrick Bas. "Using high-dimensional image models to perform highly undetectable steganography". In: *International Workshop on Information Hiding*. Springer. 2010, pp. 161–177 (cit. on p. 157).

[114] Andreas Pfitzmann and Marit Hansen. *A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management*. V0.34. Aug. 2010. URL: http://dud.inf.tu-dresden.de/literatur/Anon%5C_Terminology%5C_v0.34.pdf (cit. on pp. 13, 60, A65, A66, A68, A69).

[115] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. "The loopix anonymity system". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 1199–1216. URL: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-piotrowska.pdf (cit. on p. 50).

[116] Jon Postel and Joyce Reynolds. *RFC 959: File transfer protocol*. 1985. URL: https://tools.ietf.org/pdf/rfc959.pdf (cit. on p. 100).

[117] Tom Postmes, Russell Spears, Khaled Sakhel, and Daphne De Groot. “Social influence in computer-mediated communication: The effects of anonymity on group behavior”. In: *Personality and Social Psychology Bulletin* 27.10 (2001), pp. 1243–1254. DOI: 10.1177/0146167201271001. URL: http://psp.sagepub.com/content/27/10/1243.short (cit. on p. 186).

[118] B. Ramsdell. *RFC3851 Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification*. IETF, 2004. URL: http://tools.ietf.org/pdf/rfc3851.pdf (cit. on p. 35).

[119] Irving S Reed and Gustave Solomon. “Polynomial codes over certain finite fields”. In: *Journal of the society for industrial and applied mathematics* 8.2 (June 1960), pp. 300–304. DOI: 10.1137/0108018. URL: https://faculty.math.illinois.edu/~duursma/CT/RS-1960.pdf (cit. on p. 66).

[120] Michael Reiter and Aviel Rubin. “Crowds: Anonymity for Web Transactions”. In: *ACM Transactions on Information and System Security* 1.1 (June 1998). URL: http://avirubin.com/crowds.pdf (cit. on p. 39).

[121] Marc Rennhard and Bernhard Plattner. “Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection”. In: *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2002)*. Washington, DC, USA, Nov. 2002. URL: http://cecid.sourceforge.net/morphmix.pdf (cit. on p. 47).

[122] Marc Rennhard and Bernhard Plattner. “Practical Anonymity for the Masses with Mix-Networks”. In: *Proceedings of the IEEE 8th Intl. Workshop on Enterprise Security (WET ICE 2003)*. Linz, Austria, June 2003. URL: https://gnunet.org/sites/default/files/RP03-1.pdf (cit. on p. 36).

[123] P. Resnick. *RFC5322 Internet Message Format*. IETF, 2008. URL: http://tools.ietf.org/pdf/rfc5322.pdf (cit. on pp. 103, 122).

[124] R. L. Rivest, A. Shamir, and L. Adleman. “a method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342 (cit. on p. 18).

[125] Phillip Rogaway, Mihir Bellare, and John Black. “OCB: A block-cipher mode of operation for efficient authenticated encryption”. In: *ACM Transactions on Information and System Security (TISSEC)* 6.3 (2003), pp. 365–403 (cit. on pp. 21, 94).

[126] Phillip Rogaway and Ted Krovetz. *The OCB Authenticated-Encryption Algorithm*. Internet-Draft draft-krovetz-ocb-04. Work in Progress. Internet Engineering Task Force, July 2012. URL: https://tools.ietf.org/html/draft-krovetz-ocb-04 (cit. on pp. 21, 94).

[127] J. P. Sain-Andre. *RFC3923: End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*. IETF, 2004. URL: http://tools.ietf.org/pdf/rfc3923.pdf (cit. on p. 103).

[128] P. Saint-Andre. *RFC3922: Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM)*. IETF, 2004. URL: http://tools.ietf.org/pdf/rfc3922.pdf (cit. on p. 103).

[129] P. Saint-Andre. *RFC6120: Extensible Messaging and Presence Protocol (XMPP): Core*. IETF, 2011. URL: http://tools.ietf.org/pdf/rfc6120.pdf (cit. on pp. 103, 107, 122, A70).

[130] P. Saint-Andre. *RFC6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. IETF, 2011. URL: http://tools.ietf.org/pdf/rfc6121.pdf (cit. on pp. 103, A70).

[131] Peter Saint-Andre. *XEP-0135: File Sharing*. 2004. URL: https://xmpp.org/extensions/xep-0135.html (cit. on p. 107).

[132] Peter Saint-Andre. *XEP-0066: Out of Band Data*. 2006. URL: https://xmpp.org/extensions/xep-0066.html (cit. on p. 107).

[133] Peter Saint-Andre and P Simerda. *XEP-0231: Bits of binary*. 2008. URL: https://xmpp.org/extensions/xep-0231.html (cit. on p. 107).

[134] Peter Saint-Andre and Lance Stout. *XEP-0234: Jingle File Transfer*. 2011. URL: https://xmpp.org/extensions/xep-0234.pdf (cit. on p. 107).

[135] Saad Saleh, Junaid Qadir, and Muhammad U Ilyas. "Shedding Light on the Dark Corners of the Internet: A Survey of Tor Research". In: *Journal of Network and Computer Applications* 114 (July 2018), pp. 1–28. DOI: 10.1016/j.jnca.2018.04.002. URL: https://www.sciencedirect.com/science/article/pii/S1084804518301280 (cit. on p. 43).

[136] Jody Sankey and Matthew Wright. "Dovetail: Stronger anonymity in next-generation internet routing". In: *International Symposium on Privacy Enhancing Technologies Symposium*. Springer. 2014, pp. 283–303. URL: https://arxiv.org/pdf/1405.0351.pdf (cit. on p. 47).

[137] Len Sassaman, Bram Cohen, and Nick Mathewson. "The Pynchon Gate". In: (). URL: https://www.esat.kuleuven.be/cosic/publications/article-620.pdf (cit. on p. 44).

[138] Vahid Sedighi, Rémi Cogranne, and Jessica Fridrich. "Content-adaptive steganography by minimizing statistical detectability". In: *IEEE Transactions on Information Forensics and Security* 11.2 (2015), pp. 221–234 (cit. on p. 157).

[139] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613. URL: https://cs.jhu.edu/~sdoshi/crypto/papers/shamirturing.pdf (cit. on pp. 35, 36, 81).

[140] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. "P5: A Protocol for Scalable Anonymous Communication". In: *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. May 2002. URL: http://www.cs.umd.edu/projects/p5/p5.pdf (cit. on p. 45).

[141] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Diaz. "A Survey on Routing in Anonymous Communication Protocols". In: *ACM Computing Surveys* 51.3 (June 2018), pp. 1–39. DOI: 10.1145/3182658. URL: https://dl.acm.org/citation.cfm?id=3182658 (cit. on p. 40).

[142] Peter W. Shor. "Polynomial-Time Algorithms For Prime Factorization And Discrete Logarithms On A Quantum Computer". In: *SIAM Journal on Computing* 26 (1997), pp. 1484–1509. URL: https://arxiv.org/pdf/quant-ph/9508027v2 (cit. on p. 18).

[143] Szusanne Sluizer and jonathan B. Postel. *Mail Transfer Protocol*. IETF. May 1981. DOI: 10.17487/RFC0780. URL: https://www.rfc-editor.org/rfc/pdfrfc/rfc780.txt.pdf (cit. on p. 99).

[144] Kaushal Solanki, Anindya Sarkar, and BS Manjunath. "YASS: Yet another steganographic scheme that resists blind steganalysis". In: *International Workshop on Information Hiding*. Springer. 2007, pp. 16–31. URL: http://vision.ece.ucsb.edu/sites/vision.ece.ucsb.edu/files/publications/kaushal_2007_IWIH.pdf (cit. on p. 25).

[145] NIST-FIPS Standard. "Announcing the advanced encryption standard (AES)". In: *Federal Information Processing Standards Publication* 197 (2001), pp. 1–51 (cit. on p. 17).

[146] A. Stevenson. *Oxford Dictionary of English*. Oxford reference online premium. OUP Oxford, 2010. ISBN: 9780199571123. URL: http://www.oed.com (cit. on p. A68).

[147] Almon Brown Strowger. *Automatic Telephone-Exchange*. en. Mar. 1891 (cit. on pp. 3, 35).

[148] Mansi S Subhedar and Vijay H Mankar. "Current status and key issues in image steganography: A survey". In: *Computer science review* 13 (2014), pp. 95–113 (cit. on p. 25).

[149] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. "Towards an Analysis of Onion Routing Security". In: *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. Ed. by H. Federrath. Springer-Verlag, LNCS 2009, July 2000, pp. 96–114. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a465255.pdf (cit. on p. 42).

[150] Parisa Tabriz and Nikita Borisov. "Breaking the Collusion Detection Mechanism of MorphMix". In: *Proceedings of the Sixth Workshop on Privacy Enhancing Technologies (PET 2006)*. Ed. by George Danezis and Philippe Golle. Cambridge, UK: Springer, June 2006, pp. 368–384. URL: https://hatswitch.org/nikita/papers/pet2006-morphmix.pdf (cit. on p. 47).

[151] Biaoshuai Tao and Hongjun Wu. "Improving the biclique cryptanalysis of AES". In: *Australasian Conference on Information Security and Privacy*. Springer. 2015, pp. 39–56 (cit. on p. 17).

[152] Muldowney Thomas, Miller Mathew, Eatmon Ryan, and Saint-Andre Peter. *XEP-0096: SI File Transfer*. 2004. URL: http://xmpp.org/extensions/xep-0096.html (cit. on pp. 103, 107).

[153] Martin Tompa and Heather Woll. "How to share a secret with cheaters". In: *journal of Cryptology* 1.3 (1989), pp. 133–138 (cit. on p. 36).

[154] Florian Tschorsch and Björn Scheurmann. "How (not) to build a transport layer for anonymity overlays". In: *Proceedings of the ACM Sigmetrics/Performance Workshop on Privacy and Anonymity for the Digital Economy*. June 2012. URL: http://pade12.mytestbed.net/pade12-final6.pdf (cit. on pp. 22, 95).

[155] Alan M. Turing. "Computing machinery and intelligence". In: *Parsing the Turing Test*. Springer, 1950, pp. 23–65 (cit. on p. 185).

[156] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. “Stadium: A distributed metadata-private messaging system”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 423–440. URL: https://dl.acm.org/doi/pdf/10.1145/3132747.3132783 (cit. on p. 50).

[157] UNHR. *International Covenant on Civil and Political Rights*. 1966. URL: http://www.ohchr.org/en/professionalinterest/pages/ccpr.aspx (cit. on p. 5).

[158] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. “Vuvuzela: Scalable private messaging resistant to traffic analysis”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 137–152. URL: https://dl.acm.org/doi/pdf/10.1145/2815400.2815417 (cit. on p. 50).

[159] Michael Waidner and Birgit Pfitzmann. “The dining cryptographers in the disco: Unconditional Sender and Recipient Untraceability”. In: *Proceedings of EUROCRYPT 1989*. Springer-Verlag, LNCS 434, 1990. URL: http://www.semper.org/sirene/publ/WaPf1_89DiscoEngl.ps.gz (cit. on p. 40).

[160] Andreas Westfeld. “F5 - A Steganographic Algorithm”. In: *none* none (2002). URL: http://www.ws.binghamton.edu/fridrich/research/f5.pdf (cit. on pp. 25, 74, 75, 111, 112).

[161] Doug Whiting, Niels Ferguson, and Russell Housley. “RFC3610: Counter with cbc-mac (ccm)”. In: (2003) (cit. on pp. 21, 94).

[162] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. “Dissent in numbers: Making strong anonymity scale”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 179–182. URL: http://dedis.cs.yale.edu/dissent/papers/osdi12.pdf (cit. on p. 48).

[163] Tao Zhou and Yaobin Lu. “Examining mobile instant messaging user loyalty from the perspectives of network externalities and flow experience”. In: *Computers in Human Behavior* 27.2 (2011), pp. 883–889. DOI: 10.1016/j.chb.2010.11.013 (cit. on pp. 62, 98).

[164] Li Zhuang, Feng Zhou, Ben Y Zhao, and Antony Rowstron. “Cashmere: Resilient anonymous routing”. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 301–314. URL: https://www.usenix.org/legacy/publications/library/proceedings/nsdi05/tech/full_papers/zhuang/zhuang.pdf (cit. on p. 45).

B Short Biography

Martin Gwerder was born 20. July 1972 in Glarus, Switzerland. He is currently a doctoral student at the University of Basel. After having concluded his studies at the polytechnic at Brugg in 1997, he did a postgraduate education as a master of business and engineering. Following that, he changed to the university track doing an MSc in Informatics at FernUniversität in Hagen. While doing this, he steadily broadened his horizon by working for industry, banking, and government as an engineer and architect in security-related positions. He currently holds a lecturer position for cloud and security at the University of Applied Sciences Northwestern Switzerland. His primary expertise is in the field of networking-related problems dealing with data protection, distribution, confidentiality, and anonymity.

,