MessageVortex

Transport Independent, Unobservable, and Unlinkable Messaging

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

2023

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
Auf Antrag von

Prof. Dr. Christian F. Tschudin und Prof. Dr. Ulrich Ultes-Nitsche

Basel, der 25.5.2021 durch die Fakultätsversammlung

Prof. Dr. Marcel Mayor

Abstract

In this thesis, we introduce an unobservable message anonymization protocol named MessageVortex. It is based on the zero-trust principle, has 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 standard transport protocols like SMTP, 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 routing or transporting nodes. We have made our prototype implementation publicly available and added an RFC-style document that contains all necessary information to build a MessageVortex node, see https://messagevortex.net/.

Acknowledgments

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

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

Dr. Andreas Hueni was very supportive by challenging my work with his outside-the-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 LaTeX project for free. Due to their efforts, we can generate professionally typeset PDFs (and far more) for free.

Contents

List of Tables

List of Figures

List of Requirements

Please run LaTeX again to populate this list!

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" [**pulseDialingPatent**] which built the basis for modern automated telephone systems. According to several sources, he was annoyed 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, now called pulse dialing, became the standard dialing technology for more than 70 years until it was replaced by tone dialing.

This dialing technology was the basis for automatic messaging of voice and text messages (e.g., telex) and is the foundation for current routed networks. These networks established our communication-based society and allow us to quickly connect with any person or company we wish. However, computers do not only allow to route at high speed and throughput. They also allow the collection and analysis of data. Today, we use these networks as communication means for all purposes, and most people spend minimal thought on the possible consequences, should the wrong person get their hands on this communication.

Information data miners may use this collected data to judge our intentions, which are confidential if we have something to hide. This problem has dramatically increased in the last years as large companies and countries started to collect all sorts of data and created the means to process them. It supposedly allows judging people not only on what they are doing but also on what they already have done in the past and what they might do in the future. Past and present, numerous events show that actors, some state-sponsored, collected data on a broad basis within the Internet. Whether this is a problem or not is a disputable fact. However, undisputed is that such data requires careful handling, and accusations should then be based on solid facts. While people may classify personalized advertising as a legitimate use of data, a general classification of citizens is broadly considered unacceptable [**NCR2013**, **XKeyscore**, **Ball2013**, **Greenberg2013**, **Leuenberger1989**].

To show that this may occur even in democracies, we refer to events such as the "secret files scandal" (or "Fichenskandal") in Switzerland. From 1900 to 1990 the 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 have documented this event in depth [**Leuenberger1989**].

In 2009, whistleblower Edward Snowden leaked a vast amount of documents which suggest that such attacks on privacy common on a global scale. The documents claim that a data collection was going to be initiated in 2010. Since these documents are not publicly available, it is difficult to prove the claims based on these documents. However, a significant number of journalists from multiple countries screened these documents and claimed that the information seemed credible. According to these documents (verified by NRC), the 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 chancellor) [**NCR2013**, **XKeyscore**, **Ball2013**, **Ackerman2013**, **Greenberg2013**]. A later published shortened list of "selectors" in Germany showed 68 telephone and fax numbers targeting the German government's economy, finance, and agricultural departments. A global survey done by the freedom house [**FOTN2020**] claims a decrease in Internet freedom for the 11^{th} year in a row.

This list of events shows that big players collect and store 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 analyzes the possibility of using state-of-the-art technology to minimize a person's information footprint on the Internet.

When looking at e-voting [**haenni2008research**] compared to traditional secret voting systems, anonymity becomes crucial as the observation of voting behavior becomes an immediate threat for each identifiable voter for an opponent, as they may fear subsequent repression.

We leave a large information footprint in our daily communication. In a regular email, we disclose everything in a "postcard" to any entity on its way. Even when encrypting a message perfectly with today's technology (S/MIME [**rfc2045**] or PGP [**rfc2015**]), it still leaves at least the originating and the receiving entity disclosed, or we rely on the promises of a third-party provider that offers a proprietary solution. Even in those cases, we leak 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. In addition to the message itself, it includes all metadata and all traffic flows. Furthermore, a protocol to anonymize messages should not rely on trusting infrastructure other than infrastructure under the sending or receiving entity's control. 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, the data or the data flow. Thus, avoiding a central infrastructure is valid for minimizing the information footprint available to a single entity.

Leaving no information trail when sending information from one person to another is difficult 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. With the dawn of the Internet, the availability of information has risen to an unknown extent.

An entity in possession of such profiles may use them for many purposes. These include service adoption, directed advertising, or the classification of citizens. The examples given above show that this data's effects are not limited to the Internet but can also reach us in the real world.

The main problem with this data is that it may be collected over a considerable amount of time and evaluated at any time. It could even occur that standard practices at one time are judged differently at a later time. Governments, companies, or people could then judge others retrospectively on these types of practices. This questionable type of judgment is visible in the tax avoidance discussion [**Amat1999**].

People with a "bad", "unsuitable", or "non-conformant" information footprint may be subject to banning, repression, or information access exclusion. People must be able to control their own data footprint. Not providing those means allows any country or a more prominent player to effectively ban and control any number of persons within or outside the Internet.

1.1 Our Approach

Our approach in this work is to provide a new form of communication for such environments. Messages should be exchangeable without the knowledge of anyone including any observer on a governmental or ISP level. This unobservability must not only cover any message but all associated metadata as well. The infrastructure needed for this means of communication must be standard, off-the-shelf and unsuspicious. Communication should be secure without any or minimal trust in the infrastructure routing the messages.

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

> “ 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 March 23rd, 2021, Countries such as China (signatory), Cuba (signatory), Qatar (signatory), Saudi Arabia, Singapore, United Arab Emirates, or Myanmar have yet to ratify the ICCPR. Other countries such as the United States or Russia either put local laws in place superseding the ICCPR or made reservations rendering parts ineffective. Therefore, we may safely assume that freedom of speech is not given on the Internet.

If we transfer the right of free speech in the world of networks, then uncensored network packet flow is the equivalent in the networking world. Network packets may pass through any point in the world. A sender has no control over it. This lack of control occurs because 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 which way a packet has 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 the same routing decisions regardless of other properties than the source and destination address). Any output of these tools may, therefore, not be 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, during, or after sending.

As an example of the problems analyzing a packet route, we 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 false. The man page describes some of the possible causes.

We cannot state that data packets we are sending pass 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 means of transport for messages which keep sender and recipient anonymous to an adversary.

We feel that this work is needed, as much work in the anonymity field is focused on the aspect of "how to achieve anonymity" and analyzing it against the means of an adversary, which is simple and technocratically based. In this work, we define an adversary who observes or disrupts communication, but may also suppress the use of technology. Therefore, the focus is not only to create a protocol for anonymity but to create a protocol that is undetectable.

2 Our Contribution

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

The protocol employs a new type of programmable forwarders called *VortexNodes* (nodes) with a novel way of message mixing, moving away from a strictly chunked and onionized system to one, where routing operations allow an 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 the message at any common *VortexNode*.

Our protocol differentiates from other protocols by the fact that 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 a dedicated infrastructure. The carrier protocols may even be switched during routing, making it even more difficult to observe message traffic.

For non-traceable routing, 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 larger or smaller form than the original message by adding or removing redundancy operations. The applied padding structures the message so 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 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 non-neighboring nodes' messages is not possible.

An outside observer cannot identify messages, as they do not use a 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 server administrators' prosecution 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 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 can identify the steganographically hidden information, he may apply censorship but remains unable to trace messages according to external attributes, even assuming that he has additional information from collaborating nodes within the message path.

This protocol can even withstand a censoring adversary on a regional or super-regional scale, as our protocol hides in common protocols and remains undetectable. As the creator of a routing block fully controls anonymity, we achieve either sender or receiver anonymity. The protocol is built with crypto-agility and thus is able to adapt to the anonymity needs of its user.

Our protocol was implemented in Java , is publicly available under , and runs on RaspberryPI Zero W computers as a proof of concept, showing that weak nodes may participate in such a network. In addition to the scientific aspects of the protocol, we shed light on many operational aspects relevant for a real-world usage of the protocol and added these findings to the work.

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 degree of unlinkability (sender and receiver anonymity) towards an adversary with a high budget and privileged access to the 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 censoring or observing adversary?

   This question covers the principal part of the work. We first collect relevant concepts, systems and technologies in ?? and ??. We then elaborate on a list of criteria for the *MessageVortex* protocol in ??. In ??, we then create a list of suitable technologies and methods and explain our choice in ??. Based on these findings, we define a protocol combining these technologies and researches into a solution in ??. The implementation

of this solution is explained in **??** and then in **??** analyzed for suitability based on the criteria specified.

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 the protocol's resistance based on the definition of the protocol in **??**. First, we collected well-known attacks in **??**. We then elaborate if those attacks might be successful (and if so under what circumstances) in **??** and **??**.

3. How can design mitigate attacks targeting the 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 in **??**. We analyze the guidelines' effectiveness and elaborate on the general achievement level of the protocol by referring to 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. As a uniformed notation we 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 reflecting a tuple of information, it is written in boldface. To express the content of the tuple, angular brackets $\mathbf{L}\langle\mathbf{normalAddress}, \mathbf{vortexAddress}\rangle$ are used. If we want messages encrypted with multiple keys, we 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 $D^{K_a}(\mathbf{M}^{\mathbf{K_a}}) = \mathbf{M}$.

As notation for asymetric encryption we use $E^{K_a^1}(\mathbf{M})$ where 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 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 are contained in a tuple we use the notation $\mathbf{M}\langle\mathbf{t}, \mathbf{MURB}, \mathbf{serial}\rangle$ respectively if encrypted $\mathbf{M}^{\mathbf{K_a}}\langle\mathbf{t}, \mathbf{MURB}, \mathbf{serial}\rangle$.

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

In general, subscripts denote selectors to differentiate the same type's values, 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:

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

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, a "↩" is inserted to indicate that the system inserted a line break 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. Readers may click references to the glossary or the literature to find the respective entry. Chapter or table references are clickable as well.

Relevant Concepts and Technologies

***

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

In this part, we shed light on important concepts and technologies related to our work. Chapter ?? relates to some basic concepts of anonymity, such as a definition and some metrics. We furthermore introduce Zero Trust and several other concepts often used in conjunction with anonymity-related systems. Chapter ?? covers cryptographic-related research and summarizes some important facts which form the base for our future design. Lastly, ?? collects some research on the topic of censorship circumvention.

We focus on the general concepts and technologies of anonymity and elaborate on their relation to our problem.

5 Anonymity and Trust-Related Research

While there is much research on anonymity and trust, many basics remain insufficiently researched. Definitions for basic terms such as anonymity or censorship are rare. There is no common agreement for such terms. Measuring degrees of censorship or anonymity is even more challenging. We were unable to find metrics for measuring anonymity that cover all or even most aspects or enable the correct automated measurement of anonymity.

5.1 Definition of Anonymity

As the definition of anonymity, we take the definition as specified in [**anonTerminology**].

> “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.[2]”

We define the anonymity set as the set of all possible subjects within a supposed message. A subject’s anonymity towards an observing third party is crucially related to our adversary model.

Furthermore, we define that “sender anonymity” is available if a sender may send a message and the recipient cannot identify the sender in the anonymity set. Similarly, a system provides “receiver anonymity” if the sender cannot identify a message’s recipient within an anonymity set.

5.2 k-Anonymity

k-anonymity is a term introduced in [**k-anonymous:ccs2003**]. This work claims that entities are not responsible for an action if an observer cannot match a specific action to fewer than k entities. In contrast, the metric k may be dependent on the subject’s location and personal circumstances.

The paper 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 respective jurisdiction. Depending on the jurisdiction, it is usually impossible to prosecute someone if an action is not directly coupled to one person.

The problem is that under normal circumstances, k is either not constant or decreases over time. Therefore, an anonymity protocol must ensure that a sender or receiver set of k entities is either unidentifiable or has a sufficient size so that k is adequately sized even when decreasing over time.

5.3 ℓ-Diversity

In [**machanavajjhala2007diversity**] 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 additional information is 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 too 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 remains 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.

5.4 t-Closeness

While ℓ-diversity protects the identity of an entity, it does not prevent information gain. A subject in a class has the same attributes. This is where t-closeness [**li2007t**] 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.”

While in statistics working with cases and exact figures, we may, possibly, identify the distance between attributes of a class for a single set of classes reflecting a defined distribution at a given point in time. Whenever looking at a varying set of characteristics, such a metric seemed an impractical value. Therefore we discarded this value as a metric for our protocol.

5.5 Zero Knowledge Proofs

In [**goldwasser1989knowledge**] and later [**de1987non**] the authors introduce Zero-Knowledge Proofs (ZKP), which allow proving the knowledge of a secret without revealing

any detail about the secret itself. Other authors further broadened this concept by allowing proof that calculations (e.g., shuffles) have been carried out accordingly. ZKPs are powerful companions in today's anonymity systems to detect cheating nodes.

Their disadvantages are typically a high computational and bandwidth consumption for the proof and possibly a complex interaction between the prover and the verifier.

We attempted to secure the computation of our routing operations with ZKPs and failed. The operations carried out, especially calculations with S-Boxes, as within AES, and the concept of crypto-agility was too complex to be secured. Depending on the crypto-agility scheme used, the verifier would require knowledge of the operations carried out, which was not acceptable for our system. We therefore dropped the attempt to secure our operations through ZKPs.

5.6 Censorship

As a definition for censorship, we take

> “Censorship: The cyclical suppression, banning, expurgation, or editing by an individual, institution, group, or government that enforces or influences 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 also a form of censorship.

In our more technical view 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 Internet location as it is the only appropriate location for us. Furthermore, it limits the definition to the maximum reach within that system.

5.6.1 Censorship Resistance

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 a censoring entity.
  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 means of suppressing real-world entities participating in the system.

5.6.2 Parrot Circumvention

In [**oakland2013-parrot**] **oakland2013-parrot** 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 however a possibility 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 compare to a real Turing test. There remains only a "passive observer Turing test", enabling the potential nodes but not the observer to choose the messages.

In our design, this is covered by the blending layer, which generates the visible part of the message. The blending layer generates messages which contain either obviously machine-generated contextless messages or simple messages following tweet-styled patterns.

5.7 Single Use Reply Blocks and Multi-Use Reply Blocks

Chaum first introduced the use of reply blocks in [**CHAUM1**]. In general, a routing block 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 or unlimited number of times.

Our research discovered that if a routing protocol is deterministic, an adversary may use the traffic generated by a MURB to identify some of the message's properties. Depending on the type of attack, the block has to be repeated very often. For this reason, we limited the number of replays. The concept is tha in our case we have 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 large enough to support human communication sensibly and to not add too much overhead when re-requesting more MURBs. The number should not be too large because if a MURB is reused, the same traffic pattern is generated, making the system susceptible to statistical attacks.

5.8 Zero Trust

Zero trust is not an academically defined concept. It is widely misused by many marketing departments of well-known devices and applications related to security. The first citation of the idea was in [**kindervag2010no**] where **kindervag2010no** introduced this concept.

kindervag2010no compares the traditional approach as an M&M (crunchy shell and soft inner part) and introduces the zero trust principle in three concepts:

> "
>
> 1. Ensure That All Resources Are Accessed Securely Regardless Of Location
> 2. Adopt A Least Privilege Strategy and Strictly Enforce Access Control

> 3. Inspect and Log All Traffic
>
> ”

This concept applies to security and not to anonymity. We therefore had to adopt this concept, without violating anonymity.

1. Ensure that all resources are accessed securely regardless of location
   We ensure that control over the security-relevant parameters remains at all times within the originator of a message. The violation of transport security should not be possible by malfunctioning or poorly configured nodes.

2. Adopt a least privilege strategy and strictly enforce access control
   As a design principle, information is kept hidden as much as possible within the system. We always assume that an adversary

   - makes some or all information within his reach available to others.
   - analyzes all information within his reach.
   - willingly breaks protocol rules to gain information, disrupts information flows, or other advantages.

3. Inspect and log all traffic
   We skip that part, as it is not suitable for a system offering anonymity. Logs generated over a long period might result in data that allows reducing anonymity sets retrospectively or minimizing their size.

6 Related Cryptographic Theory and Algorithms

Whenever handling obfuscating data and maintaining data integrity, cryptography is the first tool in an implementer's hand, as a vast amount of research in this area already exists. For this work, we focused on algorithms either researched in depth and implemented or research, which seemed very valuable when putting this work into place.

In symmetric encryption in this paper always assumes that

$$D^{K_a}\left(E^{K_a}(\mathbf{M})\right) = \mathbf{M} \tag{6.1}$$

For a key $K_b \neq K_a$ this means

$$D^{K_a}\left(E^{K_b}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.2}$$

$$D^{K_b}\left(E^{K_a}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.3}$$

A good symmetric algorithm has withstood academic crypto-analysis over a considerable period of time and has not been weakened so far. Multiple algorithms are ideally not built similarly and not rely on the same mathematical problems.

The following candidates have been identified for our work:

- AES
  NIST announced AES in **standard2001announcing** 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 (2020) unbroken. It has been weakened in the analysis described in [**tao2015improving**], which reduces the complexity by roughly one to two bits.

- Camellia
  The Camellia algorithm is described in [**rfc3713**]. The key sizes are 128, 192, and 256. Camellia is a Feistel cipher with 18 to 24 rounds depending on the key size. Up until 2020, no publication claims to break this cipher.

For all asymmetric encryption algorithms in this paper, we may assume that…

$$D^{K_a^1}\left(E^{K_a^{-1}}(\mathbf{M})\right) = \mathbf{M} \tag{6.4}$$

It is important that

$$D^{K_a^{-1}}\left(E^{K_a^{-1}}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.5}$$
$$D^{K_a^1}\left(E^{K_a^1}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.6}$$

For any other key pair $K_a^p \neq K_b^p$

$$D^{K_b^{-1}}\left(E^{K_a^1}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.7}$$
$$D^{K_b^1}\left(E^{K_a^1}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.8}$$
$$D^{K_b^{-1}}\left(E^{K_a^{-1}}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.9}$$
$$D^{K_b^1}\left(E^{K_a^{-1}}(\mathbf{M})\right) \neq \mathbf{M} \tag{6.10}$$

When looking for well-researched algorithms basing on different mathematical problems and having well-defined outlines, numbers decreased dramatically.

- RSA
  In **Rivest:1978:MOD:359340.359342** the authors **Rivest:1978:MOD:359340.359342** published with [**Rivest:1978:MOD:359340.359342**] a paper which revolutionized cryptography. In their paper, the authors described an encryption method later called RSA, which required a key pair (K_a) referenced as public (K_a^1) and private keys (K_a^{-1}). This system's novelty was that anything encrypted with the public key was only decryptable with the private key and vice versa.

  RSA is up until 2020 not publicly known to be broken (unless a too small key size is used). However, **Shor97polynomial-timealgorithms** described in **Shor97polynomial-timealgorithms** an algorithm that should enable quantum computers to break RSA far faster than traditional computers. In the **??** we further elaborate these effects.

- ECC
  The elliptic curves were independently suggested by [**Miller1986**]

and [**Koblitz04guideto**] in 1986. Elliptic curve cryptography started to be widely deployed in the public space in 2006. Since then, it seems to compete excellenty 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 $\approx 1MB$. By utilizing Gaussian elimination the key size may be reduced for transport by approximately factor 10. Even the resulting key size was still impractical and thus discarded as well. We were unable to identify any quantum capable algorithm that is able to reduce the key size of McEliece algorithm.

- NTRU
  In [**Hoffstein1998**] **Hoffstein1998** 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 in March 28th 2018 to 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.

- ElGamal
  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.

As introduced in [**feldman1987practical**], homomorphic encryption was from the beginning a strong candidate to be used in 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 efficiently to use matrices was not feasible and thus rejected.

6.1 Deniable Encryption

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

6.2 Key Sizes

The question of key sizes is difficult to answer as it depends on the current and future possibilities of an adversary, which again relies on non-foreseeable research. We collected several recommendations.

Encrypt II (http://www.ecrypt.eu.org/) currently recommends for a “foreseeable future” 256 bits for symmetric and asymmetric encryption based on the factoring modulus 15424 bits. Elliptic curve cryptography and hashing should be sufficient if used with at least 512 bits. Assuming the focus is reduced to the next $\approx$ 20 years. In that case, the key size recommendations are reduced to 128 bits for symmetric encryption, 3248 bits for factoring modulus operations, and 256 bits for elliptic curves and hashing.

According to the equations proposed by **Lenstra04keylength.** in [**Lenstra04keylength.**] 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 [**CNSASuite**] (superseding well known and often used [**nsa-fact-sheet-B**]) data classified up to "top secret" should be signed with RSA 3072+ or ECDSA P-384. They recommend AES 256 bits for symmetric encryption, for hashing at least SHA-384, and for elliptic curves, a 384 bit-sized key.

As it might seem unwise to consider the recommendation of a potential state-sponsored adversary and the formulas proposed by **Lenstra04keylength.** 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.

6.3 Cipher Mode

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

- Parallelizable
  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 modes more efficiently by distributing them on multiple CPUs.
- Random access in decryption
  Random access in decryption allows efficient partial encryption of a ciphertext.
- Initialization vector
  An initialization vector has advantages and disadvantages. One disadvantage is that involved parties must share an initialization vector with the message or before distributing it. It is essential to understand that the initialization vector itself usually is not treated as a secret, as 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 focused on modes with parallelizable, random access modes and did not carry out authentication. In addition to the characteristics mentioned above, the main focus was on whether there is an open implementation 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 plaintext 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 (Cipher Block Chaining)
  CBC extends the encryption by XORing 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 disadvantage, 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 Cipher Block Chaining)
  CBC extends the encryption by XORing, not the ciphertext but a XOR result of ciphertext and plaintext. This modification denies parallel decryption and random access compared to CBC.

- EAX
  EAX was broken in 2012 [**minematsu2013attacks**] and is thus rejected for our use.

- CFB (Cipher Feedback)
  CFB is specified in [**dworkin2001recommendation**] and works as precisely as CBC with the difference that the plaintext is XORed and the initialization vector, or the preceding cipher result is encrypted. CFB does not support parallel encryption as the ciphertext input from the prior operation is required for an encryption round. CFB does however allow parallel decryption and random access.

- OFB
  [**dworkin2001recommendation**] specifies OFB and works precisely as CFB except for the fact that not the ciphertext result is taken as feedback, but the result of the encryption before XORing the plaintext. This denies parallel encryption and decryption, as well as random access.

- OCB (Offset Codebook Mode)
  This mode was first proposed in [**rogaway2003ocb**] and later specified in [**krovetz-ocb-04**]. 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 [**lipmaa2000ctr**] and is a mixture between OFB and CBC. A nonce concatenated with a counter incrementing on every block is encrypted and then XORed with the plaintext. 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 XORed 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 [**rfc3610**]. It allows for padding and authenticating 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 [**mcgrew2004galois**], and is related to CTR but has some major differences. The nonce is not used (just the counter starting with value 1). An authentication token *auth* is hashed with H_{GFmult} and then XORed with the first cipher block to authenticate the encryption. All subsequent cipher blocks are XORed 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 was analyzed security-wise in **mcgrew2004security** and showed no weaknesses in the studied fields [**mcgrew2004security**].

  GCM supports parallel encryption and decryption. Random access is also possible. However, the 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 [**Martin2010**]. 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 [**Halevi:2003**] **Halevi:2003** 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 [**tschorsch:translayeranon**]. This mode shares the same properties as EBC but without the same cleartext block's weakness resulting in the same ciphertext. Similar to XEX, it requires a tweak instead of an IV.

6.4 Summary of Cipher Modes

Table **??** shows a summary of all modes analyzed previously.

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

Table 6.1: Comparison of encryption modes in terms of the suitability.

GCM and OFB are only suitable in special cases for our protocol as they perform authentication, which we usually omit from a message. OCB and ECB are “IV-less” modes making them very attractive for us. However, we need to consider that ECB is deemed broken and the discovered flaws are relevant to us if not handled properly. Especially suitable from performance perspective are CFB, CTR, ECB, LRW, and EME. Most of these implementations are uncommon in crypto-libraries. We will use these findings when defining our supported modes in ??.

6.5 Padding

A plaintext 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.

6.5.1 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 provide a clue whether the decryption was successful or not. RSAES-OAEP is the newer of the two padding standards

6.5.2 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 available data at the end of the block. All other bytes are defined but not needed.

6.5.3 OAEP with SHA and MGF1 padding

This padding is closely related to RSAES-OAEP padding. However, the hash size is larger, 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 essential to note that the last block’s payload shrinks to $keySizeInBits/8 - 2 - MacSize/4$.

In our approach, we chose to allow these four paddings. The allowed SHA sizes match the allowed MAC sizes selected 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 ensure that the chosen signing MAC and the bytes required for padding do not exceed the asymmetric encryption’s key size. While this usually is not a problem for RSA as there are keys 1024+

[1] included in auth

[2] Requires tweak instead of IV

bits required, it is a fundamental problem for ECC algorithms as there are much shorter keys needed to achieve an equivalent strength compared to RSA.

We 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 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 pad more than just one block.

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

7 Censorship Circumvention

Several technical ways were explored to circumvent censorship. All seem to share the following main ideas:

- Hide data (e.g., Tor pluggable transports).
- Copy or distribute data to a vast amount of places to improve the lifespan of data (e.g., Wikileaks).
- Outcurve censorship measurements (e.g., use a modified client to ignore connection resets).

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

7.1 Covert Channel and Channel Exploitations

The original term of covert channels was defined by **Lampson73anote** [**Lampson73anote**] 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 as 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 hiding information within this specific channel is not a usage of a covert channel. We refer to this as channel exploitation.

7.2 Steganography

Steganography is important when it comes to unlinking information. [**6828087**] and [**subhedar2014current**] give a very rough overview. As some of the types and algorithms address specific steganography topics (e.g., some hide from automatic detection and others address a human message stream auditor), we must choose carefully. In our specific case, the main idea is to hide within the sheer mass of Internet traffic. A human auditor screening all messages within a jurisdiction is considered a minor threat for obvious reasons. We will therefore focus on machine-based censorship.

As we will later identify SMTP as one of the main transport protocols, we focussed on the type of traffic found within this and similar protocols. Most of the binary data sent in SMTP are jpg images (see **??** on page **??**). 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 [**f5**]. It is a reasonably well-researched algorithm that attracted many researchers. The original F5 implementation had a detectable issue with artifacts [**F5broken**] caused by the image's recompression. This issue was caused only due to a problem in the reference implementation, and the researchers meanwhile provided a corrected reference implementation without the weakness.

YASS, as described in [**solanki2007yass**], was not considered a candidate. Although less researched, researchers found multiple weaknesses [**kodovsky2010modern**, **li2009steganalysis**].

In general, the availability of steganographic implementations was incredibly poor. Most of the algorithms are only available as M-code, simulators, or stream encoders, skipping all real-world implementation problems.

7.3 Timing Channels

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

7.4 Technical Forms of Censorship

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

7.4.1 Making Systems Unavailable by Censoring Lookups

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

Many jurisdictions have implemented such measures. It is considered 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:

- Using a public DNS server available worldwide.
- Using another protocol to hide the traffic .
  - A protocol with tunneling capabilities like SSH may be used to reach a system outside of the reach of the censoring adversary.
  - Using a fully blown tunnel such as a VPN.
  - Piggybacking a legitimate protocol such as DNS-over-HTTPS (DoH) [**rfc8484**] or DNS-over-XMPP [**xep0418**]

7.4.2 Making Systems Unavailable by Disrupting System Traffic

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

First, a censoring adversary requires high bandwidth. All traffic of a jurisdiction or target must pass through such a filtering device. This is usually not easily feasible for a country. A very high bandwidth system, such as the great China wall, uses a different approach. Instead of filtering each packet, they concentrate on TCP connections. Each slightly suspicious packet is copied to an analyzing system while the original message is routed normally. A subsequent system then analyzes the copied packet or packet sequence. If the subsequent system decides that the traffic should be censored, a connection reset is sent to the sender and the recipient. Any client or server having standard protocol support will immediately cease communication.

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 challenging as systems may change their addresses dynamically either due to cloud-related elasticity or due to an incomplete view of a distributed system (e.g., only a Loadbalancer is visible). An IP is therefore not necessarily synonymous with a single user or server.

When looking at the client side, they are often hidden behind a network address translation (NAT) or a proxy collapsing all users onto a single IP address. The same applies to the server-side, where cloud washing and reverse proxy infrastructures optimize bandwidth usage. Sometimes, in-depth information or insight into a protocol may help narrow down a user (e.g., by a set cookie or a fingerprint). When using encrypted connections, ordinary attackers have trouble carrying out a Man in the Middle (MitM) attack. This may be feasible for a larger attacker on a state or Internet service provider (ISP) level. To do so, such an adversary requires access to a publicly accepted CA, creating fake certificates for the attacker. It may be safely assumed that such access is given considering the standard set of CAs, which is trusted nowadays (depending on the delivered trust store, we found between 100 and 200 root CAs).

Identifying a target is especially difficult if a target comprises multiple possible targets from which some may be valid. This is the case when using a reverse proxy and using the

same platform for numerous purposes. Streaming or movie platforms may contain content that should be banned from a censors' point of view and content that comprises legitimate content such as educational material. From the censors' point of view, this content can not be reasonably split. This since typically, only the provider of the service can carry our selective censoring on the system. This is why governments try to shift the responsibility of censorship to the providers by establishing self-censorship.

7.4.3 Making Systems Unavailable by Interfering with System Traffic

Censoring may be achieved in more subtle or less abusive ways, such as traffic shaping or content moderation. We already outlined that the platform provider usually has content moderation in place. This is either achieved by allowing an entity to control the platform directly or indirectly to apply censorship or use legal means to force the platform into self-censorship.

Other means of censorship are:

- Redirecting all traffic to certain systems to filter according to the needs of a censor.
- Shaping traffic in such a way that the service is deemed no longer available to people of a jurisdiction (e.g., by slowing down traffic in such a way that streaming is no longer a viable option).
- Redirecting traffic to similar platforms employing a form of censorship (either a localized form of the respective information or an alternate provider of the same form of information).

7.5 Spread Spectrum in Networking Protocols

Another possibility of sending anonymous information is "spread spectrum" transmission. In spread spectrum transmission, a radio signal is distributed in the frequency domain. This makes it difficult for an adversary to identify and disrupt those radio signals in question.

While in use when carrying out radio-electric transmission, the spread spectrum is very uncommon in network protocols. We could employ multiple protocols and packet types to transmit data. Unlike in radio signals, such data is always available as discrete information pieces, and an adversary may choose to block them at any point. Unlike in radio transmission, where the available spectrum is almost indefinite and not fully blockable by practical means, full censorship does not oppose a problem. We may completely disrupt all communication by no longer routing it.

Anonymous Communication Systems

***

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

Ling Ma, Severance

In ?? we search for common Internet protocols suitable for hiding our traffic or accommodating data. In ?? we focus on technologies employed for problems related to anonymization or information hiding in general and analyze in ?? available systems in this field.

8 Well Known Standard Protocols

8.1 SMTP and Related Post Office Protocols (1982)

Today's mail transport is mostly carried out via SMTP protocol, as specified in [**rfc5321**]. This protocol has proven to be stable and reliable. Most of the messages are passed from an Mail User Agent (MUA) to an SMTP relay of a provider. From there, the message is directly sent to the recipient's SMTP server and then to the recipient's server-based storage. At any time the recipient may 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 occur by message polling or by message push (whereas a push–pull mechanism usually implements the latter).

To understand the routing of a mail, it is essential that we 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 ?? 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 mail at its destination changes drastically depending on the type of client used. Furthermore, the mail flow and control over it may differ on the client and the message processing on the server.

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 handling 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.

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).

In addition to 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 for 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 case of a rich client, the local MDA is part of the user agent's software. 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

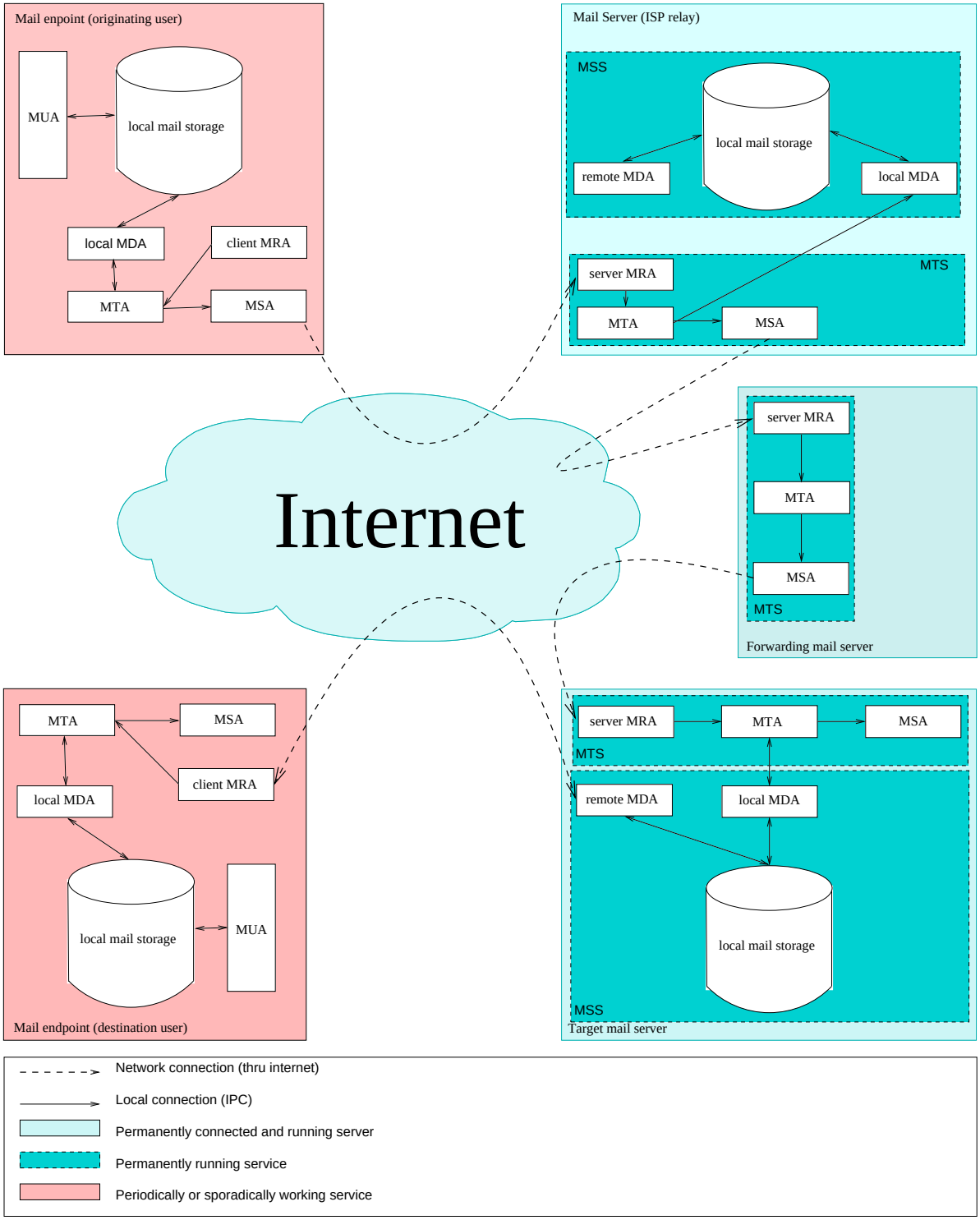

Figure 8.1: Mail agents.

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 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) is responsible for routing, forwarding, and rewriting emails. Moreover, a Mail Sending Agent (MSA) is accountable for transmitting emails reliably to another server MRA (usually sent via SMTP).

An MSS consists of local storage and delivery agents, which offer uniform interfaces to access

the local store. They 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 carried out 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 have external interfaces. These are MSA and MRA. MSA usually uses SMTP as transport protocol. This leads to some distinctive features.

- Port number is 587 (specified in [**rfc4409**]).
  Although port numbers 25 and 465 are valid and usually have the same capabilities, they are only for mail routing between servers. Mail endpoints should no longer use them.
- Connections are authenticated.
  Unlike normal server-to-server (relay or final delivery) SMTP connections on port 25, the server should always authenticate clients 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) [**rfc4409**]. Failure in completing 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. They share the properties of fat clients except 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 changes delivery and communication with the server. Instead of interfacing with an MSA and a client MDA, they may directly access the server's local mail storage. The local mail storage may be implemented as a database in a user-specific directory structure on these systems.

8.1.1 Fat Clients

The majority of mail clients are fat clients. They have a locally installed application on the client device to access mail allowing advanced features such as offline reading or bandwidth optimization. These clients score over the more centralistic organized web interfaces (web

clients) in 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 not to always be 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 (e.g., MSS). It provides a consolidated view of 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. Not doing so would result in massive interoperability problems. Most commonly, the protocols IMAP, POP and EWS are used. For email delivery, the SMTP protocol is used.

Fat clients are commonly used on mobile devices. According to [**clientDistribution**] in August 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%). **clientDistribution2** [**clientDistribution2**] as a more recent source lists in February 2014 iOS devices with 37%, followed by Outlook (13%), and Google Android (9%).

8.1.2 Server-Located Clients

Server-located clients are an absolute minority. This type of client 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 user’s home directory.

8.1.3 Web Clients

Presently, web clients are a common alternative to fat clients. Most large provider companies use their proprietary web client. According to [**clientDistribution2**] the most common web clients are "‘Gmail"’, "‘Outlook.com"’, and "‘Yahoo! Mail"’. All these interfaces do not offer a public plug-in interface. However, they typically do offer IMAP or similar interfaces. This is important for a future, generalistic approach to the problem.

8.2 S/MIME (1996)

S/MIME is an extension of 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 splitting 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 [**rfc3851**], S/MIME extends this standard with the possibility to encrypt mail content or sign it. Practically this is achieved by either putting the encrypted part of 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 the message subject nor the message size or frequency is typically hidden. This method offers limited protection assuming an adversary who is only interested in the messages’ content. It does not protect us from the adversary defined in our case.

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

8.3 Pretty Good Privacy (1996)

Exactly as S/MIME, PGP [**rfc4880**] builds on the basis of MIME. Since the trust model in PGP is peer-based, the encryption technology does not significantly differ (as seen from the security model).

Similar to 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 the subject line, the sender and receiver's real name, and the message size are leaked.

An important fact from PGP is that peer-based approaches offer 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.

8.4 XMPP

XMPP (or formerly Jabber) is defined in the RFCs [**rfc6120**, **rfc6121**, **rfc3923**, **rfc3922**] and features an own extension process on the base of XEPs. The community is very active in the development and has almost 200 proposed, drafted, active, final, or experimental XEPs.

At its core, XMPP is an open, secure, decentralized, and extensible standard for real-time capable protocol, allowing the efficient transfer of messages and signal status data. It allows single or multi-user chats and may be used as dialing protocol for voice, video file transfer, and for similar content.

We use XMPP in our work as proof of concept that a switch of protocols (in our case, SMTP and XMPP) is feasible.

The fact that the two protocols significantly differ in their cores makes it an ideal use-case. XMPP is synchronous (whereas SMTP is asynchronous, is not MIME-based (whereas SMTP is)), and has an own implementation for file transfers. On the other hand, it offers many advantages, such as the availability of end-to-end encryption or additional store-and-forward services.

9 Distribution for Anonymizing Protocols and Information Routing

Information routing and distribution is not a novelty in privacy research. Researchers around the globe have searched for means of privacy. One good example was the patent in the introduction of Almon B. Strowger [**pulseDialingPatent**]. More recent activities are the infamous "How to share a secret" [**shamir1979share**], which used Lagrange polynomials to distribute shares of information across multiple hosts for privacy. A single polynomial would be attackable. Shamir applied a $\bmod p$ 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 [**tompa1989share**].

Lagrange polynomes form an essential 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., [**aiache2008reed**]), distributing data [**shamir1979share**].

Our approach is 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 cannot tell which parts of the traffic are decoy and which part is the message, as all parts can recover the original message.

9.1 Mixing

Mixes were first introduced by **CHAUM1** [**CHAUM1**] in **CHAUM1**. The basic concept in a mix is as follows. We do not send a message directly from the source to the target. Instead, we use a proxy server or router in between, which picks up the packet, anonymizes it, and forwards it to the recipient or to 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 final mix knows the final recipient.

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

- In a low latency network, an adversary may trace the message 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 an attribute), we may follow the message.

In **RP03-1 RP03-1** 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 operates one mix is the most promising solution as static mixes always might be hunted by an adversary.

9.2 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 Mix Networks. 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 a reply address. The most well known pseudonymous remailer possibly was anon.penet.fi run by Johan Helsingius. Several times, this service was forced to reveal a pseudonym's true identity before Johan Helsingius decided to shut it down. For a more in-depth discussion of pseudonymous remailers, see **??**

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 a "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 **??**.

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 **??**.

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 **??**.

9.3 Onion Routing

Onion routing is a further development of the concept of mixes. In onion routers, every mix receives a message which is asymmetrically encrypted. By decrypting the message, the next hop's name and the content to be forwarded can be obtained. The main difference in this approach is that the mix decides about the next hop in traditional mix cascades. In an onionized routing system, the message chooses the route.

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

While tagging attacks are far more demanding (if we exclude side-channel attacks to break sender anonymity), the traditional attacks on mixes are still possible. Thus 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 ??.

9.4 Garlic Routing

Garlic routing is an improved form of onion routing. It stops onionized messages from continuously loose contents on their way. A garlic router collects multiple, independent messages into one message before routing. This compensates for the "size loss effect" of onionized systems.

9.5 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 [**crowds:tissec**], 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 JonDos. It was further developed to D-Crowds [**crowdsAttack**], ADU/RADU [**Munoz-Gea2008**], Freenet [**freenet**] and others.

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

9.6 Mimic Routes

Mimics are a set of statical mixes that maintain a constant message flow between the static routes. If legitimate traffic arrives, the pseudo traffic is replaced by legitimate traffic. An outside 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 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 with other network service streams.

9.7 Distributed Hash Tables

Anonymous file transfer is sometimes supported by Distributed Hash Tables (DHTs). Systems like I^2P (see geti2p.net), or Freenet [**freenet**] base on DHT. Hash tables are typically used for an efficient lookup of data distributed within a system. As they support the distribution of data, they may implicitly support error tolerance, robustness and, thus, availability. They furthermore may be used as distribution mechanism allowing self-organization, load balancing, and scalability.

In most anonymity systems using DHT, DHTs are either used to cloak nodes or services while enabling routing to them, or to build complex anycast structures.

9.8 Dining Cryptographer Networks

DC networks are based on the work **chaum-dc** by **chaum-dc** [**chaum-dc**]. In this work, **chaum-dc** describes a system allowing a one-bit transfer (the specific paper talks about the payment of a meal). Although all the DC net participants are known, the system makes it unable to determine who sent a message. The message in a DC-net is readable for anyone. This network has the disadvantage that a cheating player may disrupt communication without being traceable.

Several attempts have been made to strengthen the proposal of Chaum [**golle:eurocrypt2004**, **disco**, **herbivore:tr**, **Corrigan-Gibbs:2010:DAA:1866307.1866346**]. However, no one succeeded without introducing significant disadvantages on the privacy side.

9.9 Private Information Retrieval

Private Information Retrieval (PIR) [**chor1995private**] was developed by **chor1995private**. It is a public database organized in slots where some clients write into specific slots and other clients access the whole database so that the server is unable to tell what data was accessed. It is a simplified or weaker version of an oblivious transfer (1-out-of-n). PIR was described in theory and had two different approaches. A computationally secured approach (cPIR), which is the weaker one of the two approaches, and the information-theoretic secured approach (itPIR).

PIR was the foundation or an inspiration for many other systems and extensions such as CSPIR [**lipmaa2009first**], BddCpir [**lipmaa2009first**], Popcorn [**gupta2016scalable**], or Riposte [**corrigan2015riposte**].

10 Proposed Academic Protocols and Implementations

In this section, we list various proposed anonymity systems regardless of their age or state. We analyze their inner workings and try to compare them in a unified way. This comparison was a basis for selecting our approach.

10.1 Characteristics of Known Anonymity Implementations

Table **??** shows the protocols analyzed in the next sections ordered by type and year according to the classification scheme introduced in [**Shirazi2018**].

		Network structure						Routing info		Communication model					Performance and deployability				
			Connect		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
Resenders, onion routers and mixes	Chaum Mixes[1]	⊠	⟶	≠	••	⋯	✖	●	✖	•⋯	≡	✔	Ⓐ	◎	H	✉	✔	✖	@
	Babel[2]	⊠	⟶	≠	••	⋯	⊙	●	✖	•⋯	≡	✔	Ⓐ	⊛	H	✉	✖	✖	@
	Mixmaster[3]	⊠	⟶	≠	••	⋯	⊙	◐	✖	•⋯	≡	✖	Ⓐ	⊛	H	✉	✔	✔	@
	Crowds[4]	⊠	⟷	≠	•••	⋯	⊙	●	ϟ	•⋯	≡	✖	Ⓐ	⊛	L	✉	✔	✖	e
	Tor[5]	⊏	⟷	≅	•○•	⋯	⊙	●	◷	•	≡	✖	🏴	◎	L	✉	✔	✔	e
	I^2P^6	□	⟶	≠	•••	⋯	◯	●	◷	•	◇	✖	🏴	∗	L	⊷	✔	✔	e@
	Mixminion[7]	⊠	⟶	≠	••	⋯	⊙	●	◷	•	≡	✖	Ⓐ	⊛	H	✉	✔	✔	@
	$\mathcal{P}^{58}$	⊏	⟶	≠	•••	✤	⊙	◐	ϟ	📢	◇	✔	☺	◎	H	✉	✔	✖	✉
	AP3[9]	⊏	⟷	≠	•••	⋯	◯	◐	ϟ	•⋯	≡	✖	Ⓐ	⊛	L	⊷	✖	✖	e@📢
	SOR	⊠	⟷	≅	••	⋯	◯	●	✖	•⋯	≡	✔	☺	⊛	L	⊷	✔	✔	e@
	Vuvuzela	⊠	⟷	≅	••	✤	✖	●	✖	•⋯	≡	✔	☺	⊛	M	✉	✔	✔	✎
	Riffle	⊠	⟷	≅	•○•	✤	✖	●	✖	•⋯	≡	✔	🏴	◎	L	✉	✔	✔	e
	Karaoke	⊠	⟶	≅	••	✤	✖	●	✖	•⋯	≡	✖	☺	⊛	L	✉	✔	✖	📢✎
	MessageVortex	⊠	⟷	≅	•••	⋯	◯	◐	ϟ	•⋯	≡	✖	☺	⊛	H	✉	✔	✔	@
PIR	Riposte	⊠	⟶	≅	••	⋯	⊙	●	✖	📢	≡	✖	Ⓐ	⊛	H	✉	✔	✔	📢✎
	Pung	⊠	⟷	≠	••	⋯	✖	●	✖	📢	≡	✔	Ⓐ	⊛	M	✉	✔	✖	📢✎
DHT	Tarzan[10]	□	⟷	≠	•••	⋯	◯	●	ϟ	•⋯	≡	✖	🏴	⊛	L	⊷	✔	✔	e
	MorphMix[11]	⊏	⟷	≠	•••	⋯	⊙	◐	ϟ	•⋯	≡	✖	🏴	∗	L	⊷	✔	✔	e
	Salsa[12]	⊏	⟷	≠	•••	⋯	◯	◐	ϟ	•⋯	≡	✖	Ⓐ	⊛	L	⊷	✔	✖	e
DC	Chaum´s DCnet[13]	⊠	⟶	≠	•••	⋯	✖	●	ϟ	📢	≡	✔	Ⓐ	◎	H	✉	✖	✖	✉
	Herbivore[14]	⊏	⟶	≠	•••	✤	⊙	◐	ϟ	📢	≡	✔	🏴	◎	M	✉	✔	✖	✉
	Dissent in numbers[15]	⊏	⟶	≠	••	✤	⊙	◐	ϟ	📢	≡	✔	🏴	◎	H	✉	✔	✔	✉
	Verdict	⊠	⟶	≅	••	✤	⊙	◐	ϟ	📢	≡	✔	🏴	◎	H	✉	✔	✔	✉
BC	Hordes	⊠	⟷	≠	•••	⋯	⊙	●	ϟ	📢	≡	✖	Ⓐ	⊛	L	✉	✔	✖	e
	Atom	⊏	⟶	≅	••	⋯	⊙	●	?	•⋯	≡	✖	Ⓐ	⊛	H	✉	✔	✔	📢✎

Table 10.1: Classification table for anonymization protocols.

In the table, some historical systems were omitted. Additionally, SCION was omitted as it did not fit anywhere into the table. It may be seen as a remixing system, but too many aspects were intermixed with the routing logic to give really a clear classification. Furthermore, the SOR classification is highly speculative as this system has many missing aspects, making it difficult to categorize correctly. Where the paper does not give an exact indication of how a part is solved, we made guesses in favor of the work.

As key indicators for similar protocols, we identified the following characteristics:

- It needs to be peer-to-peer (•⋯•⋯•) or hybrid (•⋯○⋯•).
  The hybrid role is only allowed when no dedicated servers for the protocol are required. Dedicated servers would have the disadvantage of repression against administrators.
- They need to be fully decentralized (◯).
  An adversary may use central infrastructures to disrupt and control them.

- Routing has to be source-controlled (•···) or broadcast-based (📢).
  In any infrastructure where mixes decide about the route, an adversary may redirect a message to nodes under his control.

- The nodes must be user-defined (☺) or the system must have information-theoretic promises that even if all nodes collaborate, the system is not compromized
  In every system where the security relies on nodes' trust, a user should always be in full control.

- The system must work in a high latency mode (H)
  Every low/medium latency system makes promises regarding the traffic, which makes the system detectable.

Unfortunately, all of the protocols found implement their "own" protocol, rendering them easily censorable.

10.2 Resenders, Onion Routers, and MixNet-Based Systems

10.2.1 Pseudonymous Remailers (1981)

A pseudonymous remailer allows reaching people via a pseudonymous email address. The remailing server removes all traces of the original sender and inserts a pseudonymous email instead. The foundation of these remailers can be found in an early article by David Chaum [**CHAUM1**].

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 two compromises involving the Chruch of Scientology. Details of the closure can be found in [**penetClosure**].

It drastically shows the problem of legal prosecution even within so-called "democratic environments".

10.2.2 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. The time of the invention of the first type-1-remailers is unclear. 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 known, and authorities were able to ban infrastructure and capable of monitoring their routing activities. The standard mail logs of such servers provided valuable evidence for legal prosecution if not disabled.

10.2.3 Babel (1996)

Babel was an academic system defined in a paper by **babel** in **babel** [**babel**]. It was 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 want to remain anonymous, the initiator's RPI was 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, timestamps, and similar attacks are possible. Furthermore, the source of the RPIs on the message board was by design unclear and therefore not trustworthy.

10.2.4 Mixmaster-Remailers (1996)

Similar to Cypherpunk remailers, the Mixmaster remailers worked with onion-like encrypted messages. The protocol was based on Chaum's MixNets in [**CHAUM1**] and further developed by L. Cotrell in 1996.

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

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

10.2.5 Crowds (1997)

Crowds is an anonymity network for browsing and was the starting point for many similar systems such as D-Crowds, AN.ON and may be seen as a predecessor for Tor specialized in forwarding HTTP requests.

In Crowds, a user joins a crowd by registering at the blender node of a Crowd network a JonDo service. The network has, in addition to the blender, a variable number of nodes called JonDos. These nodes are forwarding nodes that either send a message to another random JonDo (including themselves) or forward it to the final recipient. The behavior is chosen based on a probability factor. The behavior is constant for a period (connection) and renegotiated from time to time (usually hourly). Furthermore, JonDos are required to strip any personal information from a request.

A JonDo acts as a proxy for a web browser or other JonDos. Therefore, JonDos' have plaintext access to the routed requests and replies. Messages between JonDos' are symmetrically encrypted. From the senders' point, Crowds offers perfect anonymity towards the receiver.

While the concept of blending into a crowd of members was inspiring for many other solutions, it has specific weaknesses. JonDos may be collaborating, or the blender may create subnetworks of collaborating JonDos' to break anonymity. Furthermore, the strict forwarding property makes it susceptible to the predecessor attack [**wright2004predecessor**], which intersects multiple (past) paths striving to reduce the anonymity set down to isolate the originating node.

10.2.6 Tor (2000)

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

According to [**onion-routing:pet2000**] Tor is a network consisting of multiple onion routers. Each client first picks an entry node. It establishes an identity, obtains 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 regularly 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 [**hs-attack06**, **esorics13-cellflood**, **bauer:wpes2007**, **esorics12-torscan**, **oakland2013-trawling**, **danner-et-al:tissec12**, **congestion-longpaths**] 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. 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 node. 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 achieved 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 can be found 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 analyzing 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 de-anonymized because it is using compromised routers. A generic analysis of low latency systems also relevant for Tor can be found in [**johnson2009design**].

Generally, the effectiveness of monitoring single nodes or whole networks is disputed. According to a study by **ccs2013-usersrouted** in **ccs2013-usersrouted** [**ccs2013-usersrouted**], 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 so far the NSA was unable to de-anonymize users of Tor. However, since these papers referenced “manual analysis”, the statement may be disputed when looking at automated attacks.

According to Tors’ plugable transport page, it is at the time of writing impossible to use transborder Tor traffic at least in China, Uzbekistan, Iran, and Kazakhstan. 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) and using public proxies such as Appspot.

[**saleh2018shedding**] is an excellent survey listing recent developments and attacks within the Tor project.

10.2.7 I^2P (2001)

The name I^2P is a derived from "Invisible Internet Project" according to geti2p.net. The first binary release on SourceForge dates back to 2001. The system itself is comparable to Tor for its capabilities. Major 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. Thus, it is difficult to judge its real qualities.

In **pets2011-i2p pets2011-i2p** presented in [**pets2011-i2p**] 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 paper's main critics were that the peer selection might be influenced by an adversary, enabling him to recover data on a statistical basis.

10.2.8 Mixminion-Remailers (2002)

Mixminion was the standard implementation of a type-3-remailer. It tried to address many previously unresolved issues.

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, the final recipient had to be aware of the Mixminion system.

Mixminion-Networks have been privacy-wise criticized for the following:

- Pseudonymous single use reply blocks are broken (Chapter 4.2 in [**sassamanpynchon**]).
- Central directory of mixes.
- Not enough users.

According to `https://mixminion.net`, the software's first release was in December 2002 and was discontinued in 2008. Since 2011, the sources are available on GitHub. There were forks in 2011, but currently, all forks seem to be inactive since at least 2016 as there are no new commits.

10.2.9 $\mathcal{P}^5$ (2002)

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

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

10.2.10 AN.ON (2003)

AN.ON, as suggested in [**federrath2003system**], 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 in many ways similar to the capabilities of Tor. The network was at the time of writing much 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 observe 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 leaks as well.

10.2.11 AP3 (2004)

AP3, as defined in [**mislove2004ap3**], 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 [**ccs2008:mittal**], and does not withstand our adversary as the final recipient is known to the routing nodes.

10.2.12 Cashmere (2005)

Cashmere is specified in [**zhuang2005cashmere**]. 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 Internet. The project homepage http://current.cs.ucsb.edu/projects/cashmere/ has not been updated since 2005. This suggests that this project is either dead or sleeping.

10.2.13 SOR (2012)

SSH-based onion routing (SOR). [**Egners_2012**] criticize the complex and monocultural landscape of anonymizing software and proclaims a very simple approach based on onionized SSH tunnels to forward TCP streams. The system might be modified further to forward UDP or other protocols asn well by using an instance of netcat converting UDP traffic into a forwardable data stream.

The system was not really up to date at the time. While using a common, encrypted, and well-established protocol for the onionization, the system lacks obvious protection against timing attacks. Which is due to the fact that either no access control is carried out or users have pseudonymous accounts on the target system (making them identifiable) or the predecessor attack [**wright2004predecessor**] which were all well known at the time.

Unlike most other systems, SOR does not introduce an own protocol but uses an existing protocol with many legitimate uses. This makes it difficult for an adversary to ban the protocol. Its approach in terms of hiding may be seen as somewhat similar to our approach.

10.2.14 PGA (2013)

Pretty Good Anonymity [**standtke2013pretty**] attempted to create a single node anonymity service. The client is running a local proxy which encapsulates the client traffic into the "PGA Tunneling Protocol". The protocol may hide traffic by adding additional (adaptive) decoy traffic (dummy traffic). It may be seen as a low latency encrypted SSH tunnel with additional anonymity features such as decoy traffic. It is capable of tunneling in a generic way any kind of TCP connection. UDP is not known to be supported.

We did not include it into our table as it never achieved broad adoption and there is no routing involved.

The system does not withstand our adversary, as the PGA tunnelling protocol is detectable. Unlike most of the systems, an implementation of the system in Java is available.

10.2.15 Vuvuzela (2015)

Vuvuzela was presented by **van2015vuvuzela** in [**van2015vuvuzela**]. 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 is immediately aware that clients use Vuvuzela. He is, however, unable to match up with communication peers over time. The Vuvuzela client software is available under and connects to a Vuvuzela network forming a centralized infrastructure. According to its authors, Vuvuzela infrastructure may handle up to 10 million users with an average bandwidth cost of 3.7KB/s per user.

Vuvuzela routes user messages through a chain forward and back again before redistributing the final messages to their recipients. Each node adds additional decoy traffic to further improve the anonymity of the message path. The authors calculated that each user contributes $\approx 12\frac{KB}{s}$ traffic adding up to $30GB$ per month. A server node had an average throughput $166\frac{MB}{s}$. Vuvuzela protects the messages of peer partners as long as one server in the used chain is not controlled by the adversary. It however does not protect the fact that both peer partners are using Vuvuzela.

Vuvuzela assumes that the chain of servers and the involved public keys are known to the

client ahead of time. Messages are delivered in synchronized rounds into common, ephemeral dead drops created by the users. The ephemeral dead drop design makes it impossible for an adversary to identify users over time.

10.2.16 Riffle (2016)

Riffle [**kwon2016riffle**] 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 sending 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. Such a system is far better protected against timing attacks than Tor at the price of a considerable higher latency and bandwidth.

Thus 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. However, as it uses its own protocol, traffic is identifiable and thus, censorable.

10.2.17 MCMix (2017)

In [**alexopoulos2017mcmix**] **alexopoulos2017mcmix** introduce a messaging system based on Multiparty Computation (MPC) suitable for routing up to 100K users in less than a minute for tweet-sized messages. The protocol has a theoretical parallelizable variant to increase the size of such a group. The network load remains constant, depending on the maximum supported message size. The authors estimated a constant data stream of $78\frac{MB}{Month}$ when using a 144 (SMS/tweet-sized) message and a round time of 1 minute. As in other protocols (e.g., PIR or Riposte), MCMix uses “dead drops” to replace connections to communicate between two or more entities.

The system uses a predefined hierarchy of entry servers for receiving all input data, an MPC server cluster to handle the MPC calculations, and output servers to provide messages to the final recipients. Accounts are derived by generating a public key from the username. This eliminates the need for a centralized PKI.

The authors implemented the input and the output servers but only simulated the MPC part.

10.2.18 SCION (2017)

SCION [**perrig2017scion**] is a clean slate Internet protocol. While SCION is not an anonymizing protocol, it contains many interesting features. Unlike with the traditional networks, we have the possibility of influencing the routing of data within SCION. Furthermore, with PHI [**chen2017phi**] and Dovetail [**sankey2014dovetail**], SCION may feature strong and fast anonymity features.

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

10.2.19 Karaoke (2018)

Karaoke [**lazar2018karaoke**] 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.

Karaoke uses 'dead drops" to transport messages. The access is organized in rounds, and in each round, all users access one dead drop. The access itself is controlled by a series of mixes shuffling the requests and replies. The path of the message through the mixes is chosen by the sender. Messages within the Karaoke system have a fixed size.

Karaoke does not withstand an active adversary. In an environment with a passive observer, Karaoke may make some very strong promises about privacy. However, since Karaoke uses its defined own infrastructure, its users are easily identifiable.

10.3 PIR-Based Systems

10.3.1 Riposte (2015)

Riposte [**corrigan2015riposte**] is an anonymity messaging system inspired by DC networks that scale well for tweet-sized messages. The messages are sent on a regular basis (time epochs). The system achieves sender anonymity by distributing parts of a message over multiple hosts. To reduce the size of the transferred message, Riposte uses a Distributed Point Function (DPF) described in [**gilboa2014distributed**]. This reduces the messages transferred to each server to $\sqrt{databaseSizeBytes}$.

In some ways, Riposte turns the PIR system upside down. Instead of someone writing in a database slot and then not disclosing which slot was accessed by a recipient, Riposte makes the writing of the slot anonymous, and the recipient may freely access the interesting slots. The classification is however not clear as it involves mixes as well as DC-nets.

10.3.2 Pung (2016)

Pung as introduced in [**angel2016unobservable**] is a further development of PIR [**chor1995private**] which was proposed in **chor1995private** and implemented in systems such as Riffle [**kwon2016riffle**], PIR-Tor, or Pynchon Gate.

As many other systems, Pung works in rounds. To reduce the set of records to be fetched from the PIR database, labels are applied to the records. By filtering by labels, the recipient hides within an anonymity set. The sender chooses the labels, and the recipient has to query a sufficient set of labels to create a sufficiently large anonymity set. As this is difficult to accomplish on a random scale while maintaining credible traffic, we believe that this is one of the major weaknesses of Pung.

The authors of Pung claim that a four server setup may handle up to 135K messages per minute when having 32K active users. The message's size was chosen to be 256 bytes matching the block size of the applied crypto.

As most other anonymity systems, Pung has its protocol and thus remains easily detectable and censorable. The traffic overhead is substantial and is on a per-user base. The speed of message delivery is dependent on the time of the chosen epoch.

10.4 Distributed Hash Tables

10.4.1 Tarzan (2002)

Tarzan is a P2P IP protocol using UDP to communicate. It is specified in [**tarzan:ccs02**]. 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.

10.4.2 MorphMix (2002)

MorphMix was a thesis published by **morphmix:wpes2002** in [**morphmix:wpes2002**]. MorphMix was among the first to introduce a pure peer-to-peer anonymity protocol. Users and mixes were indistinguishable, and there was no cover traffic generated to save bandwidth. For anonymity, it uses a source-controlled, onionized routing system. Nodes are discovered by querying any random first node. It was a circuit-based mix system for networking anonymity. The core of the network was collision detection. This detection was circumvented by [**morphmix:pet2006**]. Since then, no new papers were published and the project seems to be dead.

In many respects, MorphMix may be seen as an ancestor of *MessageVortex*. However, *MessageVortex* goes far beyond the capabilities of MorphMix while eliminating most of its weaknesses.

10.4.3 Salsa (2008)

Salsa was proposed in [**Salsa**] and described a circuit-based anonymization pattern based on distributed hash tables (DHT). An implementation for Salsa is available, but it is not public. [**ccs2008:mittal**] claims that by combining active and passive attacks, anonymity can be compromised.

10.5 Dining Cryptographer-Based Networks

10.5.1 Herbivore (2003)

Herbivore is a network protocol designed by **herbivore:tr** in [**herbivore:tr**]. It is based on the dining cryptographers paper [**chaum-dc**]. No herbivore client or an actual protocol implementation could be found on the Internet at the time of writing. Wikipedia lists Herbivore as "dormant or defunct".

10.5.2 Dissent (2010)

Dissent is defined in [**Corrigan-Gibbs:2010:DAA:1866307.1866346**]. It is an anonymity network based on DC-nets. A set of servers forms these DC-nets, of which at least one in the used net must be trustworthy, and none may be misbehaving. A server failure results in the stalling of all message delivery using this server.

In an attempt to improve Dissent **wolinsky2012dissent** introduced in [**wolinsky2012dissent**] a modified version. This improved version mainly addresses the scalability issues of the original design. Furthermore, the authors addressed some information leakage and scalability flaws in the original approach.

10.5.3 Verdict (2013)

Verdict [**180367**] is an improved version of Dissent using proactively verifiable DC-Nets. It uses zero-knowledge proofs (ZKPs) to detect misbehaving nodes. The authors claim that it can process 1000 senders within 10 seconds.

Unlike many other systems, Verdict withstands an observing adversary as defined within this work. However, due to the message patterns generated when communicating even when steganographically hiding the traffic, a censoring adversary would detect the traffic generated. Tampering with the protocol itself would be detectable, and thus honest nodes could exclude misbehaving nodes from such a DC-net.

10.6 Broadcast and Multicast Networks

10.6.1 Hordes (2002)

Hordes was a multicast-based protocol for anonymity specified in [**Levine:2002**]. Hordes is a Crowds system that uses multicast services for the reply, thus speeding up the latency loss of Crowds. Hordes uses the ability to handle multicast addresses by routers to generate a dynamic set of receivers and then send messages. 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 for the adversary defined in this paper.

10.6.2 Atom (2016)

Atom [**kwon2016atom**] 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 series of AWS-based EC2 instances. It provides a broadcast primitive with limited reach by grouping 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 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 carried out by trustworthy third party "directory authorities". To remain anonymous, at least one honest node per group is required. Identifying malicious users in Atom requires a collaborative effort involving the publications of the entry groups' private keys. Malicious users are proposed to be blacklisted by the directory authorities.

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

10.7 Distributed Storage Systems

10.7.1 Freenet (2000)

Freenet was initially designed to be a fully distributed data store [**freenet**]. 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 the caching of a document if requested more often.

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

Freenet features two protocols FCPv2 acts as the client protocol for participating in the control of Freenet storage. The Freenet client protocol allows us to insert and retrieve data, query the network status, and manage Freenet nodes directly connected to their 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 (the last update on the FCPv2 Page was August 8th 2020 at the time of writing).

10.7.2 Gnutella (2000)

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

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

10.7.3 Gnutella2 (2002)

Despite its name, Gnutella2 is not the next generation of Gnutella. It was a fork in 2002 from the original Gnutella and was developed in a different direction. The specification of Gnutella2 is available at `http://g2.doxu.org`. Just as its predecessor, Gnutella2

seems to be dead. The last update to the main site was in 2016 and the last update to the protocol on 2007.

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 section, we describe the core parts of the *MessageVortex* protocol. Unlike most other academic attempts, we do this based on an adversary 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 **??**. Based on these requirements, we explain our architectural concepts and decisions in **??**. We then build an outline of our protocol focusing on the protocol's main properties without going too much into implementation details. In **??**, 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 implementation details are described in the RFC draft document in **??**. Additionally, we describe the implementation's academically relevant details and their realization in infrastructure, in **??**. For operational concerns such as route-building strategies, refer to **??**.

11 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. This procedure allows us to build a coherent model for our target protocol.

We collected an overview of all isolated characteristics of **??** in **??**. These properties are vital for the success of our system. We will elaborate on success or failure in **??**.

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 account for an entity without being linked to a real identity.
RQ7	Message	Untagable	The message should be untagable (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	Easily 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 11.1: Summary table of requirements.

11.1 Threat Model

Within this work, we look at two adversaries with differing behavior. The two adversaries are an "observing adversary" (mainly 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 assume that one of these adversaries is present within any jurisdiction.

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 have executive capabilities (e.g., police, army, or secret service) to enforce this legal rule set.

We assume for our protocol that adversaries are state-sponsored actors or players of large organizations. Furthermore, 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 popular software weaknesses (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 freely modify routing information within a jurisdiction.
- 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 have similar access to resources as within its jurisdiction in a limited number of other jurisdictions.
- 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.

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. He serves us as a worst-case adversary.

11.1.1 Observing Adversaries

This adversary behaves like a traditional spy. He collects and classifies information while typically hiding his 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 a censoring adversary, as the technology is "censored." Such a classification must be carried out in this case, regardless of whether the adversary only tries to collect information or not.

11.1.2 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 his activities and typically classifies censorship circumventing actions as illegal.

In such environments k-anonymity, as specified in [**k-anonymous:ccs2003**], may not be sufficient for such an adversary. Instead, the *MessageVortex* system must hide all activities from such an adversary.

11.1.3 Realism of the Assumed Adversaries

The adversaries defined above are not realistic but "worst-case assumptions". An adversary may monitor certain spots within a network. Such spots are typically either jurisdictional borders or neuralgic points within a jurisdiction, such as the central router of an Internet service provider (ISP). However, it is not realistic that an adversary can tap any point of a network at a jurisdiction scale. Such tapping would require almost infinite bandwidth and unlimited access.

Accessing cryptographically weak protected data is possible. However, accessing or modifying such data typically requires a high amount of calculation resources. Such resources may be available for a single case, but they typically do not scale if we assume high protocol usage,

Modifying network traffic would require even higher evolved capabilities as such modification requires tapping of a network and the capability to actively modify network traffic. Such modification is in practice almost always limited to a broadcast domain. This limitation typically means that all devices within a broadcast domain receive the same messages except if we direct the message to a single device. In our model, we state that the traffic may be freely modified at any point within the jurisdiction. This assumption is not realistic underlying today's common network technologies. Furthermore, it is not realistic that a state-sponsored actor will carry out a DDoS attack against an entity within a jurisdiction, as simply blocking traffic would be far more effective and less resource binding. However, a DDoS attack may be a good solution when disrupting services within a jurisdiction not cooperating with the adversary's goals.

However, an adversary may have a limited number of accesses to the network with exactly these capabilities. As we cannot define or limit the number of access points, our defined adversaries reflect a worst-case assumption that may not be surpassed. Therefore, our adver-

saries, while not realistic, reflect a state where, if our protocol withstands such adversaries, it may be considered safe.

11.2 Required Properties for Our Unobservable Protocol

In this section, we collect the required properties for our system. We first list a property and then explain why it is essential.

11.2.1 Required System Properties

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

Users are unable to limit the route of network packets through named jurisdictions. Therefore, we must protect users of *MessageVortex* from being subject to legal prosecution in any jurisdiction. All these users need to be anonymous when sending or receiving messages. This limitation applies not only to their communication but also to the usage of anonymization technology. Unfortunately, most transport protocols (in fact, all of the common ones such as SMTP, SMS, XMPP, IP, or messengers) use a globally unique identifier for senders and recipients. These addresses are readable by any party capable of reading the packets (mainly the routing nodes). This identification contradicts anonymity.

In the threat model in ??, 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:

- Identifying the sender.
- Identifying the recipient.
- Identifying other involved parties (e.g.f, routers).
- Reading messages passed or extract meta information.
- Disrupting or modifying communication fully or partially. This may or may not include the possible identification of the traffic.

If users need to stay anonymous, they must protect their traffic from influences outside the system. 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 the transport protocol's communication.

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 cannot 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 is based on the fact that an adversary trying to establish censorship may be part of the jurisdiction's government. We may safely assume that there are legal exceptions in some jurisdictions for such entities. Having such legal means enables an adversary to introduce legally spying nodes into our system.

To withstand an adversary outlined in **??**, the messages sent even within the system must be unidentifiable by meta-information or content. "Meta-information" may refer to any 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.

If we want a user's traffic to remain indistinguishable from traffic generated from routing nodes, all traffic must have the same properties. This applies not only after "entering the system" but at any time. As a result, only an infrastructure-less approach may be used as a consequence. A hybrid or server-based approach requires infrastructure to be placed within the Internet. Jurisdictions with a censoring adversary may place focus on such systems and identify and prosecute their owners.

Furthermore, it must be impossible for an observing adversary to identify message endpoints. All nodes must look equal from the outside in terms of traffic, as well as by 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 and no differentiation in function may be achieved.

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. 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 be suspicious of 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 distrusted party. 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 the trust, we 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 problem. Thus, trusting these infrastructures is inevitable.

2. We should not trust any other infrastructures, as an adversary can misuse data passing through.

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's point of view. This prevents building social graphs and narrowing down groups of individuals.

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

Unobservability requires, according to [**anonTerminology**], an item of interest (IoI) to be undetectable from an uninvolved entity and anonymous for the involved entities. We therefore require anonymization as a property.

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 the trust into such an infrastructure. 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 accomplished in Cypherpunk remailers (see **??**). If we cascade more relay-like infrastructures and combine them with cryptography, we may achieve sender and receiver anonymity. When we then introduce 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., [**penetClosure**]).

[**anonTerminology**] 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 account 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 them by identity. Unlike other protocols, we have no identifier as we work based on the previous requirement anonymity. We will however require some type of accounting to keep adversaries from flooding our system.

11.2.2 Message Requirements

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

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

To protect a message from being followed or observed, a message requires 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 control of the previous node.

?? 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. This may only be done if any node apart from the sender and recipient node does not have access to the message being transported or the message is protected from modification.

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 carried out by introducing remotely hosted data or by introducing revocable certificate operations into the message stream and then observing the VA of a PKI for respective OCSP calls or CRL accesses. DNS or similar information lookups may be used as well. Our protocol handling must not depend on such external lookup or download mechanisms to ensure that bugging is not possible.

This property applies not only to the message content itself but also to any routing node processing. All operations carried out need to be standalone and should not be queryable or detectable from an outside observer even if he is able to manipulate the message content.

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 size), which may lead to the discovery of such meta-information.

11.2.3 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.* 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.

This bootstrapping is needed for users to extend their network at first to a reasonable anonymity set assuming an adversary inside the system. At the same time, the bootstrapping mechanism is a great danger as it allows an adversary to harvest nodes. As a result, each node must be able to control by whom a node is discoverable.

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.*

Weaknesses in algorithms are discovered quite commonly. We may therefore not rely on a single algorithm. Instead, we must create a protocol supporting processing alternatives for algorithms. This includes crypto agility, as described in [**bsiPostQuantum**].

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 sacrifice functionality or usability for security. If we want our system to be secure, we require many users to generate a sufficient level of decoy traffic. 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 into account as a requirement.

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 [**zhou2011examining**].

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

Transparent behavior is a prerequisite for reliability. If something generates acertain behavior, but a user is unable to determine the reason for it (i.e., if a user expects a different behavior), he would usually assume a malfunction. Therefore, "reliable" means not only stable by its behavior. It also means that the system has to be 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., [**nicholson2003assessing**]).

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 which means that the system needs to be available.

Availability, in this specific context, may have two differing meanings. 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 dysfunctional state and, therefore, possibly 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 legitimate 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.

We did not consider efficiency as a requirement, as our goal is to achieve anonymity under harsh conditions.

12 Rationale

In this chapter, we set the course for our system. We explain why we built the protocol the way it is. We 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 disassembles and reassembles the messages received and applies specially crafted operations, and the accounting layer tracks the quotas and protects the system's resources. The three *MessageVortex* layers (all layers except "transport") run on common Internet end-user devices such as mobile phones or tablets.

12.1 System Design and Infrastructure

All anonymity systems listed in **??** have in common that they rely on dedicated servers providing an anonymity-related service. Such specialized servers make operators or owners of such servers vulnerable in an environment where a censoring adversary (as described in **??**) exists. Therefore, our approach should be different. Instead of creating our own protocol, we describe a system where we use pre-existing standard servers without modification for our purpose. If we succeed in ivisibly piggybacking such a protocol, we may inherit the regular usage of this infrastructure as decoy traffic. Piggybacking and mimicking protocols is not new. Protocols such as Skypemorph [**mohajeri2012skypemorph**] or pluggable transports for Tor (e.g., Meek, FTE, or OBFS4) use this technology successfully for censorship circumvention.

Piggybacking is executed in a protocol-agnostic manner. On the protocol level, this requires that we separate the embedding of messages into the transport protocol from the rest of the system. This makes the system even more difficult to observe as routing graphs taking multiple protocols into account increase the complexity exponentially through their different properties.

Important properties of piggybacking are mainly:

- …the importance or significance of the transport protocol.
  The more important the transport protocol, the higher the barrier to censor the entire protocol.
- …the traffic load.
  The higher the load created by the transport protocol for analyzing the data, the more difficult it is to uncover messages.

- …the quality of the piggybacking
  The harder it is to identify a single message as part of the protocol or not, the harder it is to establish censorship.

The content of the message in the transport layer protocol is provided by the routing node and not by anyone or anything 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 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, we have to ensure that when designing our approach not to violate requirement **??**. For obvious reasons, a direct connection between the sender and recipient via any named transport protocol would violate the requirement **??**. A single intermediate node would minimally imply trust in this node and its anonymization capabilities, which is not acceptable due to the requirement **??**. When using multiple nodes, other anonymization protocols typically use three to five intermediate nodes due to their arguing. Such protocols typically 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 a **??** policy, decisions for routing may no longer take place on the routing node but must be dictated externally. Some protocols (such as a typical Crowds-based system) have weaknesses as each node may decide on the subsequent node and choose one in their favor.

For routing, we will use end-user devices. This decision is further backed by the requirement **??**. It however opposes the requirement of **??**, 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.

12.2 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 manipulatable 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 **??**.

- 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 **??** and **??**. 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 no 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. Additionally, the traffic of our system is cloaked in mimicking traffic, making it next to impossible for an outside observer to identify message flows. In other systems however, the node generating decoy or mimicking traffic is well aware of the true message flow. In our system, instead of mimicking traffic, we add redundancy information (or remove it). By doing so, a routing node no longer has insight into which part of the traffic is relevant to the message and which part is not. Furthermore, we may introduce the possibility of distributing the message content throughout multiple paths in such a way that each path has insufficient content to rebuild the message. In fact, depending on the complementing missing message, any content received or sent by a node may be valid in our system.

- 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 for highlighting traffic. An adversary may either inject message payload (corrupting the message flow) or replay the message. While we cannot keep anyone from violating the rules, we may at least implement replay protection. Furthermore, we may later discover that we are able to identify willingly induced or 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 with a sufficient number of observation points unless mimicking 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 our main routing operation.
  Originally, [**reed1960polynomial**] introduced a system allowing the use of polynomes to create error-correcting codes. In [**chaum1988multiparty**] **chaum1988multiparty**, have shown that the codes are suitable for distributing data assuming enough parties are honest and not malfunctioning. Unlike **chaum1988multiparty** proposition, we do not use the Reed–Solomon-function to achieve anonymity or privacy. Instead, we use it for decoy traffic generation. We split a message into multiple parts at several points by adding redundancy information while routing and assembling it again on the target node. 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.

- *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

from either the sender or the receiving system. The German Federal Office for Information Security (BSI) makes recommendations in [**bsiPostQuantum**] for systems and protocols, which we intend to follow.

The main text can be condensed to the following recommendations:

- A protocol or system should be crypto agile.
- A protocol or system should use 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 large, 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.

12.3 Summarizing Chosen Approaches for *MessageVortex*

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

- Piggybacks common protocols.
- Does not require specialized infrastructure 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.
- Has no strict message size and strictly avoids 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 offers routing operations instead of traditional mixing and recombination methods.
- Offers a choice of algorithms when routing.
- Offers short-lived pseudonyms to enable the identification of the original sender.

The protocol is a four-layer protocol, as shown in ?? on page ??. 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 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 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 also scalable and suitable for a clustered environment. Blending can be achieved 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 a stateless service with a minimal NOSQL-storage for all eIDs.

13 Protocol

MessageVortex is a protocol piggybacking standard transport protocols similar to S/MIME [**rfc2015**] or PGP [**PGP**]. Unlike these protocols, we require 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 credibly 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 an encrypted, structureless blob, which 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”) on how the payload blocks are processed and where to send the resulting blocks. Those operations are one of the keys as information leakage occurs in this step in most of the systems. We therefore crafted all operations very carefully to keep as much information secret as possible. These operations are key to the system as they allow us to increase and decrease the size of a message without revealing what part of the data is a decoy and what is 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. The implementation and operational aspects are discussed in **??** and **??**.

13.1 Protocol Terminology

For our protocol, we use the following terms:

- **sender:** The user or process originally composing the message. In contrast to the sender, the immediate sender is the node sending the message to the current node. It may or may not be identical to the sender.
- **recipient:** The user or process destined to receive the message in the end.
- **user:** Any entity, running a *MessageVortex* node.
- **router:** Any node 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. 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 has 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 and anyone possessing the private key (proven by encrypting with it) is accepted as representative of that identity. An eID has a workspace associated to it. Please note that an eID is not identical to an ID which is a numerical identifier for a payload block storage location within a workspace.
- **Routing Block Builder (RBB):** An entity, which builds a routing block. Typically identical to either the sender or recipient.

13.2 Key Components

The following sections describe some key components of the system. Understanding them 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 sender, recipient, or router.

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

Understanding the protocol layers' inner workings is essential to the understanding of the project as a whole. We emphasize their main function and their inner workings without going into implementation details. These details are further discussed in **??**. We mainly focus on the data and the high-level processing within these layers.

13.2.1 Nodes and Their Identities

We refer to a *VortexNode* (node) as a system run by an individual containing a 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). A node is not a server but a device connected to a regular, unmodified transport service provider. Such transport services may be an SMTP/IMAPv4/POP3 account, an XMPP account, or a similar transport protocol account.

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, because without the knowledge of such keys, no reply from a node is received. The transport endpoint itself is not a secret. The usage as *VortexNode*however is kept secret as long as the key is unknown.

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

13.2.2 Workspaces and Ephemeral Identities

We dumped the approach for a system with a global, unified storage for all message processing. Such a design would allow an adversary to flood our storage. Instead, we introduced temporary storages suitable for a set of transactions belonging to a single identity or a limited set of collaborating entities. In our system, every transaction on a node 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 message processing. Each eID has a storage assigned to which we refer as "workspace". A simplified outline of a workspace is shown in **??**.

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 send 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 payment in a 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 few 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 raises the bar for DoS attacks. Even if someone floods a node with new eIDs, already created eIDs are not affected as their workspace has already been allocated.

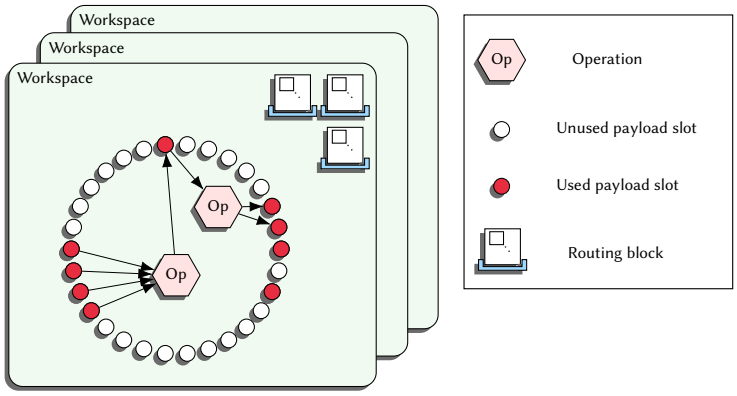

Figure 13.1: Simplified outline of a workspace in a *VortexNode*.

The workspace (see ??) 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 reflecting the current message [ingoing or

outgoing]).

This concept has certain disadvantages related to the expiration of eIDs. We will address them in ?? and ??.

13.2.3 Protocol Layers

As already introduced in ??, the protocol is built on multiple software layers. The layers are shown in ??.

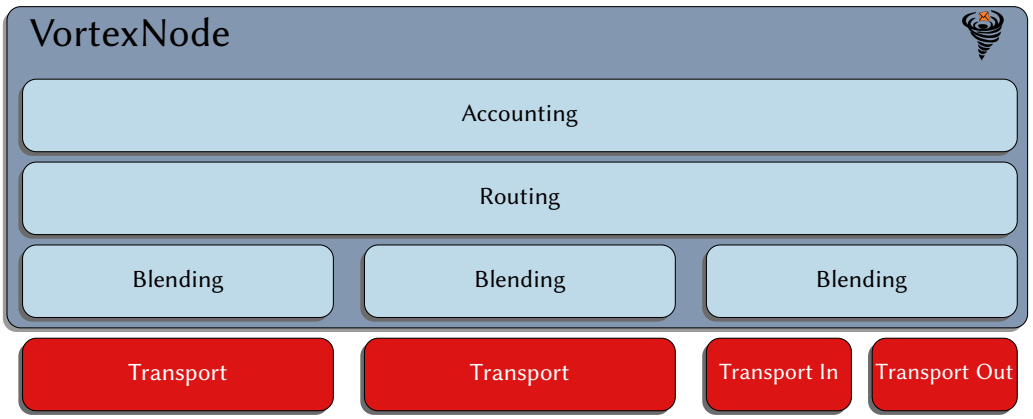

Figure 13.2: The protocol layers.

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) Blending 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, it has to authorize the message processing after the decryption of the header block by the blending layer. Secondly, the accounting layer handles all header request blocks and the reply blocks. 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*. There is always one accounting and one routing layer. Blending layers exist on a "need to have" basis. Typically, there is one blending layer per transport protocol or transport protocol account.

13.2.4 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 of a storing and forwarding medium. This medium solves two major problems. First, no NAT traversal technology such as "TCP hairpins" or "hole punching" is required. Secondly, it compensates for short outages due to regional routing problems to the end-user (e.g., networking problems on the Internet).

A transport layer should have some generic properties:

- Widely adopted
- Reliable
- Symmetrically built

For a more detailed description of the criteria, see **??**.

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 **??**, we share a short analysis going through some common protocols outlining the strength and weaknesses of common transport protocols within the Internet.

After that, we focused 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.

13.2.4.1 Blending Layer

The blending layer solves multiple problems:

- It translates 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 hiding it within a message using steganography. Another demanding task in this context is to create credible content for the transport message itself.
- Extracts incoming blocks
  Identifying incoming messages containing a possible block and extract it from the message.
- Does 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. Managing this transport layer account is the job of the blending layer.

There is no specification on the housekeeping of the blending layer, as this is specific to the requirements of the account owner. We do, however, recommend handling messages precisely as if they were on an account handled by a human unless the receiving account appears to be a machine account.

The blending is currently achieved by merging the *VortexMessage* using either F5 as described in [**f5**] or by plain blending, which is a binary embedding. For both embeddings we currently need jpeg images included in the SMTP message.

Processing a message received from the transport layer
We define the blending layer to work as follows when receiving messages:

1. Logging arrival time on the transport layer.
2. Extracting possible *VortexMessage*.
3. Applying decryption on a suspected header block of *VortexMessage*.
4. Identifying the header block as valid by querying the accounting layer.
5. Extracting and decrypt subsequent blocks.
6. Passing extracted blocks and information to the routing layer.

A more accurate and precise outline may be found in **??**.

Processing a message received from the routing layer
We define the blending layer to work as follows for sending messages:

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

For more details regarding the exact sequence and implementation decisions, refer to **??**.

Credible content creation for the transport layer
One of the most demanding tasks of the blending layer is to create transport protocol messages. In [**oakland2013-parrot**], **oakland2013-parrot** expresses that it is easy for a human to determine decoy traffic as the content is easily identifiable as generated content. While this may be 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 the 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 on F5 [**f5**] as a steganographic algorithm, which attracted many researchers. The original F5 implementation had a detectable issue with artifacts [**F5broken**] caused by the recompression of the image. This issue occurred only due to a problem in the reference implementation, and the researchers have provided a corrected reference implementation without the weakness.

We searched 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 strength is its efficiency. This algorithm is, however, only suitable for public nodes matching up to an observing adversary (as defined in **??**). 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 be suspect. This would effectively render the Dead Parrot problem as described in [**oakland2013-parrot**] ineffective.

13.2.4.2 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 **??**). These blocks are stored in a suitable list within the workspace of the eID identified by the header block.

We refer to the message processing as "routing" as it is more than just forwarding. While processing a message we may split, or reassemble a message and process complex operations

on parts of it such as adding a redundancy operation or operations such as “onion routing” or “garlic routing”.

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

$$\mathbf{ROUTING_o} = \langle [\mathbf{ROUTINGCOMBO}]*, \mathbf{replyBlock}, mapping* \rangle \tag{13.1}$$

$$\mathbf{ROUTINGCOMBO} = \langle processIntervall, K_{peerN+1}, recipient, \mathbf{nextMP}, \mathbf{nextHP}, \mathbf{nextHEADER}, \mathbf{nextROUTING}, \mathbf{assemblyInstructions} \rangle \tag{13.2}$$

$$\mathbf{PL} = \langle \text{payload octets} \rangle * \tag{13.3}$$

$$\mathbf{nextMP} = E^{K^1_{host_{o+1}}} \left(K_{peer_{o+1}} \right) \tag{13.4}$$

$$\mathbf{nextHP} = E^{K^1_{host_{o+1}}} \left(K_{sender_{o+1}} \right) \tag{13.5}$$

$$\mathbf{nextHEADER} = E^{K_{sender_o}} \left(\mathbf{HEADER_{o+1}} \right) \tag{13.6}$$

$$\mathbf{nextROUTING} = E^{K_{sender_o}} \left(\mathbf{ROUTING_{o+1}} \right) \tag{13.7}$$

$$\mathbf{operations} = \langle \text{list of operations} \rangle \tag{13.8}$$

$$\mathbf{assembyInstructions} = \langle blendingInformation, nextHop, \langle \text{mapping operation+} \rangle \rangle \tag{13.9}$$

Figure 13.3: Simplified representation of a routing block.

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 of a node 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 list exceeding the required size or long-living message. Thus, the server 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 expiration 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 $\langle$**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.

The routing system created by this layer may be seen as a source routing system if one is willing to ignore that the sender of a message and the builder of the routing information are not equal.

13.2.4.3 Accounting Layer

The accounting layer tracks all information required and assigned to ephemeral identities (eID). It is queried by the blending and the routing layer for the 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-lived lists. As the server decides whether to accept new identities or not, the size of this data is controllable.
- **Puzz**[]$\langle expiry, request, puzzle\rangle$
  The array **Puzz**[] is a list of unsolved 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 open quota requests or one identity creation request.
- **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 routing block is expired.

13.2.5 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 workings of the blocks and provide reasoning why the block has been built as it is.

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

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 rest of the message. Each PREFIX block contains a symmetrical key 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 contains three additional 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 is nowhere reused. It is only known by the two peer hosts $host_o$ and

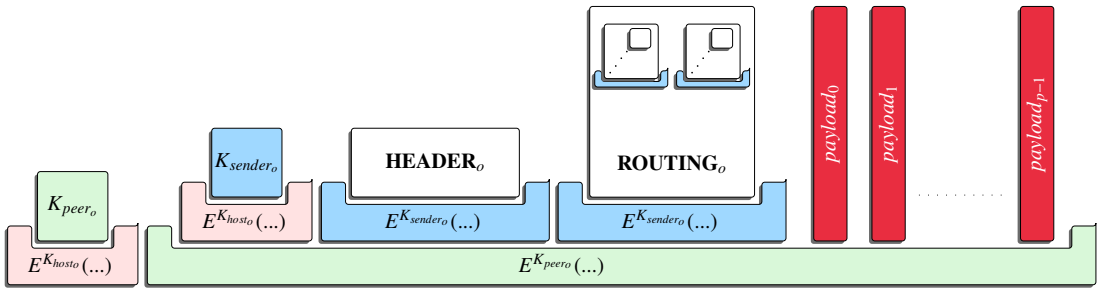

Figure 13.4: Simplified message outline visually and in math.

$host_{o-1}$, and the routing block builder (RBB). More importantly, $host_{o-1}$ does not need to know the host key of $host_o$ (K^{host_o}). Therefore, relaying a message to $host_o$ does not enable $host_{o-1}$ communication 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 **??**. Basically, we just added the routing blocks and payload to the respective workspace and waited for the routing layer to process the information.

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

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 within an eID. The $replyBlock$ provides a possibility to contact the original sender of the message without knowing him. It is only a routing block with instructions on how to prepare 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 have the payload blocks. These blocks are simply added to the eIDs workspace according to the operations included in the message.

The routing and header blocks 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 glance but is, in fact, a very important feature. Without this key, any independent observer with knowledge about the blending capabilities of a receiving node may…

- More easily identify the block structure.
  This statement remains regardless of whether ASN.1 or length prefixed structures are used. If the structure of a *VortexMessage* is easily identified, the messages may be logged or dropped.

$$\mathbf{VORTEXMESSAGE} = \langle \mathbf{MP}^{K^{-1}_{host_o}}, INNERMESSAGE \rangle \tag{13.10}$$

$$\mathbf{INNERMESSAGE} = \langle \mathbf{CP}^{K^{-1}_{host_o}}, \mathbf{H}^{K_{sender_o}}, E^{K^{-1}_{sender_o}}\left(H\left(\mathbf{HEADER}\right)\right), \left[\mathbf{R}^{K_{senderN}}\right], [\mathbf{PL}]* \rangle^{K_{peerN}} \tag{13.11}$$

$$\mathbf{MP}^{K^{-1}_{hostN}} = E^{K^{-1}_{hostN}}\left(\mathbf{PREFIX}\langle K_{peerN}\rangle\right) \tag{13.12}$$

$$\mathbf{HP}^{K^{-1}_{hostN}} = E^{K^{-1}_{hostN}}\left(\mathbf{HPREFIX}\langle K_{senderN}\rangle\right) \tag{13.13}$$

$$\mathbf{H}^{K_{senderN}} = E^{K_{senderN}}\left(\mathbf{HEADER}\right) \tag{13.14}$$

$$\mathbf{HEADER} = \langle K^1_{senderN}, serial, maxReplays, validity, [requests, requestRoutingBlock], [puzzleIdentifier, proofOfWork] \rangle \tag{13.15}$$

$$\mathbf{R}^{K_{senderN}} = E^{K_{senderN}}\left(\mathbf{ROUTING}\right) \tag{13.16}$$

$$\mathbf{ROUTING} = \langle [\mathbf{ROUTINGCOMBO}]*, forwardSecret, \mathbf{replyBlock}, \mathbf{operations} \rangle \tag{13.17}$$

$$\mathbf{ROUTINGCOMBO} = \langle processIntervall, K_{peerN+1}, recipient, \mathbf{nextMP}, \mathbf{nextHP}, \mathbf{nextHEADER}, \mathbf{nextROUTING}, \mathbf{assemblyInstructions} \rangle \tag{13.18}$$

$$\mathbf{nextMP} = E^{K^1_{host_{o+1}}}\left(K_{peer_{o+1}}\right) \tag{13.19}$$

$$\mathbf{nextHP} = E^{K^1_{host_{o+1}}}\left(K_{sender_{o+1}}\right) \tag{13.20}$$

$$\mathbf{nextHEADER} = E^{K_{sender_o}}\left(\mathbf{HEADER_{o+1}}\right) \tag{13.21}$$

$$\mathbf{nextROUTING} = E^{K_{sender_o}}\left(\mathbf{ROUTING_{o+1}}\right) \tag{13.22}$$

$$\mathbf{operations} = \langle \text{list of operations} \rangle \tag{13.23}$$

$$\mathbf{assembyInstructions} = \langle blendingInformation, nextHop, \langle \text{list of mapping operations} \rangle \rangle \tag{13.24}$$

$$\mathbf{PL} = \langle \text{payload octets} \rangle* \tag{13.25}$$

$$\tag{13.26}$$

Figure 13.5: Detailed representation of a *VortexMessage*.

- Identify the routing block size.
  The value of this information is 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.

13.2.5.1 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.

From an outside perspecive, the whole *VortexMessage* resembles a structureless data blob with a maximum of entropy caused by the encryption employed.

This is intentional and by design. Plain embedding also uses a method of splitting, which allows a message block to be embedded in chunks in the 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, much care has been applied. Any random sequence of bytes may be interpreted as valid chunking. For more exact implementation details on chunking, see **??**.

13.2.5.2 Message Structure Related to Information Leaking

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

- The 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 for communicating with the suspected peer node.
- Any information related to message size, content, or recipient.

13.2.6 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 related padding and spitting operation, and the split and merge operations.

13.2.6.1 The *addRedundancy* and *removeRedundancy* Operations

In this section, we focus on the core operation of our system. The *addRedundancy* and *removeRedundancy* allow growing message sizes in our system without allowing it to identify the decoy traffic. The Lagrange functions have been proposed in [**shamir1979share**] and were more generalized in [**mceliece1981sharing**] for sharing secrets. The general idea about all proposed schemes is to distribute pieces of information and restrict access to it so that only if a specified number of shares are captured a secret may be rebuilt. Unlike in these papers proposed, 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 distinguish message traffic from decoy traffic.

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

The operation itself is shown in **??**.

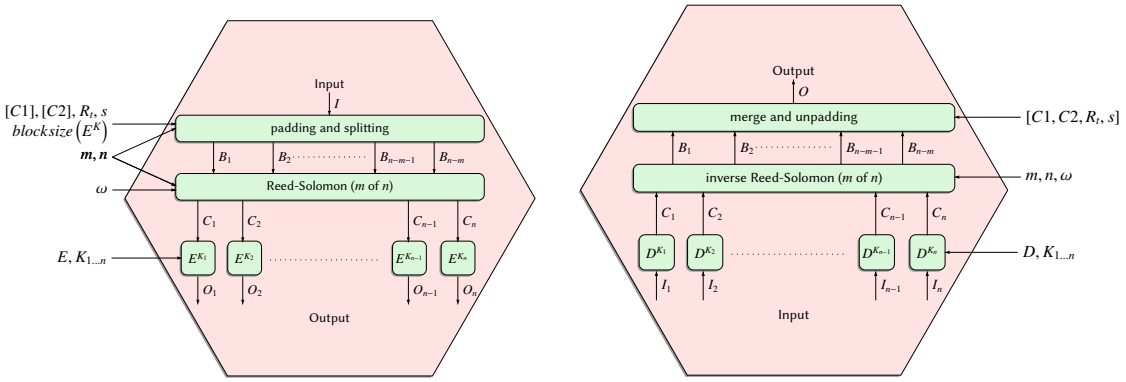

Figure 13.6: Outline of the addRedundancy and removeRedundancy operation.

It may be subdivided into the following operations:

- Padding 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.
- Applying a Reed–Solomon-operation in a given $GF(2^\omega)$ 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.
- 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:

$$
\begin{aligned}
i &= len(\mathbf{M}) && (13.27)\\
e &= lcm\left(blocksize\left(E^K\right), n\right) && (13.28)\\
l &= \left\lceil \frac{i+4+C2}{e} \right\rceil \cdot e && (13.29)\\
p &= i + \left(C1 \cdot l \pmod{\left\lfloor \frac{2^{32}-1-i}{l} \right\rfloor \cdot l}\right) && (13.30)\\
\mathbf{X} &= \langle p, \mathbf{M}, R_t(s, l-i)\rangle && (13.31)\\
&= \langle p, \mathbf{M}, R_t(s, l-(p \pmod{len(\mathbf{X})-4}))\rangle
\end{aligned}
$$

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 through a PRNG spec R_t and an initial seed value s. The message is padded up to size L. None of the resulting encrypted blocks require any padding, 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 a 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 carrying out $i = len(\mathbf{M}) = p \pmod{len(\mathbf{X}) - len(p)}$.

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 disadvantages:

- The padding is inefficient compared to simple paddings such as PKCS#7
- The padding requires an initialized PRNG to generate the padding data.
- 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 carry out 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.

$$t = n - 1 \tag{13.32}$$

$$\mathbf{A} = vec2mat\left(\mathbf{X}, \frac{len\left(\mathbf{X}\right)}{m}\right) \tag{13.33}$$

$$\mathbf{V} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & \cdots & 0^{(m-1)} \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{(m-1)} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t^0 & t^1 & t^2 & \cdots & t^{(m-1)} \end{pmatrix} \tag{13.34}$$

$$\mathbf{P} = \mathbf{VA}\left(GF\left(2^{\omega}\right)\right) \tag{13.35}$$

$$\langle \mathbf{Q}_1, \ldots, \mathbf{Q_n} \rangle = row2vec(P) \tag{13.36}$$

$$R_i = E^{K_i}\left(Q_i\right) \tag{13.37}$$

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 **??** 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 required on the block-level. The key used to encrypt the single blocks must not be equivalent. Equivalent keys have the side effect of encrypting equal blocks into the same ciphertext. We observed faint but statistically relevant reminders of the unencrypted graphs when treating the same block with the same key and different redundancy parameters. Details about this analysis are available in **??**.

13.2.6.2 The *encrypt* and *decrypt* Operations

The *encrypt* and *decrypt* operations as shown in **??** 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 encryption.

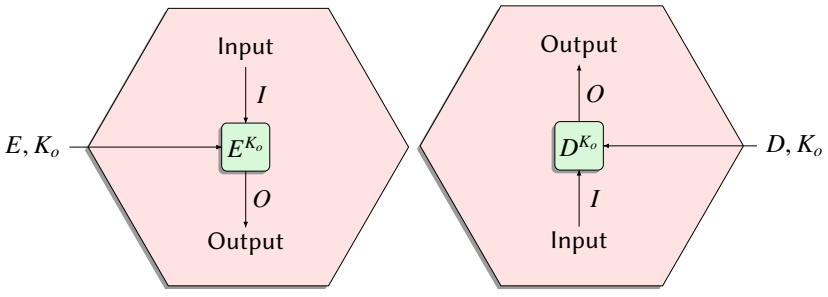

Figure 13.7: Outline of the encrypt and decrypt operation.

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 ??.

13.2.6.3 The *mergePayload* and *splitPayload* operation

The *splitPayload* operation shown in ?? 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 describing the size of the first chunk. If the fraction is "1.0", then the whole payload is transferred to the second target chunk.

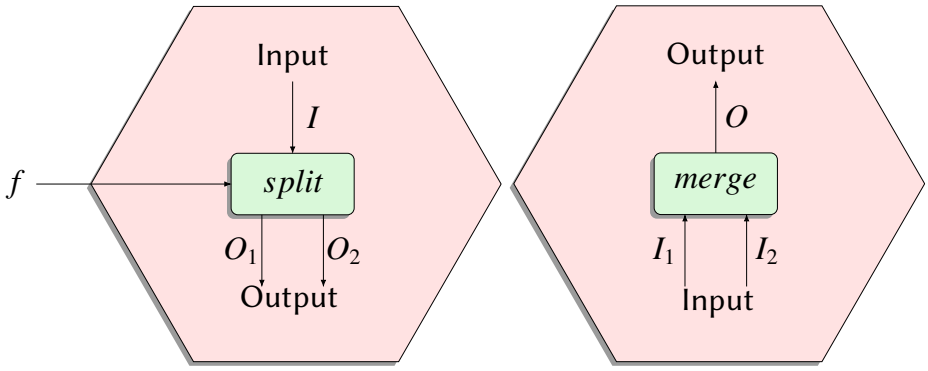

Figure 13.8: Outline of the splitPayload and mergePayload operation.

If $len(pb_1)$ expresses the size of a payload block called pb_1 in bytes, then the two resulting blocks of the splitPayload operation pb_2 and pb_3 have to adhere the following rules:

$$split(f, pb_1) = \langle pb_1, pb_2 \rangle \tag{13.38}$$

$$pb_1.startsWith(pb_2) \tag{13.39}$$

$$pb_1.endsWith(pb_3) \tag{13.40}$$

$$len(pb_2) = floor(len(pb_1) \cdot f) \tag{13.41}$$

$$len(pb_1) = len(pb_2) + len(pb_3) \tag{13.42}$$

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 has to adhere the following rules:

$$merge(pb_1, pb_2) = pb_3 \tag{13.43}$$
$$pb_3.startsWith(pb_1) \tag{13.44}$$
$$pb_3.endsWith(pb_2) \tag{13.45}$$
$$len(pb_3) = len(pb_1) + len(pb_2) \tag{13.46}$$

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 **??**.

13.3 Summary

The *MessageVortex*-Protocol is split into the four layers: "Transport" (a common Internet standard protocol), "Blending" (extracting and embedding *VortexMessages*), "Routing" (re-assembling messages according to received instructions), and "Accounting" (tracks all stored data and discards expired information).

All nodes are realized in decentralized devices such as computers or mobile phones. Messages are hidden with either plain embedding or F5 in the transport layer message. The routing layer processes messages by applying operations to them. 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 the appropriate length may be valid.

The routing itself is achieved in a temporarily allocated storage called "workspace", which is tied to an ephemeral identity (eID) represented by an asymmetric key pair. To obtain 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.

Implementation

***

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

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

This section focuses on the details of our reference implementation in Java. In ??, we explain the selection of algorithms used by the protocol in general. We then focus on the implementation of the transport (??), blending (??), routing (??), and accounting (??) layers. We then look at the usability (??) and efficiency (??). Aspects relevant to the implementations' usability and efficiency are covered in ?? and ??.

14 Algorithms, Encodings, and Protocols Selection

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) based on different mathematical problems

14.1 Encoding Scheme

As encoding scheme, we specified ASN.1 [**dis19878824**]. It is more compact than the initially selected XML-Standard and is very common in telecommunication and encryption (e.g., the representation of X509 is in ASN.1). To maintain interoperability, we choose DER-encoding as it has precisely one possible representation for every value. Such a strict definition of encoding is important when signing or solving puzzles in our case and is required to diagnose message paths.

On the downside, ASN-1-encoding is, unlike XML, unreadable by humans. As we hide the messages, we considered this a minor flaw, as we need to have a constantly-extracting program to see the messages' content.

14.2 Cipher Selection

In this protocol, many encryption and hashing algorithms have to be used. In the following, we explain the choice of these algorithms.

We decided to define fixed key sizes for symmetric ciphers as we chose block ciphers. For asymmetric ciphers, we encode the key length in the asymmetric ciphers' parameters section. Due to their mathematical differences, they are frequently flexible in their parameters such as key or block sizes.

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

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

SymmetricKey ::= SEQUENCE {
  keyType        SymmetricAlgorithm,
  parameter      AlgParameters,
  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)
}
AlgParameters ::= 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
}
```

Figure 14.1: Definition of the structures related to ciphers.

From the requirements side, we adhere to 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 origin.

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

Third, due to the onion-like design described in this document, our protocol should avoid asymmetric encryption in favor of symmetric encryption to minimize losses 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 length is incorporated into the definition. If not, the key size is handled as a parameter.

The key sizes were chosen so that the key types form tuples of approximately equal strength. The support of Camellia192 and AES192 was defined as optional. However, as they are wildly common in implementations, they have already been standardized as they build a possibility to enhance security in the future.

From these criteria, 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 associating the used algorithms with a strength. This list, however, should only serve the purpose of selecting algorithms for people without cryptological know-how.

14.3 Mode Selections

We evaluated the most common cipher modes for suitability. For *MessageVortex*, we focused on modes with parallelizable, random access modes that do not authenticate. In addition to the characteristics mentioned before, the main focus was on whether there is a reasonably tested open implementation in Java.

```
1  CipherMode ::= ENUMERATED {
2    cbc            (10000), -- required
3    ctr            (10001), -- required
4    ccm            (10002), -- optional support
5    gcm            (10003), -- optional support
6    ocb            (10004), -- optional support
7    ofb            (10005), -- optional support
8    xts            (10006), -- optional support
9    none           (10100)  -- required
10 }
```

Figure 14.2: Enumeration definition of modes in ASN.1 with support requirements.

Figure **??** shows the selected paddings and their requirement level.

Very important was that we quite often re-encrypt already encrypted content. In theory, a partially broken mode is much less problematic when encrypting already random content. However, these flaws are obvious to a crypto savvy person but are not common knowledge. By always choosing the same mode and only using onionizing schemes, the flaw remains. To avoid this, we eradicated modes such as ECB despite the fact that their simplicity could have been a gain for the protocol if properly handled.

- 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 plaintext 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 (Cipher Block Chaining)
  CBC extends the encryption by XORing 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 disadvantage, 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 Cipher Block Chaining)
  CBC extends the encryption by XORing, not the ciphertext but a XOR result of ciphertext and plaintext. This modification denies parallel decryption and random access compared to CBC.

- EAX
  We rejected as the mode was analyzed and broken in **minematsu2013attacks** in [**minematsu2013attacks**].

- CFB (Cipher Feedback) CFB is specified in [**dworkin2001recommendation**] and works precisely as CBC with the difference that the plaintext is XORed and the initialization vector, or the preceding cipher result is encrypted. CFB does not support parallel encryption as the ciphertext input from the prior operation is required for an encryption round. CFB does however allow parallel decryption and random access.

- OFB
  [**dworkin2001recommendation**] specifies OFB and works precisely as CFB except for the fact that not the ciphertext result is taken as feedback but the result of the encryption before XORing the plaintext. This denies parallel encryption and decryption, as well as random access.

- OCB (Offset Codebook Mode)
  This mode was first proposed in [**rogaway2003ocb**] and later specified in [**krovetz-ocb-04**]. 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 [**lipmaa2000ctr**] and is a mixture between OFB and CBC. A nonce concatenated with a counter incrementing on every block is encrypted and then XORed with the plaintext. 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 XORed 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 [**rfc3610**]. It allows for padding and authenticating 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 [**mcgrew2004galois**], and is related to CTR but has some major differences. The nonce is not used (just the counter starting with value 1). An authentication token $auth$ is hashed with H_{GFmult} and then XORed with the first cipher block to authenticate the encryption. All subsequent cipher blocks are XORed 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 was analyzed security-wise in **mcgrew2004security** and showed no weaknesses in the studied fields [**mcgrew2004security**].

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 [**Martin2010**]. 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 [**Halevi:2003**] **Halevi:2003** 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 [**tschorsch:translayeranon**]. This mode shares the same properties as EBC but without the same cleartext block's weakness resulting in the same ciphertext. Similar to XEX, it requires a tweak instead of an IV.

We decided to mainly use CBC. However, most of the implementations are available and lightweight. We therefore were not as restrictive as usual when defining a minimal set.

14.4 Padding Selection

A plain textstream 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 the paddings outlined in **??** to be supported.

```
1  CipherPadding ::= ENUMERATED {
2    none            (10200), -- required
3    pkcs1           (10201), -- required
4    rsaesOaep       (10202), -- optional support
5    oaepSha256Mgf1 (10203), -- optional support
6    pkcs7           (10301), -- required
7    ap              (10221)  -- required
8  }
```

Figure 14.3: Enumeration definition of paddings in ASN.1 with support requirements.

14.4.1 RSAES-PKCS1-v1_5 and RSAES-OAEP

This padding is the older one 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 is the newer of the two padding standards.

14.4.2 PKCS7

This padding is the standard used in many places when applying symmetric encryption in an up to 256 bit 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 available data at the end of the block. All other bytes are defined but not needed.

14.4.3 OAEP with SHA and MGF1 Padding

This padding is closely related to RSAES-OAEP padding. However, the hash size is larger, 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 essential to note that the last block's payload shrinks to $keySizeInBits/8 - 2 - MacSize/4$.

In our approach, we chose to allow these four paddings. The allowed SHA sizes match the allowed MAC sizes chosen above. It is important to note that padding uses space at the end of a stream. Since we are always using one block for signing, we have to ensure that the chosen signing MAC and the bytes required for padding do not exceed the asymmetric encryption's key size. 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.

14.4.4 Honorable Mention: A Padding for *redundancy* Operations

We introduced an additional type of padding not related to these paddings. For the *addRedundancy*, we required 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 pad more than just one block.

This padding is the only one for the *addRedundancy* and *removeRedundancy* operations. A specification may be found in ??.

14.4.5 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 this part in depth as it seemed irrelevant. The only criterion was that it had to create content indistinguishable from an encrypted message. This criterion arose as we used it for invisibly padding an already encrypted message.

The PRNG used for our implementation is an XORshift+ generator. It is based on the XSadd PRNG [**marsaglia2003xorshift**] 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:

$$\begin{aligned} x &= s_0 && (14.1)\\ s_0 &= s_1 && (14.2)\\ x &= x \oplus (x \ll 23) && (14.3)\\ s_1 &= x \oplus s_1 \oplus (x \gg 17) \oplus (s_1 \gg 26) && (14.4)\\ nextNumber &= s_1 + s_0 && (14.5) \end{aligned}$$

We chose 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 introduced everywhere at least one alternate algorithm that may be used to replace a broken algorithm.

To have a second choice for the PRNG, we define the Blum–Micali PRNG as described in [**blum1984generate**]. This PRNG is cryptographically secure and is defined as follows:

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

$$x_{i+1} = g^{x_i} \mod p \quad (14.6)$$

This PRNG requires significantly more calculation power than the XORshift+ PRNG. On the positive side, the PRNG is well researched, and we have found no weaknesses documented in academia.

14.5 Transport Layer Protocol Selection

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

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

All sections are structured the same. 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 concisely summarize the protocol's suitability as a transport layer. The findings of this section are listed in ??. The list here does not reflect the quality or maturity of the protocols. It is a simple analysis of suitability as a transport layer.

14.5.1 Applied Criteria

- Widely Adopted (Ct1)
  The more widely-adopted and used a protocol is, the more diffitcult 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 the 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 search for inherently reliable protocols.

- Symmetrically 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.

14.5.2 Analyzed Protocols

We were unable find a comprehensive list of protocols being used within the Internet and their bandwidth consumption. A weak reference is [**zhou2011examining**]. This weakness is founded because traffic in this report is classified among two criteria: Know server or known port. According to the report, streaming services consume more than 60% of the Internet download bandwidth. The focus of the report lies on the bandwidth-intense figures. However, leaving 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).

We first collected a list of all common Internet messaging protocols (synchronous and asynchronous in lacking such material). We then added 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 were discarded as we consider them as outdated:

- MTP [**rfc780**] (obsoleted by SMTP)
- NNTP [**rfc3977**] (outdated and has only a small usage according to [**kim2010today**])

We furthermore discarded all RPC-related protocols as they would, by definition, violate the symmetry criteria (Ct3: Symmetrically Built).

14.5.3 Analysis

14.5.3.1 HTTP

The HTTP protocol allows message transfer from and to a server and is specified in RFC2616 [**rfc2616**]. It is not suitable as a communication protocol for messages due to the lack of notifications. Some extensions 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. Port 443 is equivalent to port 80, except that it has a wrapping encryption layer (usuall TLS). The incoming connects (requests) must offer a header part and may contain a body part 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 cleartext protocols that are human-readable (except for the data part, which might contain binary data). The HTTP/2 [**rfc7540**] protocol is using the same ports and default behavior. Unlike HTTP/0.9-HTTP/1.1, it is not a cleartext but encodes headers and bodies in binary form.

To be a valid candidate as storage, unauthenticated WebDAV support, as specified in [**rfc4918**], must be assumed.

The protocol satisfies 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 symmetrical. 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 behavior would however most likely be detectable and thus no longer be censorship-resistant.

14.5.3.2 FTP

FTP is defined in RFC959 [**rfc959**]. 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” only has upload rights. He 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 disadvantages. At first, handling FTP that way is very uncommon and usually requires an own dedicated infrastructure. Such behavior would make the protocol possibly detectable again. Secondly, passwords are always sent in the clear within FTP. Encryption as a wrapping layer (FTPS) is not common, and SFTP (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.

Similar to HTTP, a disadvantage of FTP in terms of a message transport protocol is that this protocol is not symmetrical. 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).

14.5.3.3 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 many 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 asymmetrical. 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. Furthermore, the protocol violates Ct2 (Ct2: Reliable) as it is based on UDP without any additional error correction.

14.5.3.4 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 5.0 [**mqtt**].

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 to receive messages. Such behavior makes it very difficult to use it in a system without centralistic infrastructure and static routes between senders and recipients.

The protocol does satisfy the second criterion (Ct2: Reliable). It is in the end-user area (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).

14.5.3.5 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 [**amqp**].

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

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

The protocol partially satisfies 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).

14.5.3.6 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a communication protocol that is primarily destined for M2M communication. It is defined in RFC7252 [**rfc7252**]. 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 of a message transport protocol is that this protocol is not (like HTTP) symmetrical. 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.

14.5.3.7 Web Application Messaging Protocol (WAMP)

WAMP is a web-socket-based protocol destined to enable M2M communication. Similar to MQTT, the protocol is publish/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 easily.

14.5.3.8 XMPP (Jabber)

XMPP (originally named Jabber) is a synchronous message protocol used in the Internet. It is specified in the documents RFC6120 [**rfc6120**], RFC6121 [**rfc6121**], RFC3922 [**rfc3922**], and RFC3923 [**rfc3923**]. The protocol is a very advanced chat protocol featuring numerous

levels of security including end-to-end signing and object encryption [**rfc3923**]. There is also a stream initiation extension for transferring files between endpoints [**xep0096**].

It has generic routing capabilities spanning between known and unknown servers. The protocol offers a message retrieval mechanism for offline messages similar to POP [**xep0013**].

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

14.5.3.9 SMTP

The SMTP protocol is currently specified in [**rfc5321**]. It specifies a method of reliably delivering 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, containing a sender (one) and one or more recipients encoded 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, which are: the header (which contains 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 content lines 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 header's exact outline is specified in [**rfc5322**] and separated with a blank line from the body.

RFC5321 [**rfc5321**] 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 [**rfc2045**] and [**rfc2046**]. MIME allows sending messages in multiple representations (alternates) and attaching 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. Furthermore, the protocol is 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 consiered fulfilled.

SMTP is considered a strong candidate as a transport layer.

14.5.3.10 SMS and MMS

Telephone companies introduced the SMS capability in the SS7 protocol. This protocol allows the message transfer of messages no larger than 144 characters. Due to this restriction in size, it is unlikely to be suitable for this type of communication. The keys required for our protocol are already similarly sized, 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.

14.5.4 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. In consequence, M2M protocols typically use 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. Additionally, at least the data sources are typically limited in number. Such constraints make those protocols unsuitable again.

We can clearly state that according to the criteria, only a few protocols are suitable. Table ?? on page ?? 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 14.1: 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.

15 Transport Layer Implementation

15.1 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.

15.2 Implementation of an Email Transport Layer

Email supports a conglomerate of protocols. Looking at the client-side, we will find that an email is sent with an authenticated SMTP connection. The SMTP connection is somewhat

different than than the connections used to send emails to a destination. First of all, the client port was shifted in the past to a specific submission port (SMTPS: Port 465; Submission: Port 587). Such submission 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 used for automated processing of mails, but presently no longer adequate, as the same user owns multiple devices. IMAPv4 offers to organize emails 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 have done the following: Organized 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 **??**. The housekeeping on the account used for *MessageVortex* is caried out automatically and in a sensible way, comparable to a human (e.g., handling drafts, sent, and trash bin folders sensibly and keeping all mails in a flat structure by deleting old emails from time to time). The proxy transparently merges the mails from the regular and the *MessageVortex* account. This proxy mechanism keeps the messages apart on the transport layer but offers a unified look at the data.

We were unable to find any scientific data regarding what type of traffic or attachment is common on the Internet. Therefore, we analyzed the email logs (SMTP) of a mail provider. We scanned $500K$ 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 **??**.

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 15.1: 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.

We worked with F5 blending into jpeg images for our implementation, as this choice seemed to undermine credible content based on **??**.

In our current implementation, the housekeeping part was skipped. Instead, we just fetched the newly arrived messages and transferred them to local storage. The email presented to the client is provided by a local IMAP server. The persistence of these messages is not yet implemented.

15.3 Implementation of an XMPP Transport Layer

The XMPP protocol (formerly called Jabber, as specified in [**rfc6120**]) is natively not capable of transferring anything but text messages. Unlike email, XMPP is capable of true end-to-end signing and object encryption without solving the initial trust problem. While we may use 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 **??** requirement. We decided to use the extension defined in [**xep0231**] to transfer our messages, as it is simple and reliable.

To transfer a *VortexMessage*, we could embed 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 of an XMPP message relies on XML.

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

Name	Status (as of 06-2020)	Purpose
xep0047 [**xep0047**]	Final Standard	Allows sending chunked, base64 encoded data within the Jabber connections.
xep0066 [**xep0066**]	Draft Standard	Allows sending URIs of remotely hosted binary data.
xep0096 [**xep0096**]	Depreciated (ref. XEP-0234)	Improvement of [**xep0066**] allowing to send metadata and alternative URIs
xep0135 [**xep0135**]	Deferred (inactive)	Inband or out-of-band file discovery and referral service. May be used in conjunction with FTP, HTTP, SCP, or [**xep0096**].
xep0231 [**xep0231**]	Draft Standard	Allows sending inband small unchunked files and referring within the message similarly to [**rfc2397**].
xep0234 [**xep0234**]	Deferred (inactive)	Based on [**xep0166**] allowing out-of-band content negotiation of complex data streams

Table 15.2: Overview of XEPs related to transporting binary data.

Relevant documents have either reached the level standard, draft standard, or were deferred due to inactivity. We used "**xep0231**" [**xep0231**] for our protocol. It is simple to implement as a transport layer, used in many clients (e.g., Prosody, Pigdin, or CoyIM), and already a draft standard minimizing the risk of using later deprecated technology. As this XEP is a client-only XEP, a node may use any XMPP server regardless of any additional support for XEP-0135.

Embedding works 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 messages analog to the messages generated in the case of email. Some adoptions in terms of texts might be advisable.

15.4 Distributed Configuration and Runtime Store of Processing Content

A distributed storage is advisable if it works as a reliable service. This is why we defined ASN.1 structures for all elements kept in memory as shown in listing ??. 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 suitable 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.
- The draft folder contains one draft message with the current, short-living configuration.
- A long-living configuration is written to draft and then moved to the “sent items” folder.
- A configuration is first fetched from the drafts folder, then the first config object of the “sent items” folder is fetched.
- All items in the sent folder are deleted after a defined timespan (e.g., 30 days). Items not yet expired are rewritten into a new config object into the sent folder before deletion.

16 Blending Layer Implementation

16.1 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 ??.

```
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. A byte stream is extracted in both cases consisting of a prefix block containing the peer key K_{peer_o} immediately followed by the symmetrically encrypted InnerMessageBlock as described in listing ??. The string is not necessarily correctly terminated. The presence of a valid PrefixBlock signals an existing *VortexMessage* on the blending layer. That way, we ensure that the size of a potential message leaks its presence. To detect the presence of a *VortexMessage*, the host's private key $K^{-1}_{host_o}$ for decoding the message is required.

```
1  PrefixBlock ::= SEQUENCE {
2    version          [0] INTEGER OPTIONAL,
3    key              [2] SymmetricKey
4  }
5
6  InnerMessageBlock ::= SEQUENCE {
7    padding    OCTET STRING,
8    prefix     CHOICE {
9        plain            [11011] PrefixBlock,
10       -- contains prefix encrypted with receivers
11       -- public key
12       encrypted        [11012] OCTET STRING
13   },
14   header  CHOICE  {
15     -- debug/internal use only
16     plain      [11021] HeaderBlock,
17     -- contains encrypted identity block
18       encyrpted [11022] OCTET STRING
19   },
20   -- contains signature of Identity [as stored in
21   -- HeaderBlock; signed unencrypted HeaderBlock without
22   -- Tag]
23   identitySignature OCTET STRING,
24   -- contains routing information (next hop) for the
25   -- payloads
26   routing      [11001] CHOICE {
27     plain      [11031] RoutingBlock,
28     -- contains encrypted routing block
29     encyrpted [11032] OCTET STRING
30   },
31   -- contains the actual payload
32   payload      SEQUENCE (SIZE (0..maxPayloadBlks))
33                OF OCTET STRING
34 }
```

Listing 3: Definition of the outer message blocks.

16.1.1 Extraction of the Blended Message

In this section, we describe the extraction of a *VortexMessage* by the blending layer. We describe plain embedding which allows a detectable yet unreadable message, including the chunking applied to minimize detection. Furthermore, we describe the more elaborated method of using F5 blending, which results in undetectable messages at the cost of roughly eight times higher protocol overhead.

16.1.2 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 trans-

feral.

The *plainEmbedding* is a simple embedding replacing parts of the original file with the content of the *VortexMessage*. To maintain the header information, we introduced an offset as a set of fixed values. Plain embedding is easily detectable. While offset and chunking may allow us to maintain a valid file structure, the file's original content is normally destroyed. We use plain embedding mainly for our experiments. We used a specialized blending layer for better readability 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 blending layer's current implementation employing plain embedding is not suitable for production use as the messages remain identifiable or suspicious.

16.1.2.1 Chunking of Plain-Embedded Messages

In this section, we describe the chunked embedding into plain messages. Chunking is carried out by pre-pending two numeric values to a data chunk. The first number (modulo the remaining number of bytes of the file) reflects the chunk's size 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 reaching 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 full eight bits are used as the most significant bit in that case. All bits collected together are interpreted as 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 has no terminator and may surpass the file end.

The byte layout is chosen so that any byte sequence, from two to eight bytes, forms a valid chunk header. The lack of termination guarantees that no information leaks through the interpretation of any header.

Table **??** shows some valid chunking header bytes and their interpretation as offset value (without the modulo). Listing **??** shows an implementation of the algorithm.

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

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

When plain-embedding messages, we have the problem that most of the files have recurring logical structures. Such structures should not be broken. Broken files raise suspicion as

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 16.1: Example interpretation of bytes in offset values.

they are no longer displayable. Thus, we have to avoid breaking logical file structures and concentrate on structureless portions of the file when embedding.

16.1.3 Implementation of F5 Blending

In this section, we introduce the implementation of F5 blending. It is a more suitable blending than the rather simple *plainBlending* discussed in the previous section. At the same time, F5 is very old (**f5** and renains unbroken. In the reference implementation of F5 was a detectable unintentional double compression [**steganalysisf5**]. The authors of the reference implementation fixed this issue [**F5broken**], and we were unable to find newer breaches. Newer derivates, such as nsF5 [**fridrich2007statistically**] or MSET [**hosseini2015modification**], were proposed. However, we did 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 [**f5**], 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 unable to differentiate random output from a *VortexMessage*in our analysis. Only decoding with the host key $K^{-1}_{host_o}$ resulted in detecting a *VortexMessage*.

As shown in listing **??**, we publish this password and keep detection to the decoding part of our blending layer. In theory, we could have kept this password specific to the eID. However, this would 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.

16.2 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*. Verifying the identity signature (which may be achieved 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 identity status (temporary or established), further processing by the routing layer is acknowledged. For an overarching description of the whole message, processing see ??.

16.3 Decoy Content Generation

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

Using F5 or similar mechanisms for blending, we decided to ensure that our content does not rquire to pass a Turing test. Normal email conversations are two-way and have many properties such as references to previous messages and similar contexts. In order not to fall into such traps, we use common machine-generated one-way messages with generated images. Examples of 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 machine-generated for obvious reasons.

To make it more difficult for an attacker to identify the context of messages, the sending address on the transport media should not be equal to the receiving address. this makes the generation of interaction graphs much more difficult, as we will see in ??.

17 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. Furthermore, we shed light on some decisions specific to our implementation, such as encoding formats or message layout.

17.1 ASN.1 DER-Encoding Scheme for *VortexMessages*

Originally, we implemented the protocol as XML-encoded messages. This encoding however had several flaws. First, the huge amount of encrypted data within the document made the messages bulky and, at the same time, lose 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, 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 Laughs) or information-stealing attacks (e.g., XXE attacks). Furthermore, XML structures are difficult to sign and have many possible ways of layouting data.

To counter these disadvantages, we re-implemented our client with ASN.1-based DER-encoding. This type of encoding fits well with encrypted structures and is commonly used for related tasks such as key storage or signing messages 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 **??**.

ASN.1 offers three common encoding schemes:

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

As 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.

17.2 The Processing of Messages

In this section, 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.

17.2.1 Workspace Layout

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

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 17.1: Workspace layout of IDs.

17.2.2 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.

The first step of processing is the extraction of the identity. The identity can be found in the header block (see listing **??** *identityKey*) and then verified with the signature *identitySignature* (listing **??**)

If verification is successful, the message is authenticated but not necessarily ready for further processing. Unless the header contains an identity creation request, the next step is then the

```
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 OPTIONAL,
  -- 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
                            (SIZE (1..maxNumOfReplays))
                        OF SymmetricKey OPTIONAL,
  -- contains the next recipient
  recipient       [402] BlendingSpec,
  -- PrefixBlock encrypted with message key
  mPrefix         [403] SEQUENCE
                            (SIZE (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.

authorization for further processing. For proper authentication, the following preconditions must be met:

- Message must be outside a replay blocking interval
- The identity is not temporary (??)

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 routing block operations are added to the workspace, and the mapping operations remain in the routing combo.

17.2.3 Processing of Outgoing Messages

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

The sending of a message is triggered by a routing block in the workspace, as shown in ??. 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 (see **??** and **??**). This block is DER-encoded and then encrypted 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 *blendingS pecc*.

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.

17.2.4 Implementation of Operations

In this section, we focus on the implemented operations. The operations outlined in **??** were implemented in exactly 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 the platform and implementation specifics, which are outlined further.

For our implementation, we used a HashMap to keep a list of all operations. The key of the HashMap is the output ID of the resulting operations. Instead of proactively executing all operations to obtain 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 respective 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 necessary when defining the system of specialized IDs as outlined in **??**. This usage of specialized workspace IDs makes the mapping of values from one ID to another one a necessity. While theoretically feasible 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 [**IEEE754**]. This allows us to have 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 that arose in practice was that applying a Galoise field (GF) in the *addRedundancy* and *removeRedundancy* operations different to 8 or 16 cause practical problems due to their resulting sizes. To simplify applying the transformation for the average computer working with 8 bits per byte only, we added a possibility for the node to signal which sizes of GFs are supported. This enables an implementation to only focus on $GF(2^8)$ and $GF(2^{16})$.

A GF of size not equal to 8 or 16 requires the system to realign the data before processing, then applying the GF operations and converting it back to realign with 8-bit boundaries.

17.3 Handling Requests

In this section, we focus on handling 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 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 is provided in *requestReplyBlock*.

The identified requests are shown in listing **??**. The tagging of the requests is necessary to identify the request provided.

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

Listing 6: Definition of a request.

The routing blocks for replies must differentiate from normal routing blocks as they may otherwise be misused as ordinary sending blocks. A reply block for the request should always map to payload ID 32767, whereas a reply block for a normal user (to keep sender anonymity) should always 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 **??** shows the definition of a reply. A reply is expressed in a special block. This special block contains a status of the request, which is either a success or a failure and may provide additional information such as the request's outcome.

```
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.

17.3.1 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 difficult for an adversary to flood a *VortexNode* with workspace requests to cause a denial-of-service (DoS) attack.

```
1  HeaderRequestIdentity ::= SEQUENCE {
2    period UsagePeriod
3  }
```

Listing 8: Definition of an identity request.

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

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

The 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 to not leak their existence.

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

```
1  RequirementBlock ::= CHOICE {
2    puzzle   [1] RequirementPuzzleRequired,
3    payment [2] RequirementPaymentRequired
4  }
5
6  RequirementPuzzleRequired ::= SEQUENCE {
7    -- bit sequence at beginning of hash from
8        -- the encrypted identity block
9    challenge      BIT STRING,
10   mac            MacAlgorithmSpec,
11   valid          UsagePeriod,
12   identifier     INTEGER (0..4294967295)
13 }
14
15 RequirementPaymentRequired ::= SEQUENCE {
16   account        OCTET STRING,
17   ammount        REAL,
18   currency       Currency
19 }
20
21 Currency ::= ENUMERATED {
22   btc       (8001),
23   eth       (8002),
24   zec       (8003)
25 }
```

Listing 9: Definition of a requirement.

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

- Solving a CPU-bound hash puzzle
- Paying a fee in a digital currency

The CPU-bound puzzle is a hash based. The *VortexNode* provides a bit string for the identity. The header has to be resent so 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:

- Generating puzzles in a reproducible way
  This is the more elegant way of puzzles. Instead of tracking the puzzles, we generate the hash by applying the following function $left(MAC(K^1_{identity}| <$ host secret $> | <$ date and hour $>)$, hourly complexity in bits). This method has positive and negatives sides. On the positive side, we do not need to track 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 track 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 to increase the quotas. This is not comparable to the costs for an attacker as we only have to keep a list of integers where the PoW has been solved.

- Storing random puzzles during their validity time
  This method is straightforward. It requires an entry in a table per puzzle only for the lifetime considered.

The second approach has a great disadvantage: 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). This 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 only a DoS attack), this attack is feasible with considerable effort. This is why we favor the first approach.

17.3.2 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 achieved with a *HeaderRequestReplaceIdentity* request as shown in listing ??. 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. However, such a secured identity should 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 a different (possibly unlimited) quota to the identity.

17.3.3 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.

In the case an owner wants to replace an existing routing block with a new one, it is sufficient to send an empty message to the respective eID. Within the routing block, the sender provides 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.

18 Accounting Layer Implementation

The accounting layer tracks all operations allowed to a message. In this section, we list the tasks fulfilled by the accounting layer and outline them precisely.

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 number of messages containing payload blocks to be routed. This quota is measured upon the arrival of a message (inbound only). The *bytesLeft* quota is a sizing quota and is measured outbound. This quota is applied to 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 that 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 $MAC\left(K^1_{eID}|globalSecret|dateAndHour), puzzleSizeNewIdentity\right)$. That way, a DoS attack by flooding the puzzle table is no longer feasible.

The last list is the list for replay protection *Replay*[]. This 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 typically only has 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.

In ??, 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 [**rfc2119**].

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 18.1: Requests and the applicable criteria for replies.

19 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 introduced an addressing scheme compatible with email and the support of their clients without creating any collisions with the existing email address schemes.

These schemes are discussed in the next section. Then, we address the problem of linking to user agents and transparency issues.

19.1 Addressing and Address Representations

An endpoint always requires a public key and a transport endpoint. As we have no central infrastructure, we need a defined way to exchange addresses. These addresses need to be uniquely identifiable and have to work with clients. In this section, we focus on the implementation details of such an address.

If we want to use common email or XMPP clients, we must support an address format compatible with the client but which produces no collisions with ordinary addresses. Luckily, experiments showed that clients are not very restrictive in the acceptance of addresses. Most clients required either an at sign between two letters or, additionally, at least a dot in the domain part of the address. [**rfc5321**] and [**rfc5322**] specify the format for email addresses and [**rfc6120**] 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 defined our addresses as follows.

For email:

```
1  localPart          = <local part of address>
2  domain             = <domain part of address>
3  email              = localPart "@" domain
4  keySpec            = <BASE64 encoded AsymmetricKey [DER encoded]>
5  smtpAlternateSpec  = localPart ".." keySpec ".." domain "@localhost"
6  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 ".."
domain "@localhost" [ "/" resourcePart ]
jidUrl            = "vortexxmpp://" jidAlternateSpec]
```

This allows using of a regular client to host a *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 the email is restricted to 64 bytes, whereas XMPP specifies 1024 bytes as the local part's size limit. Our experiments showed, however, that none of the clients enforce these limits.

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.

19.2 Linking to Common User Agents

From an academic perspective, the protocol linking as a proxy is easy. Real-world implementation however showed many caveats. We will focus on these problems in this section and note workarounds where possible.

When combining data on asynchronous message protocols, we always have two possibilities. Either work as a transparent proxy for a single view or combine multiple sources. Another option is always to create a local repository with the disadvantage 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 feasible for two reasons. First, we must carry out 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 this in an enterprise environment is easy, as we can 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 ([**rfc6698**, **rfc7672**]) or similar technologies. Within all protocols, analyzed certificate-based authentication is very uncommon. However, such authentication would break if we carry out a MITM.

When sending an email, we can use authenticated SMTP on the client submission ports. This may be realized either as a transparent proxy or as a store and forward solution with very few disadvantages. When working as store and forward, we have the disadvantage 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. We developed an easy workaround for this scenario: Our SMTP implementation binds on 127.0.0.1 only and accepts a dummy password. Simultaneously, we build a second connection to the provider's 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 that packs an appropriate

routing block and the plain 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 message's pseudo nature, handles the message. It completes the first encryption operation and applies then the operations to send the message to all next hops with the appropriate routing blocks.

When receiving messages by mail, things quickly become 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 without delay. As with POP3 mails are stored locally, we have no problems as the client fetches and stores the mail. Considering IMAPv4, we have a several of very relevant differences. Unlike POP3, IMAPv4 stores and organizes messages on the server. The main 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 attempts 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 unified storage and offer simultaneous access for multiple devices. The problems are numerous. We need unified storage for configurations including eIDs and workspaces. Furthermore, we need a lock to avoid concurrency issues with simultaneously 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 achieved in a credible way.

20 Efficiency-Related Implementation Details

In the following section, we focus on the storage management of *VortexNodes*. As they run on mobile and similar devices, low resource consumption is essential for our system. We mainly focus on memory and CPU consumption. Network bandwidth overhead and their related problems are discussed in **??**.

20.1 Node Storage Management

In most mobile devices, storage is very limited. This applies to the disk storage but is especially true for the RAM of such devices. Our protocol supports the minimization of storage footprints 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 concurrently required RAM is limited to the event-triggered routing blocks, respectively their trigger blocks. Listing **??** 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 relative to the time of arrival. The real validity time is formed as the intersection out of the two timings, whereas both may be omitted by definitions.

```
UsagePeriod ::= CHOICE {
  absolute [2] AbsoluteUsagePeriod,
  relative [3] RelativeUsagePeriod
}

AbsoluteUsagePeriod ::= SEQUENCE {
  notBefore       [0]     GeneralizedTime OPTIONAL,
  notAfter        [1]     GeneralizedTime OPTIONAL
}

RelativeUsagePeriod ::= SEQUENCE {
  notBefore       [0]     INTEGER OPTIONAL,
  notAfter        [1]     INTEGER OPTIONAL
}
```

Listing 11: Definition of a timing trigger.

The `ReplyCapability` as shown in listing ?? allows a *VortexNode* to effectively limit the memory usage.

```
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,
```

Listing 12: Definition of a capability reply block.

20.1.1 Storage Management of Ephemeral Identities, Operations, and Payload Blocks

The ephemeral identity (eID) is the overarching unit of user 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 listing??). Unless used for reply blocks

```
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 OPTIONAL,
  -- 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.

(MURBs), eID have a very limited lifespan of a couple hours. This minimizes any storage footprint associated with an eID.

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 listing ??). 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`.

20.1.2 Life Cycle of Requests

Requests have a separate life cycle. As a request may exist prior to a corresponding workspace, which is typically assigned to a proof of work, such requests may be subject to 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 as outlined in ??. 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.

20.1.3 Minimizing the Memory Footprint of Message Processing

To limit the memory footprint of message processing, we reduced the information relevant to be kept in memory by structuring the message accordingly. A node may first extract the first header block, which is equivalent to the block size of the cipher used to encrypt the header block. If the message is invalid due to a non-existent message, we may stop there. We then start decoding the header prefix block, and if successful, the header, which is typically less than 1KB in size unless it contains a routing block. Each routing block is $1\frac{KB}{hop}$ in size (assuming a 2048 bit asymmetric key). Only the first couple of bytes have to be read, and the vast majority may then be streamed as it is mainly a binary, encrypted blob containing subsequent hops. All subsequent blocks (routing and payload blocks) are

not required to be kept in memory simultaneously. Instead, we may stream them into a data structure on persistent storage. Operations on the payload block are suitable for streaming processing either. For *encryption* and *split/merge* operations this is obvious. the transformation and retransformation of the *redundancy* operations may be achieved with a lookup table. However. it requires 256 KB on disk for a $GF(2^{16})$ transformation. The matrix operations are comparably small again as they may be carried out on an element-per-element basis with simple, calculable lookups.

We may therefore conclude that while a workspace may require considerable storage for storing all payload and routing blocks, the processing of a message can be achieved in a very memory-efficient manner if required. We may execute all calculations on payload blocks in a streamed manner, and all blocks required for routing are either very small or may be streamed again.

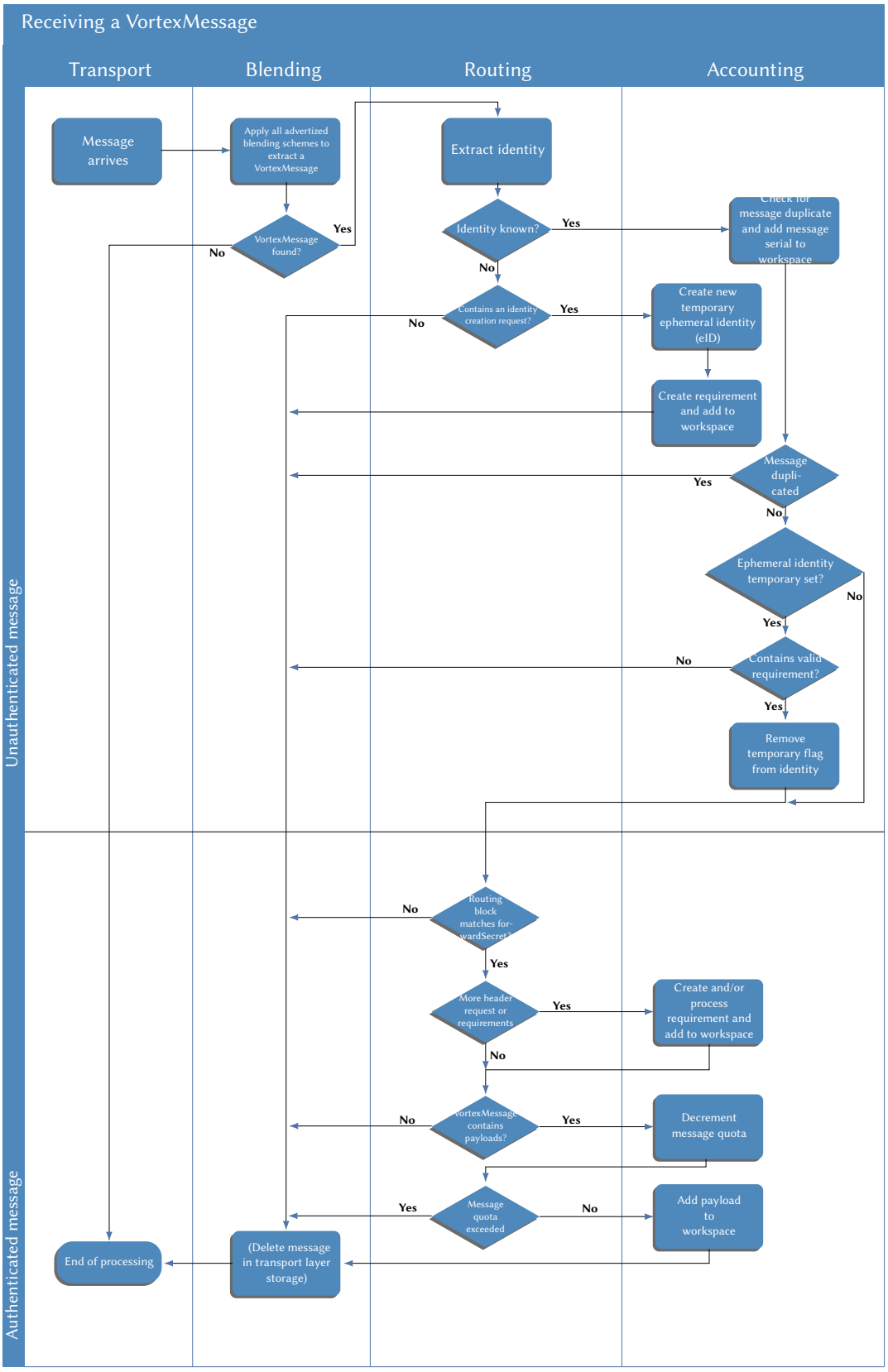

Figure 17.1: Flow diagram showing processing of outgoing messages.

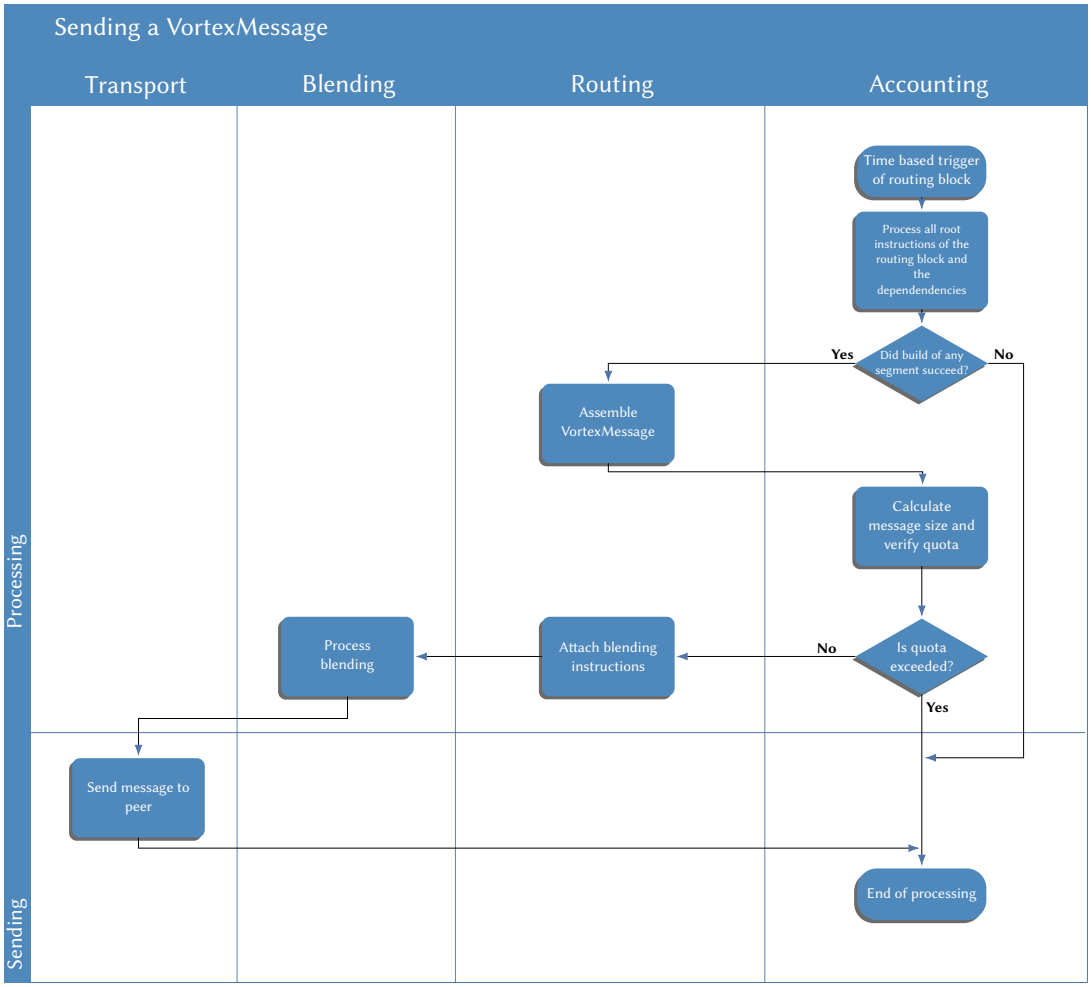

Figure 17.2: Flow diagram showing processing of outgoing messages.

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

In this part we cover operational aspects of our system. Chapter ?? covers some general operational concerns such as *VortexNode* types, or the handling of lifetimes. Chapter ?? covers routing concerns and introduces a simplified algorithm for building routing blocks. Chapter ?? addresses the problem of obtaining keys of routing nodes and bootstrapping a network. Finally, ?? focuses on problems encountered when working with real-world infrastructures.

21 General Operational Concerns

21.1 Hardware

We require no specialized hardware for running *VortexNodes*. Instead, we designed *MessageVortex* in such a way that ordinary mobile phones may act as *VortexNodes*. 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 proof-of-work algorithms' memory requirements. The hardware currently requires a network interface and a fully functional JSE VM to run the reference implementation.

21.2 Addressing *VortexNodes*

From the beginning, we were searching for an addressing scheme suitable for transparent addressing.

A *MessageVortex* address is built as follows:

```
1       localPart          = <local part of address>
2       domain             = <domain part of address>
3       email              = localPart "@" domain
4       keySpec            = <BASE64 encoded AsymmetricKey [DER encoded]>
5       smtpAlternateSpec  = localPart ".." [ keySpec ] ".." domain "@localhost"
6       smtpUrl            = "vortexsmtp://" smtpAlternateSpec
```

To allow storage of *MessageVortex* addresses in standard messaging programs such as Outlook or Thunderbird, we introduced $smtpAlternateSpec$.

The suffix "@localhost" ensures 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. However, the address is not a valid SMTP address due to its double-dotted notation. We selected this representation to differentiate *MessageVortex* addresses from valid email addresses.

The main disadvantage of *MessageVortex* addresses is that they are no longer readable by a human. The main reason for this is the required public key. We may abstract this further by allowing cleartext requests on the primary email address for the public key. The *MessageVortex* account must answer such requests with the valid *MessageVortex* address.

The $smtpUrl$ represents 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 with 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 keyless address may furthermore be used as a synonym for the receivers' real address as any potential receiver may send an unsolicited `HeaderRequestReplaceIdentity`.

21.3 Client

We did not create a *MessageVortex* client for sending messages. Instead, we used a standard Thunderbird email client pointing to a local SMTP and IMAP server provided by a *MessageVortex* proxy. On the SMTP side, *MessageVortex* encapsulates where possible mails into a *VortexMessage* and builds an automated route to the recipient. The SMTP part of *VortexMessage* may be used to automatically encapsulate all messages with a known *MessageVortex* identity into a *VortexMessage*. On the IMAP side, it merges a local *VortexMessage* store with the standard email repository building a combined view.

Using *MessageVortex* this way offers us the advantages of a known client in addition to the anonymity *MessageVortex* offers.

Using a proxy has certain disadvantages. At the moment, the *MessageVortex* client only has a local store. Such a local store makes it impossible to handle multiple simultaneously connected clients to use *MessageVortex*. However, this limitation is just a lack of the current implementation and not of the protocol itself. We may safely use IMAP storage for centrally storing *VortexMessages*. 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 have the same strength as the transmitted messages in terms of detectability
- A secured key
  Either the host key is secured sufficiently with a KDF and a passphrase (or similar), or the host key remains off-storage.

21.3.1 *MessageVortex* Accounts

By definition, any transport layer address may represent a *MessageVortex* identity. This fact may make people believe that their current email or Jabber address is suitable as a *MessageVortex*address. This statement is technically perfectly true but it should not be done for the following reasons:

- If an address is identified as a *MessageVortex* address, it may be blocked (directly or indirectly) by an adversary. Such blocking would lead to the blocking of regular email traffic as well.

- If a *VortexNode* is malfunctioning, non-*VortexMessages* should remain unaffected. Isolation is far better if we keep non-*VortexMessages* in a separate account.
- If a user no longer wants to maintain his *MessageVortex* address, he may give up his *MessageVortex*transport accounts. If he had been using his regular messaging account for *MessageVortex*, he would receive mixed messages that are difficult to filter even with a known host key.

21.3.2 *VortexNode* Types

Depending on the type of adversary within a jurisdiction, a *VortexNode* may require different properties. In ??, we defined observing and censoring adversaries. In environments with an observing adversary, the presence of a *VortexNode* 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.

21.3.2.1 Public *VortexNode*

Public nodes are nodes, which advertise themselves as standard mixes. Just like all nodes, they may be an endpoint or a mix. Typically, they accept all requests precisely as outlined in ??. As an immediate result of the publicly available information about such a node, the owner may be the target of our censoring adversary. An adversary may oppose pressure 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.

21.3.2.2 Stealth *VortexNode*

This node does not answer any cleartext requests. As an immediate result, the node is only usable by other nodes knowing the node's public key. The node is therefore only reachable on a known secrets' basis. A sender may use this node type in environments with a censoring adversary. People may form closed routing groups that route and anonymize themselves. We have to state 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 to shift the problem of detection from the routing to the less secure transport layer.

21.3.2.3 Hidden *VortexNode*

A hidden node is a special form of a stealth node. It has a predefined set of identities. Only these already known identities are processed. This behavior has certain drawbacks. A sender may not change an existing identity, and he may not create new, unlinked eIDs. 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 with

knowledge of 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.

22 Routing

Routing (as described in **??**) 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 changes relatively 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 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 this implementation's pros and cons regarding complexity and anonymity.

22.1 Strategies for Composing Routing Blocks

We have to follow certain rules when building routing blocks. The rules are:

- Valid chain of operations
  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., $D^{K_b}\left(E^{K_a}(\mathbf{X})\right)$).

- No pattern is repeated within the protocol. This constraint applies to:

  - Timing patterns in messages.
    Assuming 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 expires in a workspace within a certain amount of time) and publish these as general rules, in that case, we allow an attacker to identify such timing patterns of the net and draw precise lines which observed transport messages might 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 more difficult to analyze the system by timing patterns.

  - Operation patterns.
    By defining operations used in a fixed pattern (e.g., first, distribute a message over five independent message paths sized n), we would provide an adversary with clues to where in this pattern he is located and how close he is in regards to the beginning or 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.
    Always sending 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 an "intermediate result") breaks 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.

- Sufficient anonymity set
  We assumed not to trust others' traffic. This means that an RBB has to pick a sufficiently large set for its anonymity needs by itself. Overlapping traffic will add to the anonymity, but an RBB should not rely on that assumption.

We may use several strategies depending on our anonymity needs.

Strategies may include:

- Focusing on the 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 fewer 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 of 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 be thoroughly 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 the analysis for an adversary as there are fewer messages involved in a possible routing (assuming that an adversary has the means to magically identify all *VortexMessages*). Inversely, if the speed of a message may be generally slow, an adversary has to take far more messages into account.

- Focusing on the size of the anonymity set.
  The more involved the nodes and transport protocols in a routing block are, the more complex observation of the protocol is. By increasing the anonymity set, the likelihood of overlapping routing graphs increases significantly. Furthermore, the regular message traffic of the transport protocol may further increase the complexity for an outside observer.

- Focusing on anonymity of the eIDs.
  By using only short-term eIDs whereever 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 to a message before sending is safe, unless $n - m$ nodes 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 diagnosability.
  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 the first delivered block belongs to a message 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.

22.2 Strategies for Minimizing Impact and Maximizing Effect when Routing Foreign Messages

Keeping a single node alive can be crucial. If we assume that the a message is received and sent through the same transport account, it is relatively easy for an adversary to observe this. By sending it 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. This way, an adversary needs to carry out man-in-the-middle (MitM) attacks in the respective access protocols or gain 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 an 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 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 **??**. 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 carried out, it is 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.

22.2.1 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 found out that such messages imply privacy issues. While the error messages were discarded 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 time. Such sending may be carried out without 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 unknown but the receiver of the message is.
- 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. He 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. The protocol is limited to a maximum of 127 usages. This number should be sufficiently sized for automated messages. A minute pattern would disappear after 2 hours at the latest and an hourly pattern after five days.

For a MURB to work, the RBB has to ensure that all quotas required to the route are sufficiently sized. Such sizing is difficult to foresee in some cases. An RBB may query these identities from time to time to ensure that they do not deplete. Wherever possible, MURBs should be dropped in favor of multiple SURBs to avoid the dangers of MURBs.

22.3 Routing Algorithms Suitable for Achieving Anonymity

In **??**, we elaborate 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.

22.3.1 The Routing Block

In general, an RBB builds a routing block in three stages:

1. Create a random but "valid" directed multigraph (routing graph) where the nodes represent *VortexNodes*, and the edges represent actual messages sent between the *VortexNodes* and are assigned a label depicting the sequence in time. 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 **??**) offers a better overview over the inner workings of a routing graph. For a graph to be valid there must be at least one valid path from node 0 to any other node, including node 1 which is our main target. Furthermore, outgoing edges may only arise after a an incoming edge is present.

2. We then rewrite that graph and order it while assigning timing information to each edge, leaving sufficient time in between to process the incoming message on the transport layer.

3. As the next step, we assign operations to all involved workspaces.

Based on 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 the same node multiple times, and a routing graph may contain multiple paths between two given nodes. Figure **??** shows all paths between nodes 0 and 1 of the graph outlined in **??**. 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 to transport redundant message parts.

A possible routing mechanism creating such a graph and applying routing information is described in detail in **??**.

22.3.2 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 **??**.

Algorithm 1 Objects for building a routing block.

```
1: ▷ A routing graph
2: object RoutingGraph
```

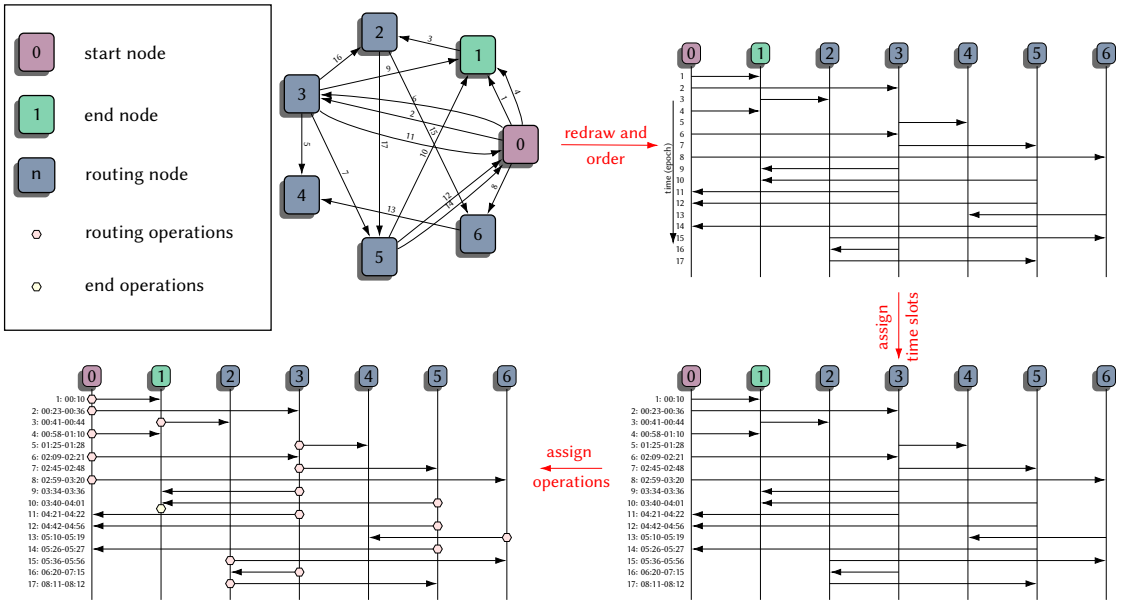

Figure 22.1: Transformation of a graph into a sequence of messages.

```
3:      ▹ Contains the routing VortexNodes (node[0]→sender; node[1]⇒receiver)
4:      nodes : Sequence<Node>
5:      ▹ Contains messages between the nodes
6:      edges : Sequence<Message>
7:  end object

8:  object Message
9:      sourceNode : Node
10:     sourceId : int
11:     earliestTime : datetime
12:     latestTime : datetime
13:     targetNode : Node
14:     targetId
15:     operations : List<Operations>

16:     procedure setTiming(min,max)
17:         earliestTime ← min
18:         latestTime ← max
19:     end procedure
20: end object

21: ▹ The projected workspace of any eID under our control
22: object WorkSpace
23:     payloads : Map<Id,Payload>
24:     routingBlocks : List<RoutingBlock>
25:     operations : List<Operation>

26:     ▹ Returns an unused id with at least <numberOfSubsequentIds> unused IDs following
27:     abstract function getUnusedId(numberOfSubsequentIds) {…};
28:     ▹ Returns a random output id of an operation unused so far and marks it as used
29:     abstract function getRandomPayloadId() {…};
30: end object

31: ▹ An object reflecting our knowledge about MessageVortex
32: object Universe
33:     knownNodes : Map<Node,WorkSpace>
34:     keysize : Integer ← 256

35:     ▹ Returns all the nodes of knownNodes
36:     abstract function getAllNodes() {…};
37:     ▹ Returns a random node of list
38:     abstract function getRandomNode(list) {…};
39:     ▹ Returns the representation of the workspace of the named node
40:     abstract function getWorkspace(node) {…};
41:     ▹ Adds a message to a workspace with all its content (payloads, operations)
42:     abstract function addMessageToWorkspaces(message) {…};
43:     ▹ returns an integer r within 0<=r<maxValue
```

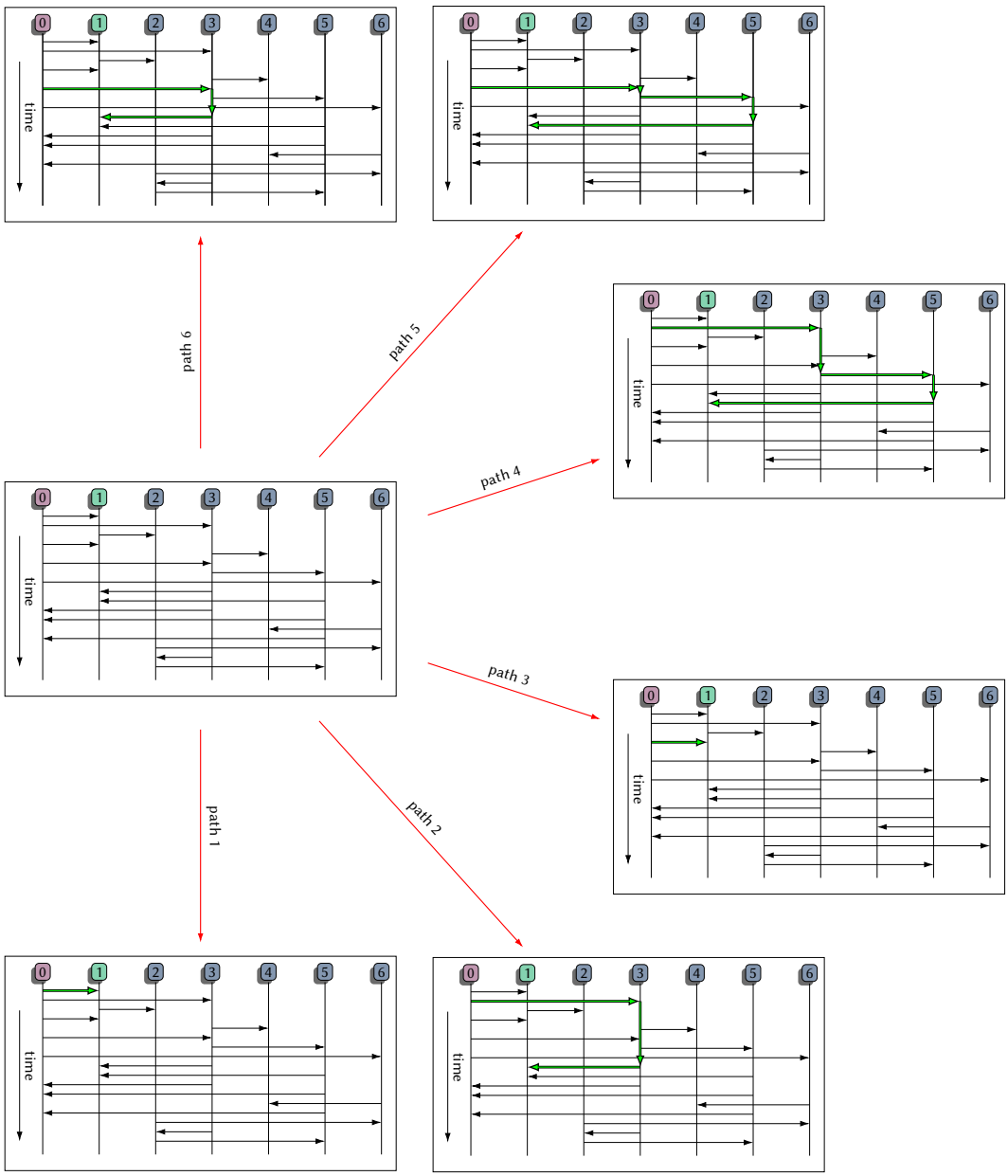

Figure 22.2: A graph containing six paths between node 0 and node 1.

```
44:     abstract function nextRandomInt(maxValue) {...};
45:     ▸ returns an double r within 0<=r<1
46:     abstract function nextRandomDouble( ) {...};
47:     ▸ returns an double r with a Gaussian distribution
48:     abstract function nextRandomGaussian( ) {...};
49: end object
```

To create a routing block, we first need a graph representing the message flow. Algorithm ?? shows a pseudo-code to create such a valid graph. After creating a graph, we need to assign timing and routing information. Algorithm ?? shows a possible algorithm for assigning this timing information, whereas algorithm ?? shows a simple generator for the routing operation. The algorithm omits IDs for simplicity allocation of the workspace as this is a "bookkeeping"-only problem.

To create a graph, we use the function ?? on line ?? as shown in algorithm ??. 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 ?? (Line ??). 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 points 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:     ▷ The maximum number of seconds until the message needs to be delivered
 3:     maxTime ← 3000
 4:     ▷ The minimum number of seconds a message has time to be on one routing node
 5:     minHopTime ← 10
 6:     ▷ The minimum number of seconds a message has time to be on one routing node
 7:     redundantRoutes ← 3

 8:     ret ← new RoutingGraph()
 9:     ret.nodes ← getNodes(startNode, endNode, numNodes, universe)
10:     ret.edges ← getEdges(minEdges, ret.nodes, universe)
11:     ret.edges ← assignTiming(ret.edges, maxTime, minHopTime, universe)
12:     ret.edges ← assignRouting(ret.edges, redundantRoutes, 0, universe)
13:     return ret
14: end function

15: function getNodes(startNode, endNode, numberOfNodes, universe)
16:     nodeList ← [startNode, endNode]
17:     while len(nodeList) < numberOfNodes do
18:         randomNode ← universe.getRandomNode()()
19:         if ¬nodeList.contains(randomNode) then
20:             nodeList.append(randomNode)
21:         end if
22:     end while
23:     return nodeList
24: end function

25: function getEdges(minEdges, nodes, universe)
26:     edgeList ← []
27:     listOfReachedNodes ← getReachedNodes(edgeList, nodes[0])
28:     while len(edgeList)<minEdges or
              len(listOfReachedNodes) < len(nodes) do
29:         startNode ← universe.getRandomNode(listOfReachedNodes)
30:         endNode ← universe.getRandomNode(nodes − [startNode])
31:         edgeList.append(new Message(startNode, endNode))
32:         listOfReachedNodes ← getReachedNodes(edgeList)
33:     end while
34:     return edgeList
35: end function

36: function getReachedNodes(edgeList,startNode)
37:     reachedNodeList ← [startNode]
38:     for all e ∈ edgeList do
39:         if ¬reachedNodeList.contains(e.targetNode) then
40:             reachedNodeList.append(e.targetNode)
41:         end if
42:     end for
43:     return reachedNodeList
44: end function
```

Function ?? is specified in algorithm ?? on line ??. In this function, we assign the timing information to the graph.

We use a custom random distribution called ??(line ??). 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 ?? shows the distribution of the implementation.

We assign the timing information by looping through our ordered set of edges. First, we calculate the earliest (earliestTime) and the maximum available time starting then (maxShare)

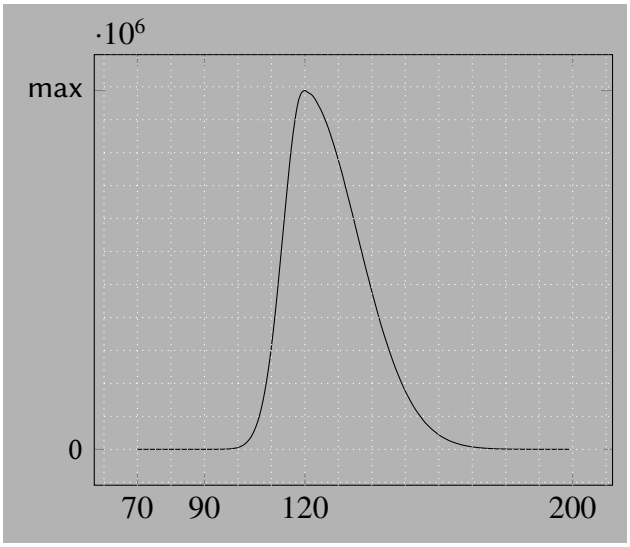

Figure 22.3: Distribution of ??(90, 120, 200) in algorithm ??.

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 ??) and latest (maxTime; line ??).

Algorithm 3 Assign Timing Information to a Graph.

```
 1: function assignTiming(edges, maxTime,minHopTime, universe)
 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, universe)
14:         earliestTime ← maxTime + minHopTime
15:         remainingHops ← remainingHops − 1
16:         maxRemainingTime ← maxTime − earliestTime
17:         e.setTiming(minTime, maxTime)
18:     end for
19:     return textedges
20: end function

21: function getRandomTime(min, peak, max, universe)
22:     value ← min − 1
23:     while value < min or value > max do
24:         value ← universe.nextRandomGaussian( )
25:         d ← universe.nextRandomDouble( )
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 to the graph itself is neither the edges or nodes nor the timing, but the operations applied to the graph. This part is covered by function ?? in algorithm ??. We assign the operations in three steps. We first assign to redundantRoutes a valid message path (lines ??-??). Then we identify “unused (sub-)routes” and assign the same operations to these routes (lines ??-??).

Operations are assigned in a recursive manner. First, we identify the routes we want to assign operations. This recursive part is achieved by the ??(line ??-??). We first identify a payload to be transported and the chain of nodes. We call ??, 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 an operation on the current node a, and on the final node we apply the reverse operation, thus rebuilding the message on the last node simultaneously.

Algorithm 4 Assign Routing Information to a Graph.

```
 1: function assignRouting(edges, redundantRoutes, messageId, universe)
 2:     if redundantRoutes < 1 then
 3:         throw "At least one route is required"
 4:     end if
 5:     routes ← getRoutes(edges)
 6:     if len(routes) < redundantRoutes then
 7:         throw "Graph has not enough redundant routes"
 8:     end if
 9:     ▹ Add operations to true routes
10:     numRoute ← 0
11:     while redundantRoutes > numRoute do
12:         currentRoute ← routes[numRoute]
13:         assignRoute(currentRoute, payloadId, currentRoute[LAST], 0)
14:         numRoute ← numRoute + 1
15:     end while
16:     ▹ Add sensible operations to decoy routes
17:     for all r ∈ getUnusedRoutes(edges) do
18:         assignRoute(r, r.getRandopOperation().getUnusedIds(1), NULL, 32769)
19:     end for
20:     addMessageMapping(edges)
21:     return edges
22: end function

23: function assignSingleRoute(route, payloadIds, lastNode, targetIds)
24:     source ← route.getSourceNode()
25:     if payloadIds.isEmpty() then
26:         PayloadIds ← source.getRandopOperation().getUnusedIds(1)
27:         payloadSet ← assignRoute(route[2-], targetIds.forward(), lastNode, targetIds.reverse())
28:     else
29:         targetIds ← assignOperation(route.getSourceNode(), payloadIds, lastNode, targetIds, universe)
30:         payloadSet ← assignRoute(route[2-], targetIds.forward(), lastNode, targetIds.reverse())
31:     end if
32: end function

33: function assignOperation(node, transportIds, reverseNode, targetIds, universe)
34:     out ← node.outEdges()
35:     in ← node.inEdges()
36:     if out > 1 or extRandomInt(3) = 1 then
37:         ▹ assign addRedundancy
38:         numBlocks ← max(out+1, universe.nextRandomInt(out+4))
39:         seed ← universe.nextRandomInt(2^256)
40:         op ← node.addRedundancy(transportIds, numBlocks - 1, numBlocks, seed)
41:         if reverseNode! = NULL then
42:             reverseOp ← reverseNode.removeRedundancy(targetIds, op)
43:             newId ← op.getUnusedIds(1)
44:             newId.addReverseIds(reverseOp)
45:         end if
46:     else
47:         ▹ assign encrypt
48:         keySize ← (universe.nextRandomInt(3) + 2) * 64
49:         key ← universe.nextRandomInt(2^keySize)
50:         op ← node.encrypt(transportIds, "AES", keySize, key)
51:         if reverseNode! = NULL then
52:             reverseOp ← reverseNode.decrypt(targetIds, op)
53:             newId ← op.getUnusedIds(1)
54:             newId.addReverseIds(reverseOp)
```

```
55:         end if
56:       end if
57:       return newIds
58:   end function
```

The algorithm outlined in this section has several of disadvantages due to its brevity. As it proves difficult to split routes in such a compact recursive manner, it was omitted. For the same reason, we always used *addRedundancy* operations, which rebuild 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 uses traffic splitting with either the *split* or *addRedundancy* operation, this weakness disappears.

22.4 Routing Diagnosis and Reputation Building

When all nodes are working as expected, no diagnostic is required. As we rely 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 analyze 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 is aware of the message content and all involved routing operations, it 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-buiding over time can be achieved by assigning to all nodes additively a small reputation value if involved in a working route and subtract a value when 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 in 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 **??**.

$$R_a = \sum_i \frac{w_i}{len\,(I_i)} \tag{22.1}$$

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 achieve 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 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 using explicit diagnosis to identify active adversaries or calculate a reputation but only to identify nodes that are offline.

22.5 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 serves two purposes. 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 each may contain 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 improves security. Both operations allow such behavior, but in a very different way. The operations *splitPayload* and *mergePayload* allow creating payload blocks with any size. However, when transmitting both sizes of a split, they add up to a full block size of the previously completed encryption operation. Thus, 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.

23 Protocol Bootstrapping

Protocol bootstrapping is especially difficult 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*.

23.1 Key Distribution for Endpoints

For endpoints, we may have at least a partial solution. Sending a *VortexMessage* as an unencrypted message to the users' true email, 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.

23.2 Key Acquisition for Routing Nodes

An adversary may make key acquisitions of routing nodes in an observing adversary environment 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 their traffic and the observable endpoints. Simultaneously, they are not suitable as nodes for communicating with environments connected to a censoring adversary. Therefore, such nodes are typically not considered to increase the anonymity set. This is because 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.

Tus, 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 later decryption.

24 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.

24.1 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 selected transport protocol. While a *VortexMessage* may be small, an size limit is possibly imposed by the transport layer. Most SMTP providers define a limit of $10\frac{MB}{message}$. Considering 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 larger 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 larger than the $900KB$

limit on SMTP due to this limitation, the assembly in a workspace allows transporting larger 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. 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), the level of waste is dramatic.

24.2 Redundancy of the *VortexNode*

At the beginning of our work, we attempted to make *VortexNodes* redundant by sharing configuration and state data over the transport media. While the idea was 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*.Thus, we dropped this idea.

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 ??, we described two different kinds of adversaries. These adversaries require different properties to be fulfilled.

An observing adversary is the less restricting one. While this adversary observes all traffic, he does not disrupt communication. Instead, he uses 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 specific 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 related technology. 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. He may furthermore track down individuals within reach and prosecute them. All other illegal system participants 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.

25 Identification of Possible Attack Schemes and Mitigation

In this chapter, we take the attacks identified in ?? and analyze our protocol on whether it is susceptible to such attacks or not.

25.1 Static Attacks

Static attacks typically address weaknesses within a protocol design. The following attacks are typically used to attack protocols similar to our proposal.

A *VortexMessage* itself is crafted in such a way that for a routing node, only minimal effort is sufficient to obtain a short-lived pseudonym (eID) of the sending party 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. In sectionsec:analysisBlendingAndTransport, we emphasize approaches allowing identification and censorship of *VortexMessages* and *VortexNodes*.

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 entirely different.

We refer to a bugging attack as an attack, which discloses the recipient by forcing him to commit a disclosing action. Such an action may be the lookup of an unusual DNS record, 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 an adversary to follow an attribute of a message through a network and, thus uncover members of a network, subsequent messages, or even a final recipient.

Static information leaking of the protocol is another possibility of how an adversary may learn IoIs on a network. Routing nodes are a vital part of any anonymity network. The most

comfortable assumption is to trust nodes. In our case, we explicitly distrust routing nodes. This means that we must identify and judge upon the footprint of available information to such routing nodes, which is done in **??**. Especially in an environment of a censoring adversary, the undetectability of a *VortexNode* is crucial, as any detectability may lead to a shutdown or even repression. We elaborate in **??** how to identify involved messages and nodes.

25.2 Dynamic Attacks

Dynamic attacks usually involve 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.

An active adversary may attack the transport layer. Most of the transport layers are not able to react to 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 cannot create additional protection on the transport layer as such modification would require a modification of the transport layer. We analyze in **??** the impact on the *MessageVortex* system.

We have identified the following attacks relevant to our system:

- DoS attacks against the transport system
- DoS by traffic replay
- DoS by traffic generation
- Attacking a single ephemeral identity of a *VortexNode*
- DoS 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

An active adversary may not follow the protocol and modify any parts of the message. The following paragraphs reflect different types 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. Such a behavior is not sensible as such a node may directly send all knowledge to such a collaborating node. If a target node does not support 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 increasing the signal to the 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 *VortexNode* identifies the block as a repeated block and rejects further processing.

An adversary may replay blocks with varying content. Such replays will not result in a DoS attack as the quota is not decreased on replayed messages (see **??**).

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 thus 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.

26 Static Analysis

26.1 Analysis of the Blending and Transport Layer

The blending layer is one of the key factors for confidentiality in an environment affected by a censoring adversary. We refer to the confidentiality of the presence of a *VortexNode* as detectability. Detectability of messages and systems, in consequence, leads to the ability of censorship by an adversary. We assume that general censorship on the transport layer (e.g., by blocking all SMTP traffic) is not an option.

In an observing adversary environment, confidentiality regarding the presence of messages is not required, as we defined in those environments legal to use *MessageVortex*. In such environments, plain embedding may be used at any time.

26.1.1 Identifying a *VortexMessage* 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 [**oakland2013-parrot**])
- Non-human behavior on the transport layer (e.g., message traffic 24x7)

If an endpoint is successfully identified, all peering endpoints of the same protocol may be identified as well by following the message flow. However, this does not enable an adversary to inject messages as the host key is not leaked.

Assuming a global observer and unencrypted traffic, the observer might discover the originating routing layer and thus identify it as *VortexNode* by following traces of the transport layer. However, in most protocols 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 [**oakland2013-parrot**] is not an issue as it only follows human communication. Thus, our system does not have to pass a Turing 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.

26.1.2 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 code from open source products, it is not perfect as anyone may know what types of messages 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 generated imagery is more difficult.

From the technical point of view, the specification for the blending layer is complete. By specifying only one steganography algorithm, we cannot switch algorithms which makes the blending layer potentially weaker as there is no seconding algorithm such as PQt providing crypto-agility. While F5 has been available for many years, no paper has been published proving the algorithm's detectability. F5 was analyzed and showed remarkable resistance to conventional attacks. Detectability depends on the density of embedded data. A payload of 5–10 percent is currently not deemed detectable in a real-world environment [**fridrich2007statistically**]. Many other algorithms such as nsF5, PQt/PQe, HUGO [**pevny2010using**], S-UNIWARD [**holub2014universal**], MiPOD [**sedighi2015content**], or HILL [**li2014new**] have been evaluated, but algorithms offering a solid implementation are rare nowadays. An implementation in Java was not available for any of the mentioned algorithms. Considering that it is far more difficult to provide a solid implementation than some emulation code for academic purposes, the lack of this is understandable yet makes it very difficult to either incorporate algorithms or test their robustness under realistic conditions.

Hiding 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 require properties, which makes them assignable to a service instead of a user 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 a size restriction for an embedding image is one of the caveats for larger messages as adaptive image size is easily detectable by an adversary.

26.2 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 undeniable when looking at the fact that almost 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 a very weak obfuscation in an observing adversary's environments.

We wanted to know if there was a simple method to detect the modifications of such a file. While most of the analysis method requires 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. While this argument is weak, we already agreed that plain embedding is not suitable for environments with a censoring adversary.

A property of encrypted ciphertext is the high entropy. Therefore, we used the Shannon entropy calculation in bytes as property and tried to show the entropy shift within the files. This detection method depended very much on the type of file used for embedding. It showed the expected behavior that file types with a similar entropy in the expected area 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 and 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 surprising that we were able to identify these features, but the fact that we could see them in entropy data was remarkable.

We then carried out an analysis identifying the typical entropy and the inner structures. The graphs in **??** show a typical analysis. In that specific 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 **??** 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, non-structured range of octets containing these characteristics. Such a file has a suitable space for embedding. For reference, **??** shows the distribution of typical *MessageVortex* blocks. We found that the entropy must be uniformly matched in the case of plain embedding.

When blending into images, BMP showed a strongly varying entropy within a file. A sampling of ten blocks at random position already resulted 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

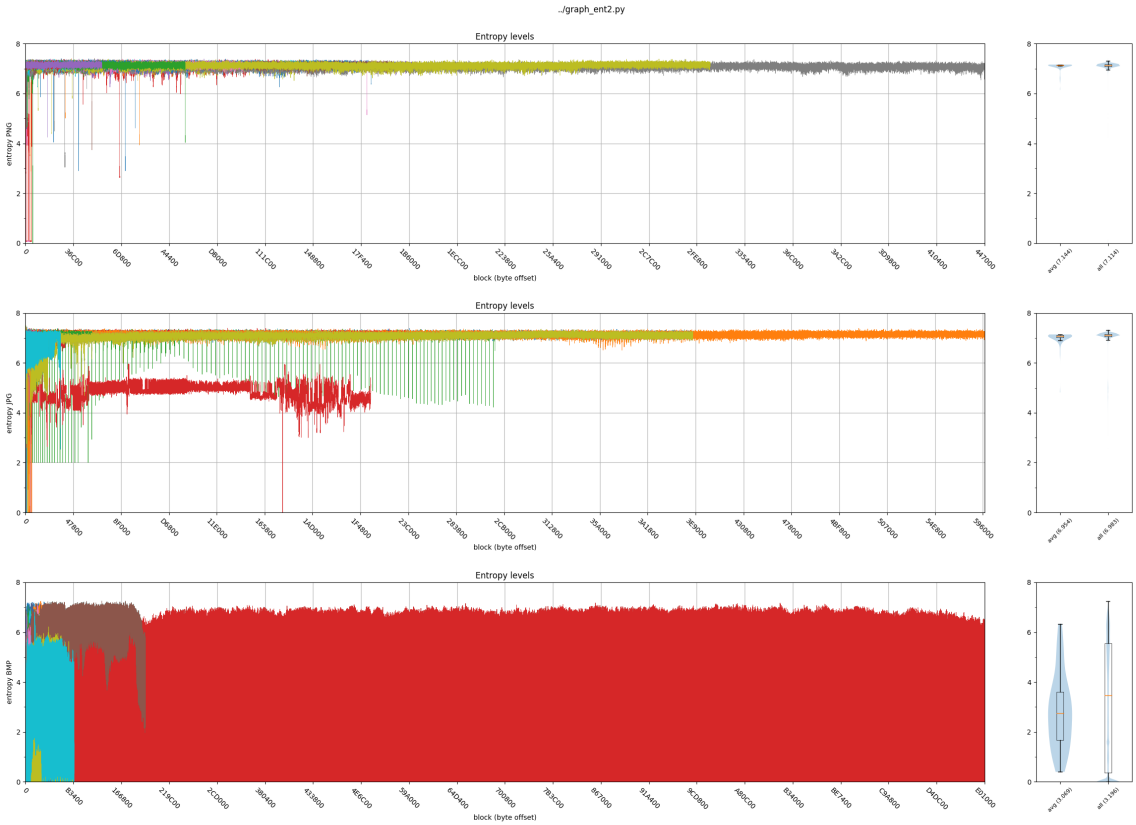

Figure 26.1: Distribution Analysis of Different, Common Graphics Formats.

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	Difficult 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. Difficult 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 26.1: Comparison of potential transport layer.

genuinely undetectable. However, such embedding may potentially lead to censorship based on the blacklisting of encrypted zip files.

OGG and MP3 are suitable. However, we were able to detect the entropy difference when taking extremely dense samples. These formats may however be suitable for not yet standardized forms of steganography. While PDF typically has low entropy and a high IQR, 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 performed 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.

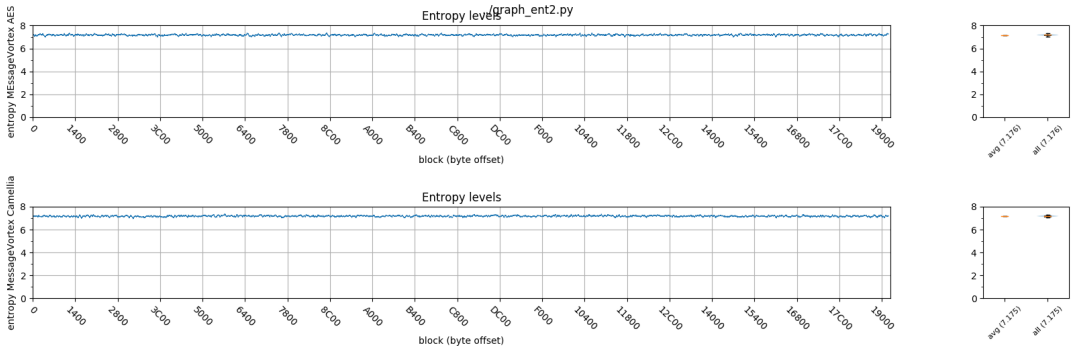

Figure 26.2: Distribution analysis of a MessageVortex block.

26.3 Analysis of Routing Layer

26.3.1 Analysis of Core Operations

The core operations form a toolset for mixing messages. Under the operational restrictions outlined in **??**, we analyze in the following section the operations and determine their capability for leaking information or affecting security.

26.3.1.1 Splitting and Merging

The operations *splitPayload* and *mergePayload* are the trivial operations of our operations set. The operations by themselves leak some information under the assumption that they were previously encrypted. A split or merge operation on its own leaks possible counterparts as the size should add up to a blocksize common in symmetric cryptography. As we outlined in **??** and **??** 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 no 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 incoming payload's nature, the output may be either decoy or true message traffic.

26.3.1.2 Encryption and Decryption Operations

All encryption steps 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 chosen by an implementation-specific pattern, the pattern may leak the implementation over time. As the analysis must be completed over a short period (the lifetime of an eID), it is up to an RBB to leak as little information as possible. However, we regard the cryptographically secured content as secure.

26.3.1.3 Add and Remove Redundancy Operations

During analysis, the *addRedundancy* operation showed the undesirable behavior that applying the operation lowered the target blocks' entropy, as shown in **??**.

Thus, we reconsidered 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 contexts have already been studied in [**mceliece1981sharing**], [**bu2017rasss**], and similar publications.

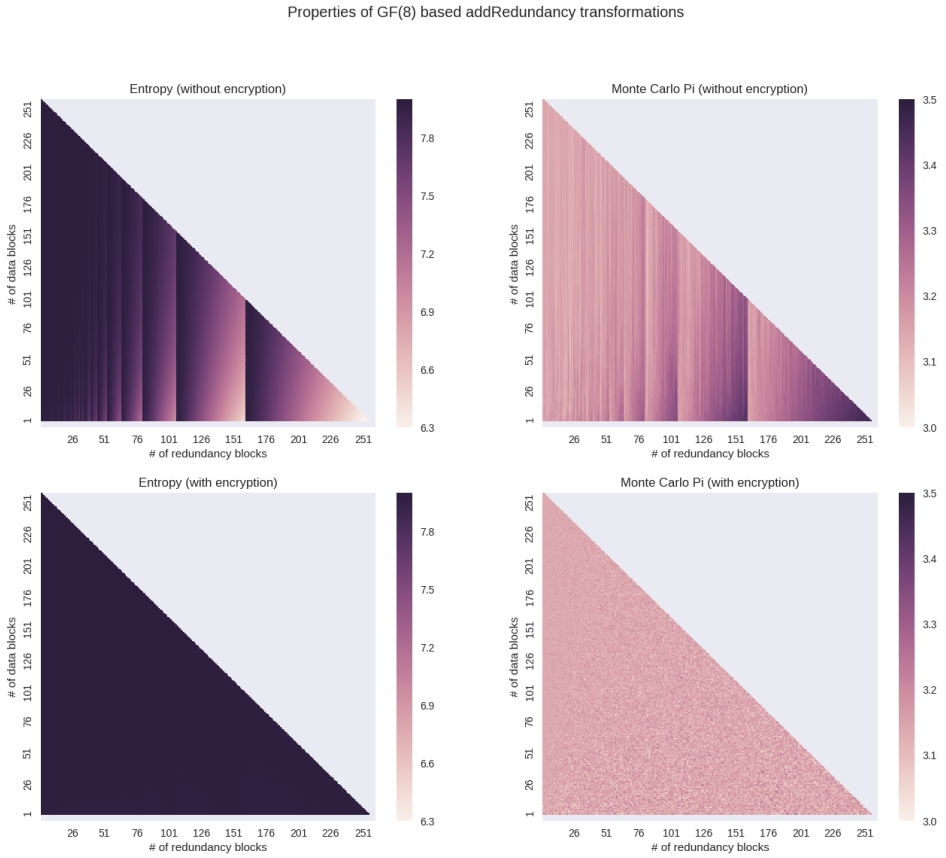

Figure 26.3: Entropy of addRedundancy with and without the encryption step.

26.4 Knowledge of a Node Sending the First Message

A sender of a *VortexMessage*, not equal to the RBB, may have knowledge about the initial routing block size and, therefore, guess the routing path's complexity. He is however,unable to gain any additional information such as time of travel or number of hops until the target is reached. The building instructions only leak minimal information which may also include some ideas about the routing block's complexity.

As with every routing node, the next hops are leaked to the sender. Again this is carried out without leaking the next hop's host key.

26.5 Intermediate Node Routing Layer

An intermediate node knows all the operations applied and the immediate next hop. It learns the routing addresses of the immediately following endpoints but is unable to use these endpoints. This inability is based on the fact that the node has no means to obtain 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 achieved 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. However, this estimate is very rough as it is bound by the block size of the symmetrically applied encryption.

26.6 Security of Protocol Blocks

To analyze the security of the protocol, we first investigate all protocol blocks. Then we look at the possibilities of block recombinations and how to gain data or ervices based on such behavior.

Assuming plain embedding, the presence of a chain of blocks may leak an existing *VortexMessage*. Currently, the protocol expects at the blending offset size and number 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 was chosen. This variable byte length 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 patterns at the beginning of streams due to reused identity blocks, a MURB must provide sufficient peer keys and prefix blocks. However, a *VortexNode* may refuse to process MURBs (only accept maxReplays equal to 0).

All blocks of the InnerMessageBlock are protected by the peer key $E^{K_{peer}}$. The forward secrets in all blocks except the payload blocks ensure 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 node 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 thus 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 cannot assemble messages for a specific host on the basis of only a routed message. 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 crucial if assuming a censoring adversary. Therefore, a

sender in a jurisdiction where 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.

27 Dynamic Attack Analysis

In the dynamic analysis, we reach out to an active adversary. An active adversary modifies traffic in a non-protocol conforming way or misuses available or obtained information to disrupt messages, nodes, or the system as a whole.

27.1 Well-Known Attacks

In the following sections, we emphasize on possible attacks to anonymity preserving protocols. Such 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.

27.1.1 Broken Encryption Algorithms

Encryption algorithms can become broken at any time. Our protocol is especially susceptible to this as it offers no perfect forward secrecy (PFS) on the transport layer. This either due 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 promptly react to such threats. This reaction should not rely on a required update of the infrastructure. Users should solely control the grade of security.

We cannot wholly prevent such attacks from happening. However, we can 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 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 neighboring nodes to support the same algorithms in order to be used.

Another way of minimizing the impact of reduced security of encryption algorithms is to use long host keys. If an algorithm's security is only reduced by a few of bits instead of being broken, then a long key minimizes the impact and ay buy some time to switch to an alternate algorithm.

A broken algorithm is severe if it leads to the decryption of the final messages on the recipient node. In such a case, an adversary would be able to rebuild the content of a workspace and thus effectively enable the adversary to obtain the message's content.

27.1.2 Attacks Targeting Anonymity

Attacks targeting users' anonymity are the main focus of this work. Many pieces of information can be leaked, and the primary goal should rely on the principles established in security.

- Preventing an attack
  Attack prevention can only be achieved for attacks that are already known and thus 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 (excepts forbandwidth 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.

- Minimizing the 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 difficult to gain additional information based on statistical means.

- Redirecting 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).

- Controlling 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 expire within a reasonable amount of time.

- Discoveruing an attack
  The protocol is designed, so 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.

- Recovering from an attack
  An attack always imposes 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 partially functional state either temporarily or permanently.

In the following subsections, we list a couple of attack classes that were used against systems listed in **??** or the respective academic works. We list the countermeasures which were taken to deflect these attacks.

27.1.2.1 Probing Attacks

Identifying a node by probing and checking their reaction is commonly achieved when fingerprinting a service. As a node is participating in a network and relaying messages probing may not be evaded. However, it may be costly for an adversary to carry out systematic probing. This should be taken into account. Both currently specified transport protocols feature an indefinite number of possible accounts. Since not the server but the endpoint address behaves, node probing is more complicated than in other cases where probing of service is sufficient.

One of the problems is cleartext requests. These requests may be used on any transport layer account without previous knowledge of any host key. Thus the recommendation in **??** is generally not to answer the requests. Routing nodes in jurisdictions not fearing legal

repression or prosecution may reply to cleartext requests, but it is usually discouraged as they allow the harvest of *VortexNodes*. A discovered *VortexNode* may leak subsequent nodes if the same account is used for receiving and sending.

One strategy to avoid this would be to put high costs onto cleartext requests so that a cleartext 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 strong adversary may have many identities in stock. Requesting an unusually long key as a plaintext identity does not make sense either, as these as well may be kept in stock. However, we may force a plaintext request to have an identity block with a hash following specific rules. For example, we may put in a requirement that the first four bytes of the hash of a header block correspond to the first four characters of the routing block. At the moment, this was rejected in the standard for practical reasons. First, as the request is unsolicited, a sender is the only one able to decide the hash's algorithm. This would allow a requester to choose upon the complexity of the puzzle. Second, any negotiation of the request's cost would result in the disclosure of the node as *VortexNode*, which might be unsuitable.

27.1.2.2 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 quite often reveal senders or recipients. Sometimes even an intermediate node in an anonymizing system.

The assumption that a hotspot arises at a specific point in our protocol is wrong. At any point in the lifecycle of a message, either payload blocks are left out until expiry, or additional traffic may be generated using an *addRedundancy* operation.

27.1.2.3 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 the forwarding of any block without an appropriate operation. This denies the possibility of tagging 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 being aware of the subsequent private keys of the *VortexNodes*. We will cover this scenario in **??**.

27.1.2.4 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.

27.1.2.5 Sizing Attacks

There are two types of sizing attacks identified as relevant for us. One is the possibility of matching messages with related sizes, and the other 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 collapsing the operations of one or multiple honest *VortexNodes* between two malicious *VortexNodes*. To do so, the second evil node 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 being transferred through the honest nodes.
2. The number of payload blocks may have changed.
3. The size may have been further obfuscated because 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 increases (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.

27.1.2.6 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 types of proof of identity, such as certificates, offer some revocation facility. While this is a perfect desirable property of these infrastructures, they have 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 carry out 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 with access to a trusted source of proof of identities. A third possibility is relaying the request to another host either by using an anonymity structure such as Tor or 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 proof could be a timestamped and signed partial CRL. It would allow a node to verify a certificate's validity without being forced to disclose itself by carrying a verification. On the downside, including a proof of non-revocation involves the requirement to accept a certain amount of caching time to be accepted. This caching cycle 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 considering effects on anonymity. Typically, the use of such protocol is only a minor issue since an ISP usually makes the resolution of a lookup. Normally an ISP would not keep a query log as such logs tend to become big, and their information content is comparatively low. In the case of a censoring adversary, an ISP may be forced to keep such a log or to provide access to the adversary.

  The bugging in general attack works as follows: We include a unique DNS name to be resolved by a recipient. This can be carried out 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 carry out 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 type of attack, plaintext 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.

27.1.2.7 Analysis by Building Interaction Graphs

Building interaction graphs is very difficult to accomplish with our system. Although we cannot quantize the effect, we still may elaborate on the difficulties. We first look at our system from an outside view and then do the same for a powerful adversary inside the system.

When looking from outside the system, interaction graphs are difficult 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) fetch the messages, and then follow all possible connections to other infrastructures known to be supported and asume to be outbound messages. By assuming that an outside observer is able to identify all *VortexMessages* and surpass all difficulties involved in following the different protocols. Then such an observer is capable of generating a graph having as nodes all *VortexNodes* and as edges all *VortexMessages*. An adversary would then require the means to identify the sender and recipient. We first claim that there is no possibility to identify such senders and recipients as there is no guaranteed minimum or maximum time for a message. As an immediate result, any *VortexNode* sending a message may be a sender of a message or only a router. Inversely, any *VortexNode* receiving a message may be either a recipient or a router. Due to the operations, sizes may increase or decrease on message paths. Therefore, an outside adversary is unable to match two adjacent messages to the same identity. Any previous message, including all subsequent messages, may have triggered the sending of the message. So from an outside perspective, we have no possibility to identify by message pattern, message size, or message sequence adjacent messages.

We assume the example routing graph, as shown in **??**.

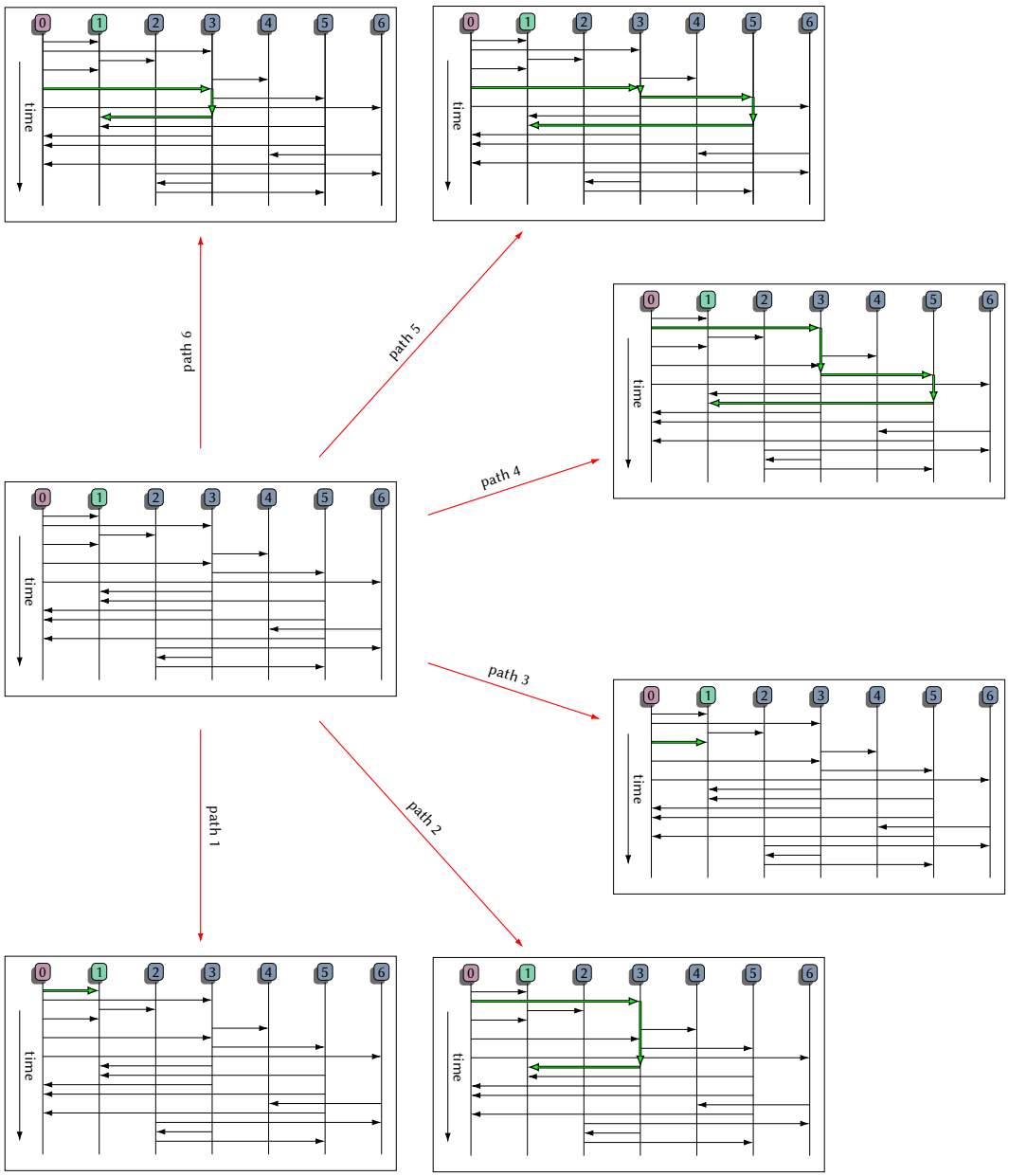

Figure 27.1: A randomly generated graph with highlighted paths to the target.

From an inside perspective, we take additional information into account. First, if an adversary has control over a routing node, he is aware of all operations carried out by this node. He knows the immediate sender and the immediate recipient of any immediately subsequent messages. If the adversary has control over two or more adjacent *VortexNodes*, he is able to collapse the operations into one big workspace with the combined operations, whereas the message transfer may be reflected in a simple mapping operation. He is also able to identify subsequent messages using the same eIDs as messages of the same RBB. He is, however, unable to tell whether or not two subsequent incoming *VortexMessages* for the same eID belong to the same or to two different messages. If the same RBB maintains multiple eIDs simultaneously on the same routing node, the node is unable to match from those eIDs to the same RBB as they share no common properties. In a worst-case scenario, this means that all routing nodes chosen by the RBB, with the exception of the sender node and the final recipient node, are under the control of an adversary. This would effectively collapse an interaction graph to a reduced graph, as shown in **??**. An adversary learns that there are two adjacent nodes to his network (node 0 and 1). Such an adversary is however unable to tell

whether node 0 was an initial sender, as any incoming message into node 0, regardless of its source or timing, may have been the cause for the original message sending. Arguing the same way, we may say that either node 0 or 1 cannot be the recipient for sure as any other outgoing message, regardless of timing or size, may have been triggered by the incoming target, and the final recipient may be any subsequent node.

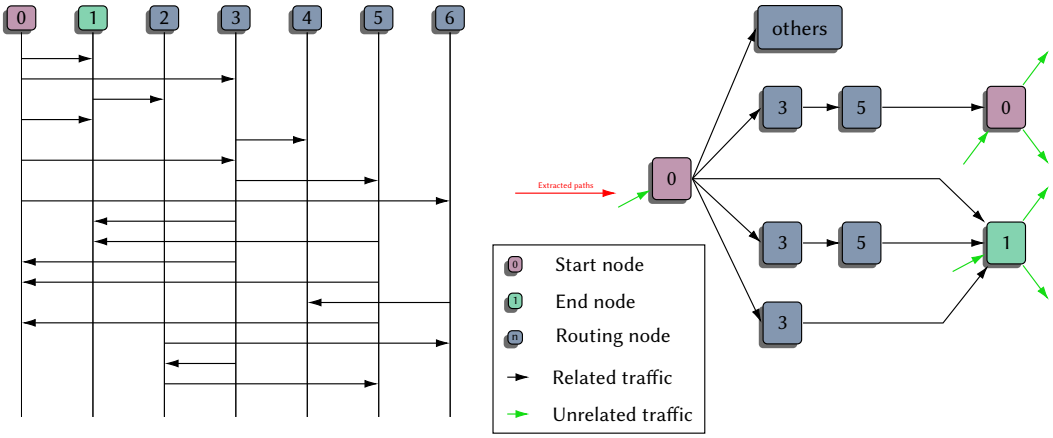

Figure 27.2: The graph of ?? assuming all nodes except node 0 and 1 are evil.

We can therefore tell that we need to trust the sending and receiving node to be safe. Furthermore, we need additional traffic generated by any non-collaborating *VortexNode*. The fact that there are neither timing nor sizing constraints makes it impossible to match any two messages to the same original sender. Therefore, our message is not traceable if there is additional traffic going through our trusted nodes, and honesty of the initially sending node as well as the final recipient node is assumed.

27.1.3 Denial of Service Attacks

27.1.3.1 Censorship

Whereas traditional censorship is widely regarded as selective information filtering and alteration, very repressive censorship can even include the denial of information flows in general. Any anonymity system not offering the possibility to hide in legitimate information flows is therefore not censorship-resistant.

27.1.3.2 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 receive 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.

27.1.3.3 Credibility Attack

Another type of DoS attack is the credibility attack. While not a technical attack, it is very effective. A system without a sufficiently big user base is offers 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 are:

- Disrupting the functionality of a system.
  This may be achieved by blocking 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 threat.

- Publicly disputing 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 that is believed to be compromised if the primary goal of using the system is to avoid being observed.

- Reducing 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.

- Disputing the credibility of the system founders.
  Another way of reducing the credibility of a system is to undermine its creators. For example, if 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.

- Disputing 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.

27.1.3.4 Denial of Service by Exhausting Quotas or Limits

A malicious node may try to exhaust quotas or limits. As we trust the sender and recipient, all other nodes are unaware of the forward secrets used in the message. The options for an adversary are then as follows:

- Resending a MURB (with different content) as often as possible to exhaust message and transfer quota.
- Creating intentionally huge, incorrect message content to exhaust transfer quota.

27.1.4 Attacking Sending and Receiving Identities of the *MessageVortex* System

An adversary's most valuable goal is breaking an entity's anonymity or monitoring their traffic by the content or the metadata. In the following sections, we analyze the possibility of determining the sender or recipient of a message.

27.1.4.1 Traffic Highlighting

Traffic caused by a routing block may be observed to a certain extent on a statistical basis. A node may generate bad message content of exceptionally large or small nature. Such messages might potentially highlight messages involved in message routing using no split or relative split operations as well as addRedundancy operations.

27.1.5 Recovery of Previously Carried out Operations

An adversary must be unable to recover parameters of a previously carried out operation. We analyzed the protocol operations carefully to ensure not to leak any of the parameters. Some operations leak apparent data, such as an encryption operation with a block cipher typically leaks its block size. However, this was 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 **??**, 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 Shannon entropy in bits per byte were already sufficient. The results became less clear when applying the same operation to random blocks while carrying out the analysis.

We have however found a flaw in the *addRedundancy* operation. When applying this operation to an encrypted block, the resulting block's entropy leaks some of the operation parameters. As a result of this finding, we added a custom padding and an additional encryption step. The repeated analysis showed that the operation no longer leaks these parameters through this channel.

27.2 Side Channel Leaking

We tried to minimize the number of possible side channels. Some of the side channels are irrelevant as trusted nodes control them. Some side channels remain unavoidable unless we restrict messages to an unrealistic minimum.

27.2.1 Software Updates and Related Data Streams

We consider assuming in today's world that updates are not needed for security reasons a foolish thing. However, downloading a software update may uncover a user. While it is feasible to transport software unseen once, transporting software on a regular basis is a tedious job. Therefore, we included a standard way of querying a new software release and receiving the new release over the *MessageVortex* protocol allowing the same degree of privacy as with all other messages sent. While the path itself is cryptographically secured, we recommend that the code should still be signed, and the signature should be verified before upgrading to a new software version.

27.2.2 Bugging in Transported Messages

Bugging in transported messages is possible as we have no clear definition of the content of a message. As the transport is currently XMPP and SMTP, the assumption of sending MIME-encoded messages is obvious. The availability of clients and the simple feasibility of gateways make it an obvious choice. If we use MIME as transport encoding, we may leak certain attributes such as the reading location of a message to a sender by including external images or signing the message with a certificate whose verification authority is tapped. Since we trust the sender and recipient node and assume that the RBB is one of them, this argument does not count towards any of the messages. Any other intermediate routing node has no means of injecting any content into the message or the routing bloks. Therefore, in terms of bugging protocol messages, our protocol is rather secure.

This statement leaves some interesting questions unanswered. First, when creating an eID on an intermediate node, we have to analyze this situation as well as trust the target node (the node on which an eID is allocated), which in such a situation is most likely not trustworthy. While the node is a final node for the request, the node is not regarded as the final destination for a message. In that specific case, the reply block the node receives maps not to ID 0, but 32767 (see **??**). This difference keeps a routing node to misuse the reply block for sending a bugged message.

27.2.3 Exploiting MURBS

Multi-Use Reply Blocks (MURBs) are another source for a side-channel attack. While technically safe, the possibility of creating repeating patterns over a network causes the possibility to recognize the communication pattern. As we have no strict timing for sending our messages, this pattern discovery remains a not easily solvable problem. To make it even more difficult, we restricted the reuse of a MURB by design to 127 times and included the possibility to of using different prefix blocks in each message. Without these prefix blocks,

the pattern would have been easily identifiable as the prefix block would have formed a byte sequence that would have been detectable in all messages using the same MURB.

Replaying a message built with a MURB does not necessarily require identifying *VortexMessages*. It is sufficient to store and replay a suspected message and trying to analyze whether a related communication pattern is visible. The pattern reflects all messages which are triggered by the message. As an adversary is unaware whether he replayed the first or an intermediate message, he cannot tell whether he was able to observe the full graph or just a subset. Furthermore, the graph generated by the replaying (assuming that the replay protection did not catch the message) may be smaller, as other parts of the message traveling through other nodes may not have been replied to, leading to non-sendable messages.

If we assume that an adversary identifies all messages and involved *VortexNodes* of a MURB, we have two things to consider. In an environment of a censoring adversary, the confidentiality of *VortexNodes* is compromised. Additionally, in an observing adversary environment with many MURBs, high replay ratios, and small routing sets, we would be able to build a list of the routing set an unknown *VortexNode* has, leading to pseudonymity for that node. We cannot see how this could be further exploited, but this fact should be mentioned.

To weaken the threats of MURBs, we eradicated all needs for MURBs within the protocol. A MURB has only to be used when a user decides to do so, and we recommend not using them.

27.3 Achieved Anonymity and Shortcomings

27.3.1 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 feasible 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.

27.3.2 Attacking Routing Participants

While very difficult in our case as we do not have "dedicated" anonymization infrastructure, it might be possible to identify the routing network members due to flaws in the blending layer. It is possible to scare off or block members of a routing network. It is far more difficult in a network where the members are mobile. Any user may change his identity, including

the endpoint, without losing its known peers by notifying known communication partners about the change. 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. The transport layer provider may then be forced to de-anonymize the customer related to the account (if possible), or the relating account on the transport layer may be blocked.

To counter a possible threatening de-anonymization, a *VortexNode* owner must maintain anonymity towards the transport layer provider. Presently, this is easily achieved the XMPP protocol. The account is typically not linked to any subsequent user information, such as telephone number or email. Email accounts are more restrictively regulated. Providers of accounts without registration of phone numbers or subsequent email addresses exist (e.g., Yandex) but are rare. In both cases, a user might be identified by its IP address. This is why concealing the IP address while connecting to the transport layer is an advisable practice. Using Tor when accessing the transport layer may suffice. 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.

27.3.3 Attacking Anonymity through Traffic Analysis

As traffic and decoy traffic are chosen by the RBB, frequency patterns cannot be detected, unlike the router that created 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 difficult 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 value. By monitoring the endpoints used by an ephemeral identity, we might calculate a “likelihood of matching” for two ephemeral identities. Luckily this is not feasible 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 attempting 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 de-anonymization 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 were 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 the nature of the received message. If

an RBB avoids repeating patterns of blocks on nodes, it is impossible to link the ephemeral identities of two non-adjacent nodes. For example, repeated patterns may arise 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.

$$
\begin{aligned}
&\text{node f:} \\
&\qquad pb_2 = D(pb_1 \\
&\qquad pb_3 = E^{K_t}(pb_2) \\
&\text{node f+1:} \\
&\qquad \vdots \\
&\text{node f+x:} \\
&\qquad pb_4 = D^{K_t}(pb_3)
\end{aligned}
$$

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 destroy data without knowledge 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 on the success or fail of an operation.

As the communication pattern is defined by the RBB and is not always the same, it is difficult to judge the security. However, we may look at some generic examples and show that we can achieve the goals of Byzantine fault tolerance, privacy and unlinkability, and anonymity. **??** 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 protocol properties. 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 must consider that two adjacent nodes collaborating may build one combined workspace to execute all operations. Therefore, they are able to link all operations of these two adjacent nodes and follow all incoming and outgoing paths. Therefore, we may assume that two adjacent nodes or an uninterrupted series of collaborating nodes may be substituted by one node.

A routing node $n_{1,}$ may not know if a *VortexMessage* received from s is the result of processing another message or the message was injected on node s. Furthermore, if s was acting as a routing node, it successfully unlinked the 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

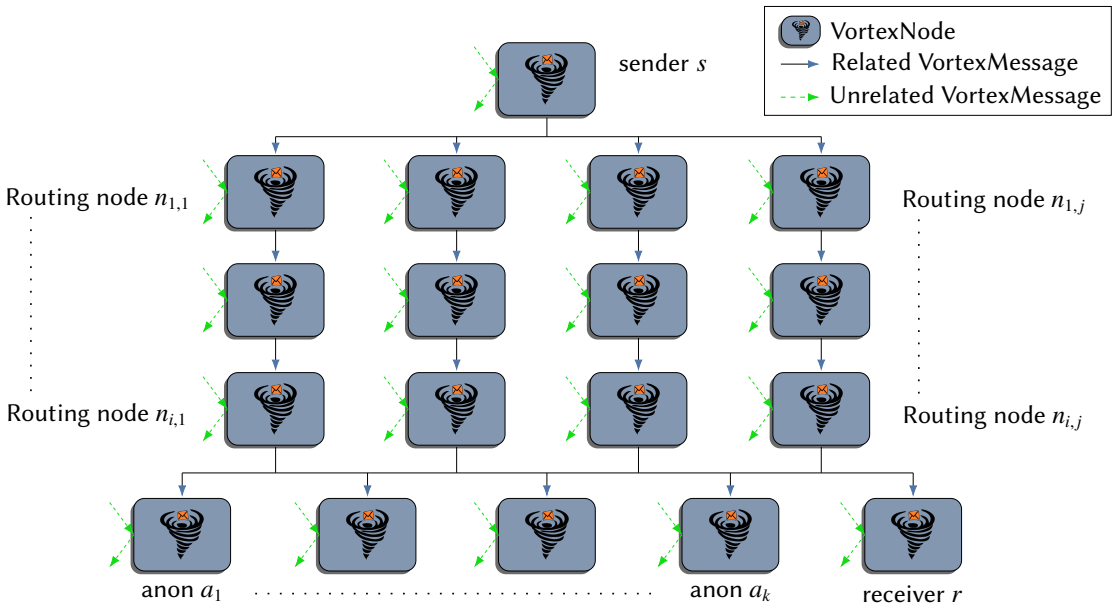

Figure 27.3: A possible path of a VortexMessage.

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. Thus, we have shown that anonymity, unlinkability, and fault tolerance against a misbehaving node may be achieved depending on the chosen routing block. 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.

27.3.4 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 of the network depends on other factors as well, such as delays through anti-UBE or anti-malware measures or delays through local delivery between multiple nodes.

27.3.5 Attacking Anonymity through Throughput Analysis

Increasing the throughput to highlight a message channel is impossible 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.

27.3.6 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.

27.3.7 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 prefix block's size may leak the key size. However, this is a minor issue as the header is structureless and may not be identified.

To an adversary knowing the decryption key (evil routing node), the header block's content 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. However, this activity only matches the minimal activity of an endpoint identity as an endpoint may have multiple ephemeral identities on one node.

27.3.8 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.

27.3.9 Attacking Anonymity through Bugging

Bugging is one of the most pressing problems. The protocol has been carefully crafted not to allow any bugging. However, the use of MIME messages in the final message enables the bugging of the message itself. A bugged message content may breach receiver anonymity to the sender of the message.

27.3.10 Attacking Anonymity through Replay Analysis

Due to the replay protection, no traffic may be generated or multiplied except for the attacking node's traffic. As this information is already known to the node, there is no value in doing so.

27.3.11 Diagnosability of Traffic

27.3.11.1 Hijacking of Header and Routing Blocks

An attacker might try to recombine a third party's header block 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 unknown, the attacker cannot 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 one-time 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.

27.3.11.2 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 some loops' sub status 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 achieve this by taking specific excerpts of the received message at the recipients' node and route these blocks back from the recipient to the sender.

27.3.11.3 Partial Explicit Routing Diagnosis

If a message fails to deliver according to an implicit routing diagnosis, additional messages may be sent to collect 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 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.

28 Analysis of the Effectiveness of Attack Schemes

In the previous sections, we have identified some potential technical weaknesses of the protocol. These weaknesses condensed to the following recommendations:

- Avoid using MURBs (SURBs are not a problem)

- Avoid fixed/repeating patterns or sub-patterns when routing
- Keep workspace (eID) lifetime short
- Avoid linking two different workspaces on the same node
- Ensure that each sub-path of a message contains at least one trustworthy node or two non-collaborating adversaries.

A routing node may further improve the effectiveness of the protocol by…

- Create credible decoy content.
- Use different addresses on the transport level for sending and receiving..
- Use long host keys.

29 Analysis of Degree of Anonymization *MessageVortex* in Comparison to other Systems

It is difficult to make a clear statement in terms of anonymity. To allow a comparison, we work with traditional anonymization systems and compare them to our system and outline the differences.

29.1 Comparing *MessageVortex* to Remailers

All remailer systems are identifiable due to their traffic. We leave aside simple remailer types such as nym remailers and concentrate solely on the most advanced type-3 remailer (Mixminion). Although development has been seized, we can still compare our system's effectiveness compared to a Mixminion system.

Mixminion is an onion routing system working with a fixed message size of 32KB. It relies on a central directory containing all nodes. This is a functionality we can rebuild with *MessageVortex* by using the *decrypt* operation. Unlike with the *MessageVortex* system, Mixminion relies on a public directory. Even compared with a hypothetical system having steganographically hidden services and only locally known routing nodes, our system scores in the following way over a Mixminion router:

- No detectability due to timing-related constraints.
  Mixminion had no synchronized timing constraints. Messages were sent as soon as the subsequent message node was known and the message decrypted. This behavior makes a node identifiable as it is not a common pattern.
- No detectability due to constant sizing of the messages.
  Messages were equally sized into 32KB chunks.
- No identifiable goal in the case of identified messages by a global observer.
  An observing adversary able to match message sizes may identify messages in a low traffic network and identify sender and recipient.

- Possible resistance against a Byzantine node. A Byzantine node would disrupt communication in a Mixminion system. On the other hand, *MessageVortex* may compensate for such behavior with the possibility of redundant data.
- Possibility of redundant routes (*addRedundancy* operation or just redundant message transfer).
  Messages were assembled according to their sequence number. The possibility of redundant routes is not foreseen in the protocol.
- Possibility of monitoring successful delivery.
  Within the Mixminion protocol, there are no means defined to get delivery reports.
- Able to build a localized trust relationship for routing nodes over time.
  As we have the possibility to identify successful message delivery within *MessageVortex*, we may build a localized trust.

Both protocols share some common strengths, such as the possibility to specify routing nodes by the sender of a message or the possibilities of reply blocks.

Having the possibility of mimicking a type 3 remailer and improving the communication scheme makes our system superior to such an improved type 3 remailer. An unmodified type 3 remailer cannot work in an environment with a censoring adversary as defined in this work due to its detectability.

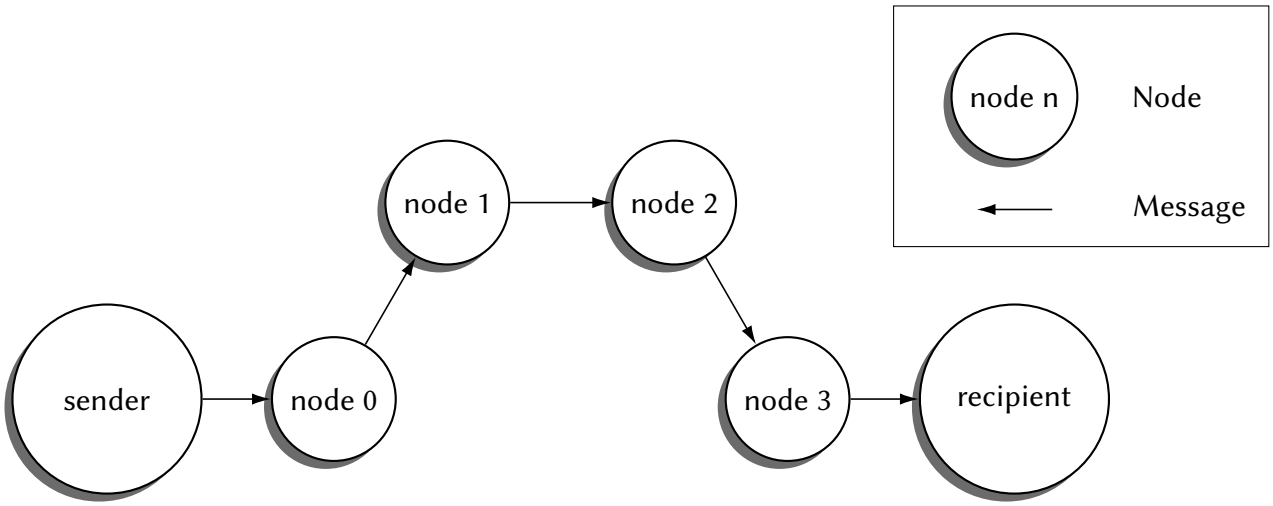

Figure 29.1: A typical Mixminion mix cascade.

29.2 Comparing *MessageVortex* to a DC Network-Based System

A DC network may not be built with our *MessageVortex* protocol. However, if we add a hypothetical *XOR* operation, we may build such a system. In the early stages of development, we had an *XOR* operation. When analyzing our system, we were unable to discover good use-cases for such an operation. We assume that the sender and recipient are part of the DC network ring. If not, additional problems regarding entry and exit nodes would prevail.

When comparing such a hypothetical *MessageVortex* system with an *XOR* operation with an again hypothetical DC network using steganographically hidden messages, we conclude that *MessageVortex* could mimic the behavior of such a network. Making further comparisons along those lines, we have to say that *MessageVortex* scores in the following ways over such a hypothetical system:

- No detectability due to timing-related constraints.
  DC networks send messages as an immediate result to the subsequent node. Messages are assumed to be sent as soon as the subsequent message node is known, and the current message has been processed. This behavior makes a node identifiable as it is not a common pattern.

- No detectability due to constant sizing of the messages.
  Messages in a DC network may or may not be fixed in sizes. They have, however, a constant size per round. This constant sizing makes involved nodes identifiable.

- Possible resistance against a Byzantine node. A Byzantine node would disrupt communication in a standard DC network system. On the other hand, *MessageVortex* may compensate for such behavior with the possibility of redundant data.

- Possibility of redundant routes (*addRedundancy* operation or just redundant message transfer).
  Messages are transferred as a block. The possibility of redundant routes is typically not foreseen in DC networks.

- Possibility of monitoring successful delivery.
  Within DC networks, there are no means defined to have delivery confirmations.

- Able to build a localized trust relationship for routing nodes over time.
  As we have the possibility to identify successful message delivery within *MessageVortex*, we may build a localized trust.

Additionally, in a typical DC network, the set of involved nodes is fixed and known. This leads to the problem that the discovery of one network node leads to the full discovery of a ring or even more. If the sender and receiver are not part of the DC network ring but use entry and exit nodes, the involved parties' discovery is even simpler as a global observer may focus on these nodes' traffic.

Having the possibility of mimicking a DC network and improving the communication scheme would make our system superior to such an improved DC network. An unmodified DC network cannot work in an environment with a censoring adversary as defined in this work due to its detectability. The ring-like communication pattern, as described in **??** is very uncommon in standard Internet protocols and thus easily detectable by a global observer. In an observing adversary environment, the peer partners may not be anonymous due to their traffic from and to the DC network ring.

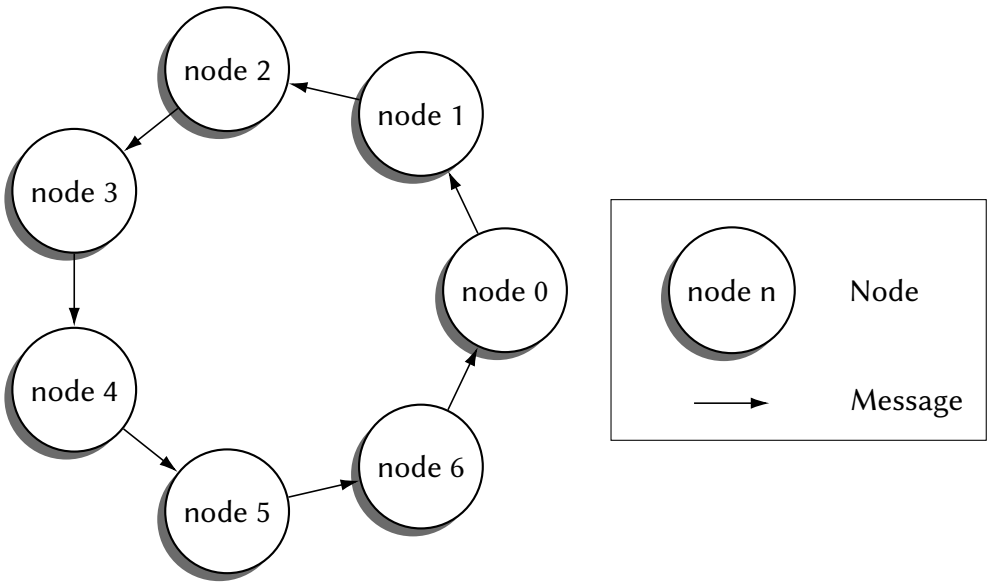

Figure 29.2: A typical DC network communication pattern.

29.3 Comparing *MessageVortex* to a Broadcast-Based System

A broadcast-based network (BCN) hides a message transfer so that every involved node sends an equally sized message or decoy traffic to all other members of the system. By doing so, they build a full mesh of equally sized messages between all involved nodes, making it impossible for an adversary to identify who was sending message traffic and who was sending a true message. Messages sized larger than a simple message transmission are split up into multiple messages. Again, we were unable to find a system that does not use an own censorable protocol. We therefore,assume again a hypothetical BCN piggybacking a common Internet protocol. In this part, we make two comparisons: One comparison involving a BCN with only one mesh and a second one with a BCN cascading multiple overlapping BCNs, which we refer to as a cBCN. In both cases, the communication mesh, as shown in **??**, is identifiable as this is a very unusual communication pattern in common Internet protocols.

A BCN is a high-load network with a very suspicious and uncommon communication pattern in Internet protocols. *MessageVortex* may rebuild such behavior by crafting routing blocks that trigger such a message pattern. Assuming no overlapping pattern, it would always expose the sending node as the first node sending a message. In such a case, privacy would be equal with a single peer broadcast, as shown in **??**. When assuming an overlapping pattern, a first node is no longer identifiable when using a full mesh in the case of a *VortexMessage*. Atom [**kwon2016atom**] cascades multiple BCNs into a cBCN. To achieve a cBCN, Atom relies on a central directory infrastructure. Additionally, to a simple BCN, Atom offers zero-knowledge proofs.

In the following section, we compare *MessageVortex* mimicking a BCN to a traditional BCN. We assume again that the transport layer is steganographically secured comparable to *MessageVortex*.

In such a case, we may conclude that *MessageVortex* scores over a BCN...

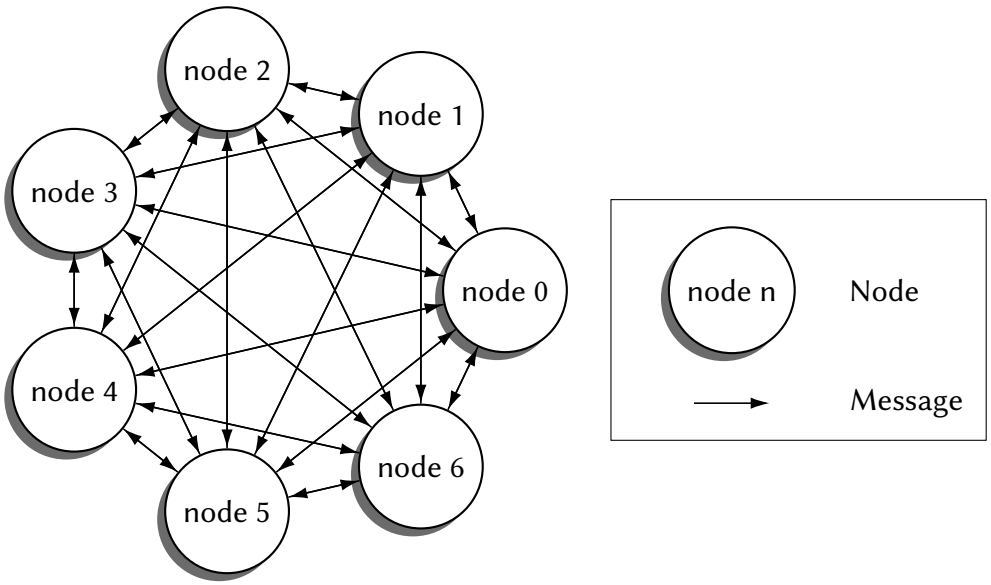

Figure 29.3: A typical broadcast network communication pattern (full mesh).

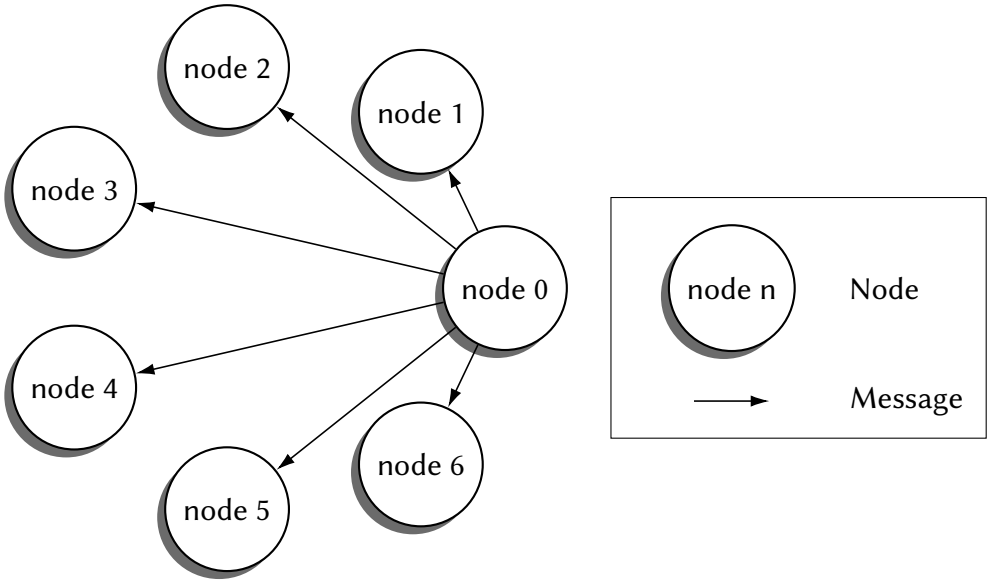

Figure 29.4: A reduced broadcast network communication pattern (single broadcast).

- Equal detectability due to timing-related constraints.
  If *MessageVortex* is mimicking a BCN, timing is essential. Therefore, detectability remains the same for both systems. We could argue that a *MessageVortex* could mimic an adapted version of a BCN not working in epochs and just mimicking the communication pattern. While this would make a difference in traceability, it does not affect detectability positively or negatively.

- No detectability due to constant sizing of the messages.
  Traditional BCNs have fixed message sizes. *MessageVortex* may mimic the communication pattern with or without such a restriction. The message size may not leak any properties when using *MessageVortex*. Messages may travel in parts or as a whole piece.

- Possible resistance against a Byzantine node. A Byzantine node may disrupt communication in a standard BCN by flooding the network. On the other hand, *MessageVortex* may compensate for such behavior with the possibility of redundant data. If we assume that a Byzantine node is not flooding the network completely, a BCN is likely more

robust than a network of *VortexNodes*. A BCN will score at least better if the direct path between the sender and recipient is not affected by the Byzantine node.

- Possibility of redundant routes (*addRedundancy* operation or just redundant message transfer).
  Messages are transferred as a block. The possibility of redundant routes is typically not foreseen in a BCN.

- Possibility of monitoring successful delivery.
  By default, a node has no means to observe successful delivery in a BCN. Using *MessageVortex*, we may do this either by implicit or explicit diagnostic covering one or more epochs.

- Able to build a localized trust relationship for routing nodes over time.
  As we can identify successful message delivery within *MessageVortex*, we may build a localized trust. A traditional BCN lacks this possibility. Extended networks such as Atom may surpass this limitation.

To conclude, *MessageVortex* may offer at least the same properties as a BCN or cBCN. Unlike Atom, which is unable to offer zero-knowledge proofs. As an alternative, *MessageVortex* offers diagnostics. By using multi-path message transfer, *MessageVortex* may reduce bandwidth waste and improve throughput. These capabilities of *MessageVortex* come at the price of local node storage, complex routing operations, and RBB strategies. On the other hand, processing and scalability of *MessageVortex* do surpass Atom's capability by far as the rather complex processing of Atom is very limiting.

30 Recommendations on Using the *MessageVortex* Protocol

The following sections list recommendations using the *MessageVortex* protocol. It is a summary of the previous sections.

30.1 Reuse of Routing Blocks

Routing blocks should not be reused if avoidable. The reuse of a routing block may leak some limited information to an adversary node, such as the approximate message size or message frequency of an unknown tupel using this network.

When using MURBS with a high replay count, a traffic pattern may be identifiable in the network used and thus allow an adversary to identify a message flow.

30.2 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, linking multiple nodes may leave a trace from one identity to the next. It is advisable to recreate regularly all ephemeral identities from scratch. This guarantees an unlinking from previous ephemeral identities.

30.3 Recommendations on Operations Applied on Nodes

All operations carried out on a single node have to be crafted so 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. Again, we emphasize 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. However, such an operation is 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 should be avoided or limited to an absolute minimum. Such an occurrence allows the linking of two ephemeral identities.

30.4 Reuse of Keys, IVs, or Routing Patterns

An RBB should avoid the 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.

30.5 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 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 **??**).

30.6 Message Content

Although it is possible to embed any content into a *VortexMessage*, 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 plaintext 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 carrying out a reverse lookup on an IP address may leak valuable information.

30.6.1 Splitting Message Content

Message content may 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.

To split a message into multiple parts and add redundancy information simultaneously, the *addRedundancy* operation should be used instead.

30.7 Routing

The basics of routing are described in **??**. We collect in the following sections the recommendations regarding the routing strategies.

30.7.1 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 so 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.

Furthermore, predeployed diagnosis blocks within the message path are a good possibility to simplify the possibility of explicit routing diagnosis.

30.7.2 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. Thus, 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.

30.7.3 Anonymity

Anonymity is greatly dependent on the routing block's quality and the chosen anonymity set for a single message and a communication tuple over time.

30.7.3.1 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 accountable 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.

Discussion and Conclusion

***

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

In this chapter, we outline the main results of our work. We emphasize the weaknesses and concentrate on the technologies able to complement our new protocol.

31 The Achieved Properties of the Protocol

31.1 Measuring up to the Requirements

This section analyzes the level of achievement concerning the requirements defined in **??**. We will elaborate on ach requirement and discuss the level of achievement. In case of failure, we highlight reasons for the failure and elaborate on the consequences of the current flaws. An overview of all requirements can be found on **??** on page **??**.

In our opinion, our system meets the requirement **??** as long as the blending layer obeys the criterion opposed to it. Assuming that the dummy content is not distinguishable from other traffic by a censoring adversary and F5 is not broken, then a *VortexNode* should be truly undetectable from the outside.

The requirement **??** is met as there is no difference in the nodes. All nodes serve as possible endpoints, and all nodes carry out routing. There is no technical difference between the nodes, which may allow differentiation between endpoints and anonymity routers.

The requirement **??** is met in a wider sense. We do not require any trust in any routing nodes. However, we need either message traffic to trusted nodes from a non-cooperating adversary or an honest *VortexNodes* with additional traffic within our anonymity set.

The requirement **??** is under full control of an RBB. The RBB controls the number of hops and the nodes involved. He may therefore achieve unlinkability by combining the operations accordingly.

The requirement **??** is met if not assuming an adversary within the system. It furthermore can be accomplished in various grades (k-Anonymity) by the RBB if an adversary is within the system running nodes by the RBB. In such a case, all independent message paths of a *VortexMessage* must contain at least one honest *VortexNode* with additional traffic. As soon as this condition is true, an adversary can no longer conclude any potential anonymity set. Even the sender and recipients alone may be sufficient, assuming additional traffic is being routed through these nodes.

The requirement **??** is fulfilled as all elements required for accounting have an expiration date. Requests beyond that date are discarded. The information to be kept is limited to an absolute minimum and may accommodate multiple 100K identities per *VortexNode*.

The requirement **??** is fulfilled as no pattern may be followed through the network. All information visible to an outside or inside observer is discarded at the following node.

The requirement **??** is only partially met. As we did not specify the type of payload that may be transported but suggested MIME-encoded messages, bugging is possible. However, message bugging in MIME-encoded messages is well known. Most of the clients offer appropriate countermeasures such as suppression of external imagery loading and similar. When sticking to the recommendation to send text-only messages, bugging is not possible.

The requirement **??** is only partially fulfilled. As we allowed the use of MURBS, replaying a message is possible. As it is an optional feature and normal messages have replay protection, this flaw's impact is minimal and intended.

The requirement **??** remains one of the major flaws of our protocol. This flaw has systemic reasons. The possibility of discovering *VortexNodes* in an environment of a censoring adversary, no matter how built, is enabling the adversary to harvest a network of nodes. It subsequently means that any possibility of narrowing down potential nodes may be hazardous in such an environment. We believe that, unless we have broadly accepted protocols using broadcast into huge domains, a protocol may not solve the problem of identifiable peer nodes. We, furthermore, believe that such protocol support is unlikely to arise due to bandwidth reasons.

The requirement **??** is met as we have built in the possibility to vary any algorithm. Wherever possible, we named and included independent alternatives based on different mathematical problems into the standard. It is furthermore possible to signal non-standard algorithms. As long as two nodes support the same standard, they are capable of communicating.

In our eyes, the requirement **??** turned out to be the least successful of all. While we may automate the *MessageVortex* protocol and all its needs in an observing adversary environment, the use in an environment of a censoring adversary is not possible for a non-savvy individual or a small organization. This is because the dummy traffic generated to carry *VortexMessages* has to be individualized and credible. Coding skills are required to meet this requirement, which opposes to **??**. While not unsolvable, we consider this problem as difficult to solve.

The requirement **??** can be met in various degrees. The degree of reliability depends on the number of stable working nodes in an anonymity set and the strategy chosen by an RBB to build the routing block. We consider this requirement as met.

The requirement **??** is met in our opinion as we offer the possibility to explicitly or implicitly diagnose the entire message traffic at any time.

The requirement **??** is met as our system remains functional via alternate message paths if a *VortexNode* is no longer functional. However, as we cannot adapt a messages' route, the system's availability is controlled by the RBB.

We consider the requirement **??** as met as the protocol offers the possibility to match two messages to the same sender (even if not knowing its identity) by matching the eID. To keep this possibility for a recipient, both sender and receiver have to collaborate as the sender needs to use the same eID for all messages, and the recipient must allow usage of such eID for the entire period.

Overall, we consider this work in its current state as a partial failure due to the lack of the requirement **??**. This miss causes the protocol to be only of limited use to a single individual operating in a censoring adversary environment.

31.2 Achieved Level of Anonymity and Detectability

We have to emphasize when discussing anonymity that our system is unlike most other systems. As we have an adversary defined that other systems do not withstand, we have to compare anonymity on multiple levels. Within these levels, anonymity and detectability complement each other as breaking detectability might lead to a node or a respective user's de-anonymization.

These layers relevant to anonymity or detectability are:

- The detectability of the system by…

  - detecting or identifying transport layer accounts.
  - detecting or identifying *VortexNodes*.
- The detectability and tracing of single *VortexMessages*.
- The traceability of a message over multiple *VortexNodes*.
- The identification of *MessageVortex* users by...
  - the sending *MessageVortex* user
  - the receiving *MessageVortex* user
  - an adversary within the anonymity set
  - an outside adversary

The detectability of a system depends on multiple factors. If the blending is detectable, a *VortexNode* is identifiable and may uncover the respective user. In environments with a censoring adversary, such identification may be deemed as dangerous. In such environments, our system heavily depends on the individual implementation of the blending. In its current state, coding skills are likely needed to remain undetectable as not following a pattern is their key, and our standard implementation may be deemed a pattern. The traffic generated for accessing a transport layer account is not especially susceptible to detectability as the always-connected pattern is very common among devices and services these days. Special care has to be taken if protocols offer housekeeping features for the transported messages. In these cases, access patterns should match the chosen service pattern (e.g., delete INBOX emails after 30 days). A node may be identifiable by the transport layer owner as an atypically behaving user by not doing so.

Single *VortexMessages* may be detectable from the outside, as covered in the previous paragraph. Apart from that, sending and receiving *VortexNodes* are always aware of the transport layer address's identity. This means that anonymity is no longer possible if a censoring adversary is part of an anonymity set. In such a case, the adversary would be capable of uncovering involved *VortexNodes* by harvesting node transport addresses over time. A solution for this problem does not exist as long as we do not assume a widely deployed protocol employing broadcasting (or at least multicasting) with huge domains.

Messages are not traceable as long as we have at least one honest or non-collaborating (to the current adversary) *VortexNode* in a message path due to the message properties. As soon as two adjacent *VortexNodes* collaborate, they may collapse all operations of the two into one workspace.

Identification of *MessageVortex* users may be achieved in multiple ways. If an RBB composes unsuitable routing blocks, anonymity is broken. We outlined before that *MessageVortex* may build the same messaging patterns as Mix-, onion-, BC- or DC-networks but with additional security-related features such as redundancy or the split of messages. In general, this makes our protocol at least equivalent or even superior to the technologies mentioned earlier. Unlike those systems, our system is not limited to specific message patterns, making our system more suitable in an environment of a censoring adversary. A unique fingerprint of composing messages may identify the sending user. He furthermore may be identified by bugging a message sent with a reply block. For an outside observer, a sending user may be determined if there is no additional traffic running over its routing node. Therefore, receiving traffic (to be routed or not is irrelevant) adds to a message's anonymity. The receiving user may be identified by a bugged message. From the outside, a receiving user may or may not further

deliver messages. The same applies to any routing node. This does not give any indication of a received message.

32 Weaknesses of the Protocol

The protocol has several weaknesses which we were unable to compensate accordingly. The complexity of the algorithms for an RBB is definitely high compared to other protocols. Nevertheless, it is possible for a single RBB to create and maintain a network of ready eIDs for routing. Given a sufficient set of nodes, this routing works comparable to other protocols. It scales very well under high loads as all nodes act independently, and no non-parallelizable asks are within the whole system. However, once adequately bootstrapped, it is easy to use as a user may use it with typical clients such as email clients and offers an unmatched degree of anonymity in our belief.

33 Missing Research

33.1 Lack of Base Data

One problem we encountered is the lack of available statistical data regarding true Internet environments. There is much data available that may be easily extracted (such as SNMP MIBs). However, when it comes to true insights into the Internet, we have only very limited data. There is some data available about censorship in China and in Turkey in our specific case. It would have been tremendously welcomed if we had comparisons in the communication patterns of persons. Questions about “What protocols are used to transfer messages either in human-to-human or machine-to-human communication”, “Which types of attachments are common among specific protocols”, or “What are common threats today” seem not to be researched. There are some pseudo-scientific papers available, shedding light on some questions. However, these papers do not follow scientific standards and are often misunderstood to boost certain products. An excellent example of this trend are papers describing the dangers solely from the perspective of anti-malware or firewalls, which typically fail to list threats related to social engineering. Available data is often collected cheaply by querying SNMP MIBs, using statistics collected by a product cloud, or by filtering traffic of static sources list to identify streaming traffic. Continuously monitored and generally available data about routed traffic within the Internet would have offered tremendous help for our work.

33.2 Lack of Implementations

One of the actual weaknesses of the protocol lies in the lack of implementations available for anonymity. Available implementations of steganographic algorithms in C/C++ or Java are rare. Moreover, we were unable to find any partial essay of implementation for creating dummy traffic. Therefore, one weakness may be found in the lack of adaptation of protocols and algorithms from the scientific world. Most of the anonymity systems exist only as partial implementation or as simulators. Especially an alternative available to the implementation of F5 would be sensible and helpful. While such an implementation may be retrofitted in the system, the lack in the current state is regarded as a weakness. The same may apply to

algorithms such as NTRUencrypt. While this algorithm was implemented and specified in terms of encryption and decryption, a binary layout for the key was never specified. Such layouts are, however, crucial for a world of inter-operation. The lack of such specifications and implementations makes our implementation of *MessageVortex* weaker in portability. We are, of course, capable of creating our implementation and specify our binary layouts. However, such implementations lack a proper peer review and violate interoperability basics, which are a major concern in all protocols.

The lack of other, comparable protocols makes the *MessageVortex* protocol weaker. Having no real competitor in a class makes it very difficult to measure and compare a solution. Assuming a censoring adversary is a hard-to-fulfill territory, most people instead seem to focus on a single problem without true implementation than on a solution for a real-world problem. Claiming that anonymity is solvable is acceptable in the authors' eyes as long as we can describe realistic real-world or clean slate approaches, and these approaches must be implementable. The authors encountered multiple solutions, which were good ideas but lacked a realistic view. Achieving in an environment where there is no inside observer or just regional observers is straightforward but not realistic.

33.3 Further and Missing Research

The current blending layer is by far too simple in its inner workings. It creates contextless messages based on an easily recognizable scheme and is not suitable to mimic human communication. A good blending layer would be capable of mimicking not only machine-like traffic but even human-like traffic. Atypical communication patterns such as 24x7 communication may be broken into typical patterns by mimicking three sending accounts with different overlapping communication patterns. The system does not necessarily have to pass a full Turing test. It would be sufficient to create credible human communication between machines sounding human-like. Research in AI already succeeded in generating credible communications between two robots. It is unknown whether such "small-talking" implementations would create credible content. As we defined that an adversary has enormous but limited resources, this blending is sufficient if it is carried out "good enough" so that an adversary cannot identify the traffic as generated content. What criteria would apply here is a topic for further research. Applying more research to this topic would require adding 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. Of special interest are steganography algorithms covering movie and audio file formats. This may be especially interesting when it comes to mimicking other communication patterns such as social network apps using voice messaging.

Anonymity has effects on the behavior of humans. We have found that although there is some research in this field (such as [**postmes2001social**]), the evidence is very weak. Although the possibility of anonymity is undisputed among so-called free countries, the disadvantages (e.g., misuse for criminal acts) of anonymity are apparent. More research in this field is required. On the other hand, a lack of anonymity awareness, especially in

“non-free” jurisdictions, has been observed, which would be another relevant field of research.

34 Potential and Improvements

34.1 Improvements in Blending

Our current implementation is very rough and requires coding or individualization when used in a censoring adversary environment. Generating the decoy traffic should be far better feasible by using recent developments in deep learning (DL) and natural language processing (NLP).

Such implementations would have the potential of generating undetectable decoy traffic. While the current traffic is bound to machine-to-human communication, deep learning implementations would have the possibility of building proper communication between two artificial identities. The implementation would not have to pass a Turing test. Instead, it would be sufficient if an outside observer cannot identify the communication partners as “non-human” or “suspicious”.

34.2 Operations Agility

In our current implementation, operations are statically encoded. While the current set was chosen carefully, it would have been better to, analogous to the requirement of crypto agility, select a set of supported operations. Such selection possibility was forgotten at the start, and adding it to the work’s current state turned out to be very challenging. Nevertheless, we believe that such “Operations Agility” would add to the system’s value.

It would allow extending the system with new types of operations reflecting state-of-the-art anonymity research without disrupting an existing network.

34.3 Simplified and Anonymity-Conformant Bootstrapping

Bootstrapping is currently based on human-to-human communication. While this is possible and, in most cases, feasible, it is impractical and reduces the ease of use of the system. The handshake forces us to exchange transport endpoint addresses and node keys. We could simplify our approach by introducing decentralized stores offering SURBs if a short common secret is known. Analogous to a PIN when using WPS in a WiFi system, such small secrets could be used to do the first handshake simplifying the tedious procedure a bit.

Such an approach will be secure if the rendezvous-point is under the control of an observing adversary, as only the common knowledge of both short secrets allows the identification of the SURB. By trying to brute-force the SURB, an adversary would invalidate the SURB on its first use.

35 Closing Words

While working on our system, we were amazed at how broad the field of anonymity and the number of means to attack anonymity is. Anonymity is, in our belief, achievable in any environment. Depending on the type of anonymity and environment, it has a relatively high price tag for the user. It will always be more comfortable to remain traceable than to be anonymous. It is up to all researchers in the field of anonymity to reduce this pricetag. In our belief, this is a topic research has to pursue in subsequent works. Our statement here would be: Challenge accepted.

Our tool is neither good nor bad. Precisely as a crowbar is a useful household tool, it may be misused to carry out illegal things or threaten life. On the other hand, recent development in many countries shows that there is always an excuse for legislative power to intimidate people not in favor of their opinions. Therefore, it is our firm belief that despite the inherent disadvantages of all anonymity systems, they are necessary to keep at least the world as free as it already is.

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-08
Published: 5 April 2021
Intended Status: Experimental
Expires: 7 October 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 the 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 7 October 2021.

Copyright Notice

Table of Contents

1. Introduction

Anonymization is difficult 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 that 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, all routing nodes are 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 Appendix 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 by themself or provided by the receiver to perform the transmission. While a message may be anonymous, there are different degrees of anonymity as described in 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 have 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 peerpartner:

- A transport address (always) and the public key of thisidentity, 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 version 1 of the protocol. The message PrefixBlock contains an optional version indicator. If the protocol verion is absent protocol version 1 should be assumed.

3. Layer Overview

The protocol is designed in four layers as shown in Figure 1.

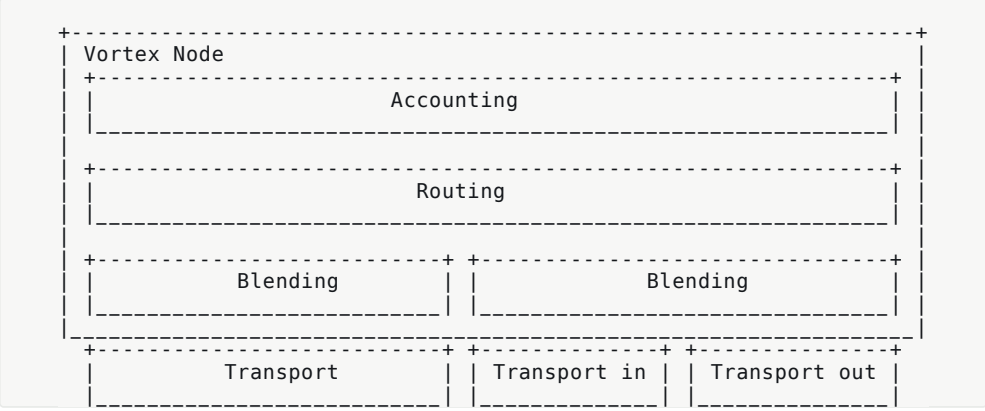

Figure 1: Layer overview

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 that 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.

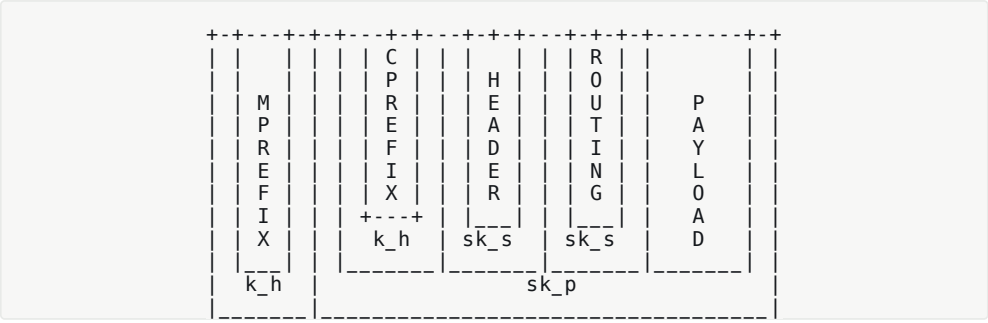

Figure 2: Vortex message overview

4.2. Message Prefix Block (MPREFIX)

The PrefixBlock contains a symmetrical key as defined in Appendix 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. 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 that 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 larger 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 10).

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 of 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., MIMEtype) 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 chunk header. The chunk header consists of two unsigned integers of variable length. The integer starts with the LSB, and if bit 7 is set, then another byte follows. There cannot be more than four bytes whereas 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 remaining document bytes) to be skipped after the current chunk to reach the next chunk. There is no "last chunk" indicator. A gap or chunk may surpass the end of the file.

```
pos: 00h   02h   04h   06h   08h...400h  402h  404h  406h  408h  40Ah
val: 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 follows.

```
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 a duplicate according to reply protection, then a node MUST NOT decrement the message quota.

The message processing works according to the 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 linked identity, 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 cleartext requests if the node does not want to officially disclose its identity as a Vortex node. 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 ephemera 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 eaderRequestIncreaseTransferQuota 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 parsable VortexMessage. Any node SHOULD reply to any encrypted HeaderRequestCapability.

A node SHOULD NOT reply to cleartext requests if the node does not want to officially disclose its identity as a Vortex node. 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, unlinked 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 receives 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 replies 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 it occurs within the 'valid' period but is 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 unit 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 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. 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.

10. 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. In addition, 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, 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 simple 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 workings. Each is aware of 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 the 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 header) to de-anonymize a recipient. Careful attention is required when handling anything other than local references when processing, verifying or rendering a message.

11. References

11.1. 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>.

11.2. 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
  maxNumOfRequests        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
  maxRoutingBlks          INTEGER ::= 127
  -- maximum number a block may be replayed
  maxNumOfReplays         INTEGER ::= 127
  -- maximum number of payload chunks in a message
  maxPayloadBlks          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
  maxAssemblyInstr        INTEGER ::= 255


  --****************************************************************
  -- Types
  --****************************************************************
  PuzzleIdentifier ::= OCTET STRING ( SIZE(0..32) )
  ChainSecret      ::= OCTET STRING  (SIZE (16..64))

  --****************************************************************
  -- Block Definitions
  --****************************************************************
  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
  }

  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 OPTIONAL,
    -- 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
                         (SIZE (1..maxNumOfReplays))
                       OF SymmetricKey OPTIONAL,
  -- contains the next recipient
  recipient      [402] BlendingSpec,
  -- PrefixBlock encrypted with message key
  mPrefix        [403] SEQUENCE
                           (SIZE (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
```

```
  }
END
```

A.2. The MessageVortex Ciphers Structures

```
MessageVortex-Ciphers DEFINITIONS EXPLICIT TAGS ::=
BEGIN
  EXPORTS SymmetricKey, AsymmetricKey, MacAlgorithmSpec,
          MacAlgorithm, CipherSpec, PRNGType;

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

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

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

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

  SymmetricKey ::= SEQUENCE {
    keyType       SymmetricAlgorithm,
    parameter     AlgParameters,
    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)
}
AlgParameters ::= 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 {
  cbc           (10000), -- required
  ctr           (10001), -- required
  ccm           (10002), -- optional support
  gcm           (10003), -- optional support
  ocb           (10004), -- optional support
  ofb           (10005), -- optional support
  xts           (10006), -- optional support
  none          (10100)  -- required
}

CipherPadding ::= ENUMERATED {
  none           (10200), -- required
  pkcs1          (10201), -- required
  rsaesOaep      (10202), -- optional support
  oaepSha256Mgf1 (10203), -- 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     AlgParameters
  }

  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 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] OCTET STRING OPTIONAL
    }

  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,
    relative [3] RelativeUsagePeriod
  }

  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 and added more precise spec about blending. Change in spec for XMPP blending (from XEP-234 to XEP-231). Restructured ASN.1.
2	03-2019	Language and consistency improvements. Added example for chunked plain embedding. Added pseudo-code for incoming 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 improvements.

Version #	Date	Changes
5	09-2020	Reinserted lost ASN.1 specs (unintentinally lost in last two versions). Added changelog. Modified padding to improve credibility of bad values.
6	02-2021	Removed some outdated references and updated draft according to the final research document. Refining of language.
7	04-2021	Lectorate and improved rendering.

Table 1: changes in versions

Author's Address

Martin Gwerder
University of Applied Sciences and Arts Northwestern
Switzerland
Bahnhofstrasse 5
CH-5210 Windisch
Switzerland
Phone: +41 56 202 76 81
Email: rfc@messagevortex.net

B Glossary

adversary In this work, we referr to an adversary as any entity opposing to the privacy of a message. For a more throughout definition, refer to section **??**

anonymity We refer to the term anonymity as defined in [**anonTerminology**]. "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 containing 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 *VortexNode* 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 node, 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 Rights). Due to their overlapping nature, multiple jurisdictions may have contradictory rules applying for the same event.

IMAP IMAP (currently IMAPv4) is a typical protocol used between a Client MRA and a Remote MDA. It has been specified in its current version in [**rfc3501**]. 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 emails to other destinations outside the server-based store.

[1] footnotes omitted in quote

ID A numerical identification reflecting a single payload chunk in a workspace of an eID.

IoI The Item of Intrest (IoI) are defined in [**anonTerminology**] 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 [**rfc2033**]. 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 typically supports a specific access protocol to access the data stored within a local message store.

Local MDA A Local MDA typically gives local applications access to a server store. This may be done through 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 [**ms-oxcmapihttp**], 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 that 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 type of routing block is provided by a sender to give a

node the possibility to route back answers without the knowledge of the location of the sender. In contrast to a SURB, a MURB may be used multiple 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 larger 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 operations 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*.

payload block A single block attached to a *VortexMessage* representing either the message or the decoy content.

privacy From the Oxford English Dictionary[**OXFORD**]:

> “
>
> 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.
>
> ”

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 [**anonTerminology**].

> “ 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 feasible 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 furthermore contain additional 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 **??**.

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 to both the human creator or initiator of a message, as well as the process of assembling and preparing the message.

immediate sender The actually peering sender. This is the sender which sent the current message.

server admin We 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 [**rfc5321**].

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 the 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 [**anonTerminology**].

> “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 [**anonTerminology**]. “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 [**anonTerminology**].

> “ 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 (⇒ reads “implies”):

$$\begin{aligned} \textit{censorship resistance} &\Rightarrow \textit{unobservability} \\ \textit{unobserability} &\Rightarrow \textit{undetectability} \\ \textit{unobserability} &\Rightarrow \textit{anonymity} \end{aligned}$$

user Any entity operating a *VortexNode*.

VortexMessage The encoded message passed from one *VortexNode* to another. 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.

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)[**rfc6120**, **rfc6121**] 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.

C Bibliography

E Index

E Short Biography

Martin Gwerder was born 20. July 1972 in Glarus, Switzerland. He is currently a PhD 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 security-related problems dealing with data protection, distribution, confidentiality, and anonymity.