A System for Hiding Steganography in Plain Sight

Andreas Toftegaard

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Recent leaks have shown that Dolev-Yao-type adversaries are no longer far fetched within online communication. Major governments are not only capable of monitoring the Internet, they can also manipulate, analyze and catalog both metadata and content for later sophisticated searching. Adversaries of this strength, are only kept at the gates by strong cryptographic algorithms, used for encrypting secret messages. The counteracting trend is quickly becoming to encrypt everything by default. This approach successfully prevents messages from being read by unintended recipients or eavesdroppers, but still leaks the fact that secret communication is happening between involved parties. This may lead to regulations on keys, and users cannot plausibly deny their usage of the system. It is not hard to imagine scenarios where this alone, can cause great trouble to one or both involved parties. Hiding existing communication is a problem solvable by steganography, and not traditional cryptography. This thesis will present a steganographic implementation of a system, that allows for hiding secondary communication within harmless messages. It prevents unauthorized parties to detect communication in a secondary channel. It will modify traditional emails to always contain a stego-object, which may or may not contain a message. The steganographic element and strength lies within the fact that all emails from this client may or may not contain hidden information. System strength is increased as it becomes ubiquitous. If the user does not specify a message for the steganographic block, one will be chosen at random - providing the sender with plausible deniability of secondary communication. In particular, the project implements a plugin for a common email client, allowing users to embed size-limited secondary messages into normal emails. Emails formed by said plugin will always contain a stego-object, that might contain a secondary hidden message. This project revolves around the development of said plugin, and the theory supporting the specific idea and use of steganography. A proto-

type is developed, tested and reasoned about. This prototype proves that our basic idea of always embedding a steganographic block, indeed hides existing secondary communication. This approach provides the user with a very valuable plausible deniability. Supporting steganographic theories will need to be explored, to form a reliable foundation. Plugin development has reached a stage where it is ready for distribution, although some future work has been identified.

Preface

This thesis is prepared at DTU Compute in fulfillment of the requirements for acquiring an M.Sc. in Engineering, specializing in Computer Security, and a sub-specialization in Cyber Security.

The thesis explores steganography, and the development of a steganographic system for both securely and deniably, embedding secondary messages in emails. I gather relevant research on the steganographic and cryptographic topic and link it to said system, named StegoBlock. I evaluate the system with a steganalysis, to establish its security.

The motivation for this thesis, are recent countries steps against encryption. Ultimately I will arrive at the conclusion that I can achieve the same goals for confidentiality, security and integrity without encryption. Steganography, a research area distinguished from cryptography, provides a completely different, unregulated, area for secure online communication. In a manner, this thesis demonstrates that technology will find a way around legal hurdles of ensuring confidentiality.

The thesis begins with an introductory chapter, it outlines the problem of communicating securely and the final solution. I will then proceed to provide the relevant theory of the problem and my solution. This is, naturally, largely within steganography and cryptography. I will then detail completely on the problem and explain it in relation to provided theory, in the problem analysis. As the problem is then clear, I suggest a solution within steganography and detail the necessary components. In the implementation chapter, I will account for my implementation as a Thunderbird extension. Lastly, I present a thorough evaluation in the form of a steganalysis - accounting for the security of said sys-

tem. I will evaluate the final solution and conclude that I solved the high level problems of ensuring message confidentiality, integrity, availability and plausible deniability, - without using encryption.

Lyngby, 31-December-2016

Andreas Toftegaard

Acknowledgements

I would like to thank my wife and kids for support and patience. A thank you to my supervisor Christian D. Jensen, for thorough guidance and counseling. Also thanks to Elmar W. Tischhauser for cryptographic and cryptanalysis counseling.

Contents

Chapter 1

Introduction

From recent leaks by Edward Snowden and others, we have learned that the US intelligence agency NSA, not only has direct access to major online communication companies servers, but has also deployed comprehensive wiretapping of US internet backbones[Coh06]. Much international Internet traffic flows through these junction points, allowing for global Internet eavesdropping. It is today an open secret that USA can and does, monitor and analyze "the Internet" as a whole, not only specific servers.

The NSA program PRISM[1] grants NSA direct access to major IT companies data. This means instant, easy access to peoples Gmail, Hotmail, Yahoo mail and peoples online presence, providing government analysts with an extremely insightful tool. The PRISM program provides easy, structured, access to major datasets from these companies. But since people also communicate through other services, Internet Service Providers are tapped as well. Raw data is collected, stored and structuralized for later analysis. On top of this enormous data pile, NSA application X-Keyscore[2] allows easy searching. Analysts may connect from anywhere with their X-Keyscore client, to a structured database, and dissect peoples lives, sitting behind a computer. Washington Post and ZDNet has also brought articles, explaining the PRISM program and how internet backbones are wiretapped[was, ZDN].

[1] PRISM: `https://en.wikipedia.org/wiki/PRISM_(surveillance_program)`
[2] X-Keyscore: `https://en.wikipedia.org/wiki/XKeyscore`

All internet traffic cannot be stored forever, that would be too costly. NSA is reported to be able to store around 12 exabytes in their Utah Data Center[3]. The NSA harvests almost incomprehensible large amounts of data, from many different sources, e.g: phone, internet, satellite. According to David Adrian et. al [ABD+15] it is very realistic that NSA also breaks some crypto schemes, due to poor implementation or too short key length.

Today it is highly realistic to consider FSB, GCHQ, NSA and other major intelligence agencies, as Dolev-Yao[DY83] type adversaries. In fact, they are even stronger, as they can break some crypto, and perform side channel analysis. Their paper features a formal model for protocol verification, based on their adversary model. A Dolev-Yao attacker can eavesdrop, intercept and synthesize any message in a network. He is, in some sense, the network. The only limitation to this attacker, is a strong cryptographic system. The model treats crypto system as a black box, meaning it cannot analyze or investigate it. Performing side channel analysis or circumventing it in other ways is not possible. The Dolev-Yao attacker has previously been scrutinized for being unrealistically strong for most applications[BZ14, PHGW16], but in the light of information provided by Edward Snowden and other whistleblowers - we learned that this is no longer the case. In fact, we learned that major intelligence agencies are even stronger, as they are not limited to treating crypto as black box.

With StegoBlock we initially aim to:

Enable parties to securely exchange messages, without a Dolev-Yao strong adversary able to eavesdrop or synthesize messages. Nor should he be able to confidently tell if the parties did in fact exchange hidden messages.

For a novel online communication scheme, we would like to have the properties of the CIA triad: Confidentiality, Integrity and Availability. Confidentiality, as we wish to keep our messages private. No one but ourselves and our intended recipient, should be able to learn our messages. Confidentiality can also be referred to as privacy. Without confidentiality, anyone may read our messages. With message integrity, we can say for certain a message was not altered in its transfer. We know we are reading what the sender intended. Integrity does not automatically follow of confidentiality. Unable to read message contents, an adversary may still alter a message. Without an integrity check, the recipient cannot validate the message. Lastly, we must also secure the availability. A recipient must be able to access the message. Keeping a message encrypted on a hard drive stored in a treasure chest, will keep it confident and unchanged - but without no good use. Achieving the first 2 sides of the CIA triad would be

[3]NSA UDC blueprints and estimated capacity: `http://www.forbes.com/sites/kashmirhill/2013/07/24/blueprints-of-nsa-data-center-in-utah-suggest-its-storage-capacity-is-less-impressive-than-thought/#5a9457851c85`

trivial, without the last availability-side. By making a message available - we risk making it available to the adversary as well.

By examining the leaked internal presentational slideshow[4] of aforementioned X-Keyscore (specifically slide 15), used by NSA analysts to search information on specific people, we learn that analysts start by looking for anomalies. Anomalies like: "Someone whose language is out of place for the region they are in", "Someone who is using encryption" or "Someone who is searching the web for suspicious stuff". Persons fitting that description could be terrorists, but also mere employees working abroad, through VPN connections - depending on the definition of "suspicious stuff". So, ideally we would like to avoid seeming suspicious, but still maintain privacy.

Steganography is the science of hiding information in such a way that it does not attract the attention of adversaries. On top of hiding, some algorithms promise also perfect security[Cac04], meaning that an adversary with unbounded computational power will have no advantage. Steganography distinguishes from encryption, which solely promise to protect a message, and no means of hiding communication. One could say that steganography also hides communication metadata. We will explore different types of steganography throughout this thesis, but first argue the need for steganography by introducing the classic "Prisoners problem", proposed by Simmons[SC84]. Even though this scenario depicts alleged criminals, in the process of further criminal acts, other scenarios could easily be thought up. We will use the "Prisoners problem" as the basic example to explaining why steganography can be preferred over encryption.

We consider the usual suspects Alice and Bob, they are locked in widely separated prison cells. Their goal is to develop an escape plan together. They may only communicate through a warden, Wendy. In a simplified scenario, Wendy inspects all communication and will thwart suspicious messages and not allow any encryption. She may also try to alter messages or forge messages from Alice or Bob. In order for Alice and Bob to devise their escape plan, they need steganography, so they can communicate without arousing Wendy's suspicion and to authenticate messages. For instance, they could send letters explaining their favorite food and movies - but by assembling all capital letters, a secondary secret message is formed. Wendy is not in on the scheme, she wouldn't notice. Only Alice and Bob knows, they can communicate in private.

The prisoners problem is interesting, as it relates and easily explains the problem of being watched while communicating. We cannot communicate in the clear, or encrypted without being watched. Like the prisoners, we need a subliminal

[4] X-Keyscore slides: `https://www.theguardian.com/world/interactive/2013/jul/31/nsa-xkeyscore-program-full-presentation`

channel to communicate in private and to avoid reprisals of disagreeing opposers. Alice, Bob and Wendy from the prisoners problem will be recurring subjects in this thesis.

Many citizens face similar challenges to Alice and Bob. Several countries, like Pakistan[5] and Turkey[6] have already taken a hostile stance against encryption. From Edward Snowden's[7] leaks over the years, we know the US is intercepting and inspecting huge amounts of internet communication[8]. Adversaries are becoming Dolev-Yao-like strong, perhaps even stronger, and have the measures to recognize and prevent encrypted traffic. We need steganography to ensure freely, private communication, without worrying of later reprisal.

1.1 Our solution

People are having their communication watched and analyzed by extremely potent adversaries. This is an obvious problem, as we would like to ensure confidential communication between anyone desiring so. We will try to solve this problem, by offering an extension to a popular email client. The extension, which is named StegoBlock, extends existing emails with additional information. StegoBlock can be installed on top of Thunderbird, an email client by Mozilla. By installing, users may continue to use email as normal, but extended functionality will add a subliminal channel to all emails sent. It is no requirement for recipients to use StegoBlock or Thunderbird. They may continue to read primary messages as normal. They are simply unable to process the secondary message.

Users must continue to write emails as usual, but will be able to enter a short secondary message as well. This secondary message is kept confidential and hidden. It will have no effect to users without StegoBlock installed, they will never notice it, unless inspecting email source code. An email consists of headers and body. Metadata and data. We will hide the secondary message in the email metadata, in a specially crafted header. Unless one knows where to look, the message will be hidden.

Users are not required to enter secondary messages, for the subliminal channel.

[5] Pakistan bans encryption: `https://www.theguardian.com/world/2011/aug/30/pakistan-bans-encryption-software`

[6] Turkey charges over encryption software: `http://www.aljazeera.com/news/2015/09/vice-news-fixer-arrested-encryption-software-150901200622345.html`

[7] Edward Snowden: `https://en.wikipedia.org/wiki/Edward_Snowden`

[8] PRISM surveillance program: `https://en.wikipedia.org/wiki/PRISM_(surveillance_program)`

They may choose to not enter one, then one will be picked at random. This random message will not be readable by the receiver, but only serves the purpose of always transferring something in the subliminal channel. By doing so, users are provided plausible deniability, i.e. they can plausibly deny actively putting something in the channel. This is superior to traditional crypto schemes, where encrypted messages always hold some meaningful value. With these, users may be threatened or blackmailed into disclosing their key, as the adversary may validate if they provide the correct key.

Consider Alice and Bob again. Alice is now a whistleblower inside a highly advanced intelligence agency. Bob is a journalist, eager to publish her information of corporate embezzlement. For obvious reasons, Alice wants to stay anonymous. Alice may send an encrypted email to Bob, but even though it may not be breakable, she will have leaked the fact that she communicated with Bob. For this reason alone, Alice exhibits suspicious activity. She might later be blackmailed, threatened or otherwise forced into disclosing her crypto key to her conversation with Bob. Adversaries will be able to tell, if she provides the real key or not.

In the depicted scenario, encryption alone is not enough for Alice and Bob, she will need some form of plausible deniability. We have before hinted at plausible deniability, being a great advantage of steganography. It is in fact one of the major advantages of our StegoBlock application. Plausible deniability refers to the condition, where *a person can, plausibly and legally, deny any knowledge of, or association with some specific reality, in a way that they have deliberately provided beforehand.* This protects the person from undesired repercussions from being associated with said specific reality. Alice can only save herself from potential blackmail and threats by setting up a cover of her communication in advance. She must be able to present something plausible to the adversary, something believable - but obviously not the real message.

Contrary to common practice, we will use steganography for achieving confidentiality, instead of encryption. People living under regimes with encryption bans, will be able to use our solution as a legal alternative. In that way, StegoBlock is a technological work-around, to a naive legal approach. We will use some cryptographic elements to implement StegoBlock, but none that are in fact classified as encryption. Today we see that strong encryption is becoming ubiquitous. Movements like "HTTS Everywhere"[9] tries to make websites default to providing their content over HTTPS. "Let's Encrypt"[10] offers free SSL certificates to everyone. Traditionally certificates would cost an annual fee, preventing some website owners from deploying HTTPS. Now they are free.

[9]HTTPS Everywhere: `https://www.eff.org/https-everywhere`
[10]Let's Encrypt: `https://letsencrypt.org`

Popular instant-message providers like Facebook Messenger[11], WhatsApp[12] and Viber[13] offer end-to-end encryption. Mail providers like Proton Mail offer the same, for traditional emails. The Google Chrome browser will even punish HTTP-only websites, with a visual representation.

Even government funded surveillance must have a hard time catching up, as encryption moves from exception to rule. Some governments, like Pakistan and Turkey, may only have capacity to identify and block applications using encryption, but the need for confidentiality is however still present, perhaps even more. Several countries have bans on strong end-to-end encryption. People may end up in jail for using encrypted messaging services. We can provide people with private communication, and also the ability to plausibly deny any communication with a specific person. But if they still end in jail for simply having our application installed on their device - we consider our solution suboptimal. As encryption is a subarea of cryptography - using other areas of cryptography is considered fine. We will design StegoBlock to conform to the CIA triad, without using encryption and ensuring plausible deniability. StegoBlock users may present plausible deniability to any message they have exchanged with another.

We will perform thorough analysis of the confidentiality and usability of StegoBlock. We will reason that messages are in fact confidential, even without encryption. We will ensure that all encoded StegoBlock messages always adhere to some target character distribution - making it infeasible to reason about its contents by statistical analysis. We argue that our embedding method is so strong, that reversal without knowing the key is infeasible. The target distribution combined with a fixed length, imposes a max length on the secondary message. We will argue that we can still encode a reasonable amount of messages, by analyzing and testing large amounts of real world emails.

1.2 Scoping

Our StegoBlock application belongs somewhere in the gray zone between cryptography and steganography (but certainly not encryption). Some cryptographic primitives like random number generators will be used. We will detail on these, we will argue that we use them correctly - but we will not prove any formalities about them. We will generally assume, within reason, that used tools and libraries are correctly implemented and secure.

[11] Secret Conversations whitepaper: `https://fbnewsroomus.files.wordpress.com/2016/07/secret_conversations_whitepaper-1.pdf`

[12] WhatsApp E2E encryption: `https://www.whatsapp.com/faq/en/general/28030015`

[13] Viber security overview: `http://www.viber.com/en/security-overview`

1.3 Thesis structure

This thesis will continue with the following chapters and structure:

State of the art Explains the theory needed to support our solution. The relevant theory within steganography, cryptography and other areas will be examined.

Problem analysis We will break down the problem and examine possible solutions to each subproblems. We will consider pros and cons of different solutions, before ultimately deciding on the overall idea of StegoBlock.

Design Based on the theory, we will present a design chapter, detailing a very specific solution, in the form of a Thunderbird plugin.

Implementation Based on the designed solution, we will present an implementation. We detail the hurdles we overcame and how each component is designed.

Evaluation/Steganalysis The evaluation chapter examines our implemented solution like a typical steganalysis, and ultimately confirms our claims of confidentiality, integrity, plausible deniability and usability.

Conclusion Lastly we will conclude on our results and learnings.

1.4 Summary

Online communication is being wiretapped by major intelligence agencies like FSB, GCHQ and NSA. Specifically NSA program PRISM grants instant access to major internet communication companies databases. Furthermore we have also learned how the same agencies wiretap the very backbones of the Internet. Massive data centers allows for indexing these huge amounts of data. Analysts are capable of querying this data, for "suspicious stuff" - for example people using encryption.

We propose using steganography, to ideally avoid seeming suspicious. Steganography is by definition not encryption, and might be treated as non suspicious. At least for some time. Steganography may hide messages in such a way that adversaries are unable to tell if there is in fact a message.

We develop a steganographic solution, implemented as a Thunderbird email client extension. Users may enter secondary messages, that are encoded into

a generated block of text. The block adds a second subliminal channel, for secure messages without using encryption. We wish to make a solution that can withstand even these very strong adversaries, and work around possible legal regulation on encryption. We will also provide users with plausible deniability, so they can reasonably claim, that they did not write any secondary message.

Chapter 2

State of the art

2.1 Steganography

Steganography is the science of hiding information in such a way that it does not attract the attention of adversaries. For achieving confidentiality, steganography is the other major research topic, where encryption is the first. Because we initially ruled encryption out, due to possible regulations, we cannot use the latter for StegoBlock. Naturally, we turn to steganography instead.

2.1.1 History

The first mentions of what classifies as steganography, are two examples in ancient greek historian Herodotus's Histories[KP00]. The first about Histiaeus who tattoos a message on his trusted slave's shaved scalp. Histiaeus then waits for his hair to grow back, before sending him on his way to Aristagoras. When Aristagoras shaves his head again, the message is revealed. Another example is of Demeratus, who writes a secret message on a writing tablet, before applying wax onto it, which was common. The message was then only visibly after the wax was removed.

An ancient variety of steganography is watermarking, which is commonly known from bank notes, passports, tickets, postal stamps and other paper materials in need of counterfeiting resistance. Watermarking has its roots in the process of paper creation, where paper millers would embed their own watermark into their paper - assuring customers of the papers quality and origin. The first watermark known, dates back to 1292 to the town Fabriano in Italy[KP00], a time where competing paper mills needed to distinguish their brands from each other, because of varying quality, strength and format.

Steganography comes in other varieties, which we will explore further, (cf. §2.1.3). The examples mentioned, highlights that there has been a need for not only protecting, but also hiding information for thousands of years.

2.1.2 Today

Most fortunately, we can now send subliminal messages without tattooing slave scalps. In our modern information age, most messages are now digital and sent via the Internet. There is however still a need for conveying messages without raising suspicion, or to watermark or fingerprint material. Steganography is heavily used in the copyright protection business, where watermarking is used to protect against digital copies of material. For instance, watermarks can be embedded in CD's in such a way, that a normal computer program cannot simply copy the audio tracks and burn them onto a different CD. Similarly, computer games and applications can resist starting if an original CD or DVD is not present. Some communication protocols can also rely on embedding hidden information into images, thus allowing secret communication.

We have also seen people communicating by embedding secret information into seemingly innocent pictures, then uploading them to online image boards. The image is viewed by hundreds or thousands, but only a few persons inaugurated to the scheme, will know to look for a subliminal message. Detecting such messages can be extremely difficult, as images can be uploaded at so many different websites.

Another example is invisible ink. Ink that only shows under specific circumstances. Some ink may be invisible to the naked eye, unless viewed under ultraviolet light. Most people never expect letters in invisible ink, they may not even own an ultraviolet lamp - only the parties involved in the scheme will know.

Invisible ink has been used during several wars, as secret communication form. We have also seen microdots, where information was photographed and shrinked

to tiny dots on some cover letter. It could be regular letters or even newspapers. Regular readers would not notice, but other spies would be able to magnify and read the hidden message.

2.1.3 Principles and forms

In steganography, we usually wish to hide the message m. This involves a harmless message, called a cover-object c. The message is embedded into this cover-object, transforming it into the stego-object s. Sometimes, a stego-key k is used for embedding/extraction. Some algorithms do not require cover-objects, but instead generate cover-objects from the message or pre-analyzed text corpora. These types rely on Context-Free-Grammars and appear promising for hiding text within text. Peter Wayner[Way09] has written extensively on this topic, and an implementation named SpamMimic[1] is available. If adversaries, human og computer, are unable to distinguish any stego-object from a cover-object, the steganography scheme is secure. Although breaking a steganography system normally consists of three parts: Detecting, extracting, and disabling embedded information, a system is already insecure if an attacker is able to prove the existence of a secret message[KP00].

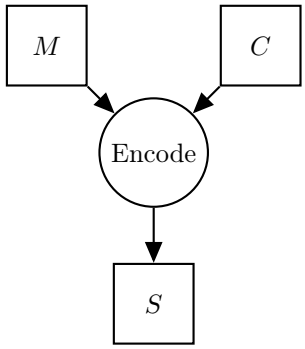
Embedding a message m into a cover-object c in a way that will prevent a third party to distinguish a set of cover-objects and stego-objects from each other, is non trivial. Not all data types are equally suitable. The transformation cannot alter the cover-object in a way that makes it appear "odd". The cover-object must contain a fair amount of redundancy, so that this can be replaced with a message, without altering the original perception og the object. Inspecting different algorithms has shown that noisy, redundant cover-objects, like images, are better suited for steganography, than very precise cover-objects like human readable text. These types are better, as they often offer higher bitrate, better disguise and more robustness.

The literature on steganography distinguishes between 3 forms: *Pure, Private key and Public key*[KP00]. Each with a set of advantages and disadvantages.

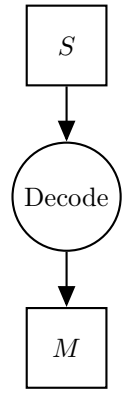
2.1.3.1 Pure steganography

The security of the system lies within the secrecy of the algorithm. Parties do not need to exchange any keys before use, only the algorithm itself must be known in advance. Tattooing messages on slave scalps is a form of pure

[1] SpamMimic: `http://www.spammimic.com`



(a) Encoding process, Pure steganography



(b) Decoding process, Pure steganography

steganography. Once adversaries learn this method, people would have their head shaved on the regular, just to make sure hidden messages are found. Pure steganography violates Kerckhoffs's principle, which we will return to shortly.

Formally, pure steganography can me expressed as an encoding and decoding function E and D:

$$E : C \times M \to S$$

$$D : S \to M$$

Where:
C is the set of all cover-objects, M is the set of all messages and S is the set of all stego-objects.

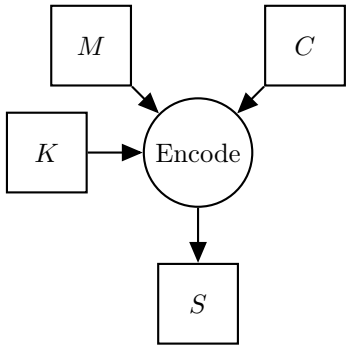
Under the condition:

$$|C| \geq |M|$$

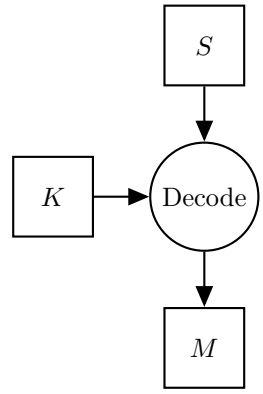
And with the property:

$$D(E(c, m)) = m \quad \forall \quad m \in M, c \in C$$

Informally, this means security is uphold by keeping E and D secret. There is no stego-key involved. The cover-object must be longer than the message, and that embedding m into c and extracting with D, will again reveal m - for all possible messages and cover-objects. By every message transformed with pure steganography, an intruder learns something about the plain text. There is usually no seed, nonce, alternating key or anything else to introduce entropy between messages.



(a) Encoding process,
Private key steganography



(b) Decoding process,
Private key steganography

2.1.3.2 Private key steganography

For pure steganography, we can assume that an attacker will learn D and E over time. We generally consider it unsafe. Private key steganography introduces the stego-key k. The objective is similar to symmetric cryptography. We will have both plain text and a stego-key as input parameters for our public function D. We may also inject a seed, which should introduce enough entropy in the output stego-object. If and only if, the receiving party knows k, he can reverse the process and extract the message.

Formally, private key steganography can be expressed as an encoding and decoding function E and D:

$$E_K : C \times M \times K \to S$$

$$D : S \times K \to M$$

Where:
C is the set of all cover-objects, M is the set of all messages, S is the set of all stego-objects and K is the set of all stego-keys.

Under the condition:

$$|C| \geq |M|$$

And with the property:

$$D_k(E_k(c, m, k), k) = m \quad \forall \quad m \in M, c \in C, k \in K$$

This obviously poses the problem of both sender and receiver knowing the key. In cryptography, we usually consider an alternative secure channel for transferring keys. This could be with a key exchange scheme, where the most well known might be the Diffie-Hellman key exchange protocol by Whitfield Diffie and Martin Hellman[DH76]. For private key steganography we will operate with the same assumption and leave key exchange outside the equation.

The StegoBlock application developed for this thesis will implement a private key steganography scheme. It depends on a stego-key for encoding and decoding.

2.1.3.3 Public key steganography

Similarly to asymmetric cryptography, public key steganography does not rely on a separate secure channel for key exchange. Instead, communicating parties are equipped with a key-pair: A private and a public key. The public key, used for encoding, is stored in a publicly accessible database. The private key, used for decoding, is kept private. In relation to Alice and Bob, Alice would use Bob's public key, to encode her message. The message would then only be decodable with Bob's private key.

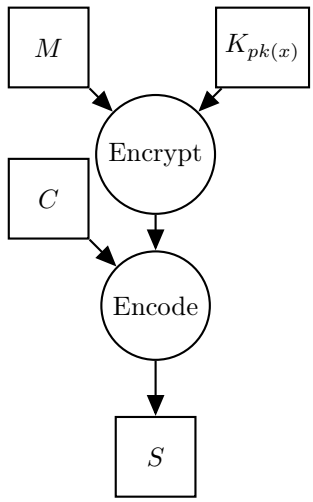
Typical public key steganography systems rely heavily on asymmetric crypto, according to Petitcolas and Katzenbeisser and a system has been proven by Ross Anderson[KP00, AA96]. The fact that any text can be encoded by a steganographic system, means that it could be cipher text from any asymmetric encryption scheme as well as plain text. Because the encoding function accepts any message and cover-object from the sets of all M and C, this could of course also be the output of a crypto system. All cover-objects in C are promised indistinguishable from each other, whether or not they include a secret message. If Wendy suspects a message to contain a hidden message and analyzes it, she will arrive at a cipher text, indistinguishable from what she would arrive at, whether or not the message contained a message.

Formally, public key steganography can me expressed as an encoding and decoding function E and D:

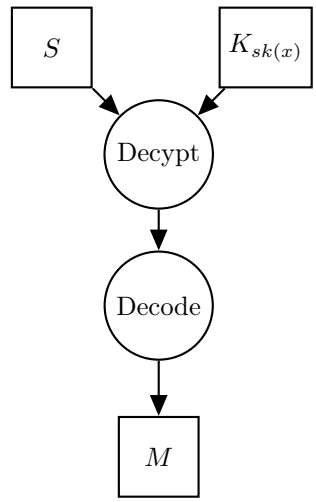
$$E_K : C \times M \times K_{pk(x)} \rightarrow S$$

$$D : S \times K_{sk(x)} \rightarrow M$$

Where:
C is the set of all cover-objects, M is the set of all messages, S is the set of all stego-objects and K is the set of all stego-keys.



(a) Encoding process, Public key steganography



(b) Decoding process, Public key steganography

Under the condition:

$$|C| \geq |M|$$

And with the property:

$$D_{sk(x)}(E_{pk(x)}(c, m, pk(x)), sk(x)) = m \quad \forall \quad m \in M, c \in C, sk \in SK, pk \in PK$$

Because the encrypt/encode and decrypt/decode functions are independent, and the key is used for the crypto scheme, one can refer to the scheme as pure steganography - even though it does not in practice conform to the previous definition.

2.1.3.4 Kerckhoffs's principles

In 1883, Auguste Kerckhoffs formulated in the January and February issues of *Journal des sciences militaires*, 6 principles that any military grade cryptographic system should uphold. Especially one of these principles, are widely recognized amongst cryptographers. It has great applicability to steganography as well. His second principle is: *The system must not require secrecy and can be stolen by the enemy without causing trouble*[Ker83]. This boils down to, the crypto scheme must be able to sustain public knowledge. Only the key, which is chosen individually, is to be kept secret. Systems failing to comply with this

principle, are often referred to as "Security by obscurity", and are generally regarded as bad, for instance by NIST[oST08].

If security exists only by keeping the cryptographic mechanism secret, like a black box, the system will be completely broken when an adversary learns the inner workings and discovers a flaw. Cryptography is supposed to keep a message secret, by replacing it with another secret that is much easier to hold. Modern crypto systems are very complex, and keeping them secret would then only replace one secret with another large one.

If security is not bound to secrecy of the key, the system can only be used between trusted parties, and any object encrypted by the system, in the past or future, will be vulnerable when an adversary finds a flaw. Keeping a system secret does however not imply that it is flawed, but it will not be scrutinized by public experts. Some military grade cryptography algorithms are kept secret as an extra layer of protection, for instance NSA Type 1 cryptographic products[2].

2.1.4 Steganalysis

In short, steganalysis concerns itself with the ability to detect if a cover-object contains a stego-object or not, but also what it would require to destroy any stego-object, without also destroying the perceptibility of the cover-object. In order to discuss steganalysis, we will first establish a representation of all steganographic techniques. We can represent those as $C = p + t$, where C is potential for a carrier of hiding information within it. p is the portion of the carrier which will produce perceptible differences if manipulated. t is opposite, the portion of the carrier that we can manipulate without producing perceptible differences[KP00]. This expression allows us to describe a kind of attack, where an attacker may destroy the stego-object that may be embedded in some cover-object. As t describes some range of the cover-object within the imperceptible range, there exists some t', so the attacker can create $C' = p + t'$. This will be perceived as C, since p - the perceptible portion was left intact. By changing the imperceptible region, an attacker may destroy a stego-object - without even knowing if there in fact was a stego-object. The robustness of a steganographic system is its ability to withstand such attacks, and thus steganalysis is a critical exercise to perform - both for security and durability reasons.

We can also consider embedding messages, or part of messages in perceptible regions, with the increased risk of being detected, especially by humans - but

[2]NSA Type 1 products: `https://en.wikipedia.org/wiki/NSA_product_types#Type_1_product`

also with added robustness, as perceptible regions are not easily replaceable. Attacks and analysis of steganography includes detecting, extracting, counterfeiting (embedding fake information over the existing hidden information), and destroying messages. StegoBlock does not try to guard against all these attacks. In particular, it is not very robust. But it does protect against extracting and counterfeiting.

Steganalysis resembles cryptanalysis, but techniques vary. In cryptanalysis, we consider attacks of known-plaintext, chosen-plaintext and ciphertext-only. Each with increasing difficulty. We may analyze each of these three ways to observe an algorithm. In steganalysis, we similarly have [KP00]:

- **Stego only attack** Only the stego-object is available for analysis. We can also refer to this setting as "blind", because of the very limited information level.
- **Known cover attack** The cover-object and corresponding stego-object are both available. The analysts may look for linkage between them.
- **Known message attack** At some point, the hidden message may become known to the attacker. Analyzing the stego-object for patterns that correspond to the hidden message may be beneficial for future attacks against that system. Even with the message, this may still be very difficult.
- **Chosen stego attack** Both the steganography algorithm and stego-object are known.
- **Chosen message attack** The steganalyst generates a stego-object from some steganography tool or algorithm from a chosen message. The goal in this attack is again to determine corresponding patterns in the stego-object that may point to the use of specific steganography tools or algorithms.
- **Known stego attack** The steganography algorithm is known and both the original message and stego-object is available.

There are incredibly many ways of hiding digital information. Detecting them may be harder for some than others. The strength of steganography lies in hiding. Remember that *a system is already insecure if an attacker is able to prove the existence of a secret message.* Since embedding techniques can be so different, it is not possible to pinpoint a single detection method. A steganalyst can however use some basic techniques that will take him far. In general it pays off to look for unusual patterns (which the human brain is very good at), to look for anomalies or to examine redundant or invalid data. Let us consider basic

examples and ways to examine cover-objects, to establish how these techniques can be of help.

Earliest stegosystems were physical, meaning the stego-objects were "real". It could be slave scalps, but also seemingly ordinary books or letters. A stegosystem with real letters, printed on paper, may vaguely shift letters or lines in different directions, to convey a secret message. For instance, one could encode a secret message in a newspaper, by slightly tilting or shift individual letters. The intended recipient will know to look for these letters, and by extracting only these, a subliminal message is revealed. The casual observer may not notice this, or know what it means - but the steganalyst will know to look for these unusual patterns and proceed to examine their meaning. In much the same way, a digital letter may look completely normal when shown compiled on screen, but has a secret message embedded in its markup. This could be by added invisible characters, elements or attributes - something that will not be rendered. The steganalyst should examine the source and look for patterns in these invisible elements. If they really do not have any effect on the rendering, it may be that a secret message is embedded.

Consider also a parties communicating over a network, setting TCP packet headers to invalid values. Such values would normally be discarded, except if the party knows they contain a special hidden information. This resembles the Chaffing and Winnowing technique (cf. §2.2), where chaff packets are automatically filtered by the intended recipient. A steganalyst will have to look for such invalid data, and assess if they might hold secret meaning or if they are simply the results of misconfigured clients.

There truly are vast ways and places to digitally hide information. An adversary will first have to expect the presence of a stego-object before spending time on information extracting. If he does not detect this, he may very well already have lost. Contrary to cryptography, it does not matter how much processing power the adversary has - he must expect its presence first. Should the embedded message also be encrypted before embedding - he may not even have a way of confirming that a message was indeed embedded - before breaking the encryption.

Processes for analyzing different stego-object types and different embedding types are widely different. There is no silver bullet, that tells if an object contains hidden information or not. Techniques depend on object type. Techniques for analyzing text is different from analyzing audio. When performing steganalysis, it is also common to encounter false positives and negatives. An analysis may falsely indicate that some object contains information, while it does not. An attacker may waste time and resources on extracting attempts. An analysis may also not indicate any hidden information, because the embed-

ding scheme is too sophisticated. We cannot offer a single solution for all kinds of data, but we can look at some examples and learn from the techniques, so we can adjust them for other areas.

2.1.4.1 Image analysis

Images are nice cover-objects for steganography. Their potential for hiding information, C is usually excellent. An image typically has large amounts of redundancy and areas where manipulation has low perceptibility. A very common way of hiding information in images, is by manipulating the least significant bits (LSB). Essentially the human visual system and its inability to distinguish small anomalies in a large image, is exploited. By replacing the least significant bit of every element, we can easily encode a secret message in an image. Consider the example[3] of hiding the letter "A" in a 24 bit image. "A" has the binary value `1000001` and a length of 7. We would need 7 elements to encode our value. As mentioned before, the cover-object must be larger than our message, thus we will need an image of at least 3 pixels, since each pixel is made up of 3 values, expressing either Red, Green or Blue. Consider a random image of 3 pixels, which in binary may be expressed as:

10000000.10100100.10110101, 10110101.11110011.10110111, 11100111.10110011.00110011

We can encode our "A" into this binary sequence by replacing each LSB:

10000001.10100100.10110100, 10110100.11110010.10110110, 11100110.10110011.00110011

Bits underlined, highlight a replacement of the existing value. Since a bit can either be 1 or 0, the average replacement ratio should be 50%. Changes to images by an LSB algorithm will largely go unnoticed by humans, especially if the image contains many details. Images with monotone backgrounds or gradients will be bad - as a human would quickly notice the anomalies. Remember that there is no exact method for a steganalyst to decide the steganographic algorithm used, if any. He would have to suspect something embedded. In the example of LSB in images, it would however be trivial to perform a statistical analysis of the bits in the image. Any even-valued bit will either keep its value or be incremented. It cannot be decremented. The opposite is true for odd-valued bits. This fact creates an asymmetry which is easily detected by techniques devised by Dabeer et. al. [DSM+04]. This statistical anomaly can be overcome with a more sophisticated LSB technique, named LSB-matching or ± 1-embedding. We will adjust for the statistical anomaly, by increasing/decreasing other parts of each

[3] Example from: `http://www.lia.deis.unibo.it/Courses/RetiDiCalcolatori/Progetti98/Fortini/lsb.html`

pixel accordingly. This technique does however also generate anomalies, but significantly harder to detect. Cancelli et. al. provides one method, among others, for this [CDJCB05]. A targeted steganalysis approach to examine the image example, would be to first assume LSB embedding, or another common algorithm, and to verify it. A blind approach would be to make statistical analysis of the stego-object, without assuming any specific embedding algorithm. Statistical analysis is a very powerful tool for detecting steganography.

2.2 Chaffing and winnowing

Noticeable previous work has been made, to allow confidential communication without encryption. This work is particularly interesting, as it was made to highlight the flawed logic in banning strong encryption. In the late 90's we also witnessed strong pressure against encryption. USA placed an export ban on strong cryptographic schemes, but history has shown that it was ineffective. As the Internet became ubiquitous, strong encryptions schemes did as well, it was impossible to regulate.

"Chaffing and winnowing", by Ronald Rivest attracted a lot of academic attention in 1998[Riv98]. It is presented as an alternative to both encryption and steganography. The idea uses the analogy of separating chaff from grain, a process known as winnowing. StegoBlock is very similar to "Chaffing and winnowing". To quote Rivest:

As usual, the policy debate about regulating technology ends up being obsoleted by technological innovations.

This is what we are trying to achieve again with StegoBlock, to be pedantic in the debate about encryption. Rivest utilizes an authentication method to achieve confidentiality. The concept is very straight forward: All packets are authenticated by appending a MAC of the packet. A transformation in the form of: $packet \rightarrow packet, MAC$. Notice that the original packet was not transformed, it is in the clear. We remember that any message transmitted on a network may be transmitted in one or more packets. Let us relate this to Alice and Bob. Before Alice sends her message to Bob, she or her network adaptor, will break it into one or several packets. Each packet is authenticated by some secret key she shares with Bob. Rivest proposes a key exchange protocol, like Diffie-Hellman, for Alice and Bob to agree on a shared secret. Packets are now authenticated by some secret, known only by Alice and Bob. Bob will now recompute the MAC and drop any packet with a mismatching MAC. This is already done by network adaptors today, for instance on wireless networks,

where every client receives all traffic, but only accepts traffic intended for that client. This is the *winnowing* process.

What is left now, is simply to add *chaff*. Rivest first proposes that Alice could send out one or more bogus packets for each "real" packet. A bogus packet with an invalid MAC. Bob would automatically filter those away, they would have no influence on their communication. Packets will consist of a sequence number, and packet contents. The MAC function would be defined as: $MAC(sequenceNo, packetContents, key)$. Examples of packets could then be:

```
(1,Hi Bob,465231)
(2,Meet me at,782290)
(3,7PM,344287)
(4,Love-Alice,312265)
```

If Alice added chaff, it could be exemplified by:

```
(1,Hi Larry,532105)
(1,Hi Bob,465231)
(2,Meet me at,782290)
(2,I'll call you at,793122)
(3,6PM,891231)
(3,7PM,344287)
(4,Yours-Susan,553419)
(4,Love-Alice,312265)
```

It is easily seen how an adversary will be unable to make the decision which packet in the sequence, is the correct. In the scheme presented until now, creating a chaff packet is difficult. It would have to express some meaning. Chaff packets without meaning would be easily distinguishable from real packets. For instance: `(1,Hi Larry,532105)` would be easily distinguishable from `(1,fjSJswer,196845)`. To remedy this situation, Rivest first proposes to only transmit a single bit in every packet. The chaff packet should then consist of the opposite bit. This scheme is computationally hard to break and easily implementable. Transferring packets of single bits, does however have unnecessarily much overhead. Network traffic speed would become drastically decreased. There are however non-encryption algorithms for transforming packet contents into what appears as random characters. Rivest proposes his own "all-or-nothing" and "package transform" algorithms. This would allow for easy chaffing of larger packets. We can now amuse ourselves with how we achieved

confidentiality without encryption. We utilized the existing network adaptor process of filtering away unintended packets and generating chaff-packets. Rivest goes on to emphasize how the chaffing process is independent of knowing the secret, and how it can be distributed. Alice could even knowingly or unknowingly have Charles sit between her and Bob, generating chaff. She could securely delegate the chaffing process to some third party.

In somewhat the same way, our extension StegoBlock will accept some message we wish to keep confidential. Then add enough "chaff" around and in between the letters it consists of, to simply distort the message without altering any character. The final block contains the message and a whole bunch of chaff. We add chaff with a PRNG, to which the seed is kept secret between the sender and recipient. The recipient can easily "winnow" the StegoBlock, by initializing a PRNG with the same seed, bringing it into the same initial state and remove chaff. Only by knowing the secret, one can separate the chaff from the grain (message). We will elaborate much more on our scheme throughout the thesis.

2.3 Cryptography

We employ several cryptographic primitives in StegoBlock, but none of which are classified as encryption. In particular, we will use random number generators and hash functions - each belonging to the cryptographic toolbox. These tools are used in encryption schemes, but are not encryption themselves.

2.3.1 Randomizing algorithms and RNG's

Being able to generate random numbers is critical. Computers today are deterministic. This is usually highly desirable, as we obviously enjoy the certainty of arriving at the same result, every time we perform the same calculation. Much effort has gone into eliminating randomness in computers. But every so often, our applications need randomization for some reason or another. Unfortunately, it is then impossible to generate something truly indeterministic on a deterministic machine, at least without some indeterministic input. While computers can process extremely complex calculations, for instance that $2^{74,207,281}$ is a prime number[4], they are also bad at flipping coins. The best they can offer are Pseudo Random Number Generators (PRNG), that generate what we could perceive as random, provided with some seed. If the seed is random, the output will be

[4]Largest known prime number: `https://en.wikipedia.org/wiki/Largest_known_prime_number`

random - or one could say that if the input is random, so is the output. This is beneficial, because the PRNG algorithm adapts the inputted randomness to some criteria. The PRNG translates the seed to a sequence of numbers that can be perceived as random and outputs a pattern. Since it's the same pattern followed on each run (with same seed), the numbers outputted are only what we define as *pseudo random*.

Pseudo random numbers are good enough for most applications. If one needs a random sample of words in a dictionary, then a pseudo random number might be just fine. If one needs a random color for painting a computer desktop - a pseudo random number will be just fine.

Other applications crucially need random numbers. For instance the setup of a Secure Socket Layer connection in browsers. Early versions of the Netscape browser used a PRNG for generating random numbers, needed for SSL initialization. They seeded their PRNG with the concatenation of 3 values: Time of day, the process ID, and the parent process ID[5]. For a sophisticated attacker, these values are quite predictable, and thus the outputs were not random. Even though the precise values could not be derived, they could be approximated. This would lower the range of values to try considerably, and brute forcing the correct value would become computationally feasible:

"Optimizations such as those described should allow even a remote attacker to break Netscape's encryption in a matter of minutes."[daw96]

Choosing a seed with enough entropy, if the application requires it, is crucial for PRNG's. Algorithms running in environments with I/O access, may calculate random numbers seeded by thermal or atmospheric noise. This raises complexity, and even requires dedicated hardware at demanding times.

Any PRNG conforming to requirements of cryptography, can be classified as a Cryptographically Secure Pseudo Random Number Generator (CSPRNG). The requirements are:

Next-bit test Given the first k bits of a random sequence, the $k+1$'th bit cannot be predicted by a polynomial-time running algorithm, with probability of success better than 50% [Yao82].

State compromise extensions Should the PRNG's state, or part of it, be revealed - it must be impossible to reconstruct any output previous to

[5]Random number generator attack: `https://en.wikipedia.org/wiki/Random_number_generator_attack#Predictable_Netscape_seed`

that state[6].

StegoBlock has to permute an array of characters, in such a way that it can be reversed only if one knows the key - and such that the permutation is one of every possible permutations. Technically, we need a cryptographically secure pseudo random number generator, for a randomizing algorithm.

Consider the string $s =$ "*hello*". It has 5! (factorial) different permutations. A shuffling algorithm executed with input s, must be able to return all 5! permutations with equal probability, before we can begin considering it secure. Obviously such an algorithm has a need for randomness. Let us examine shuffling algorithms, also known as randomizing algorithms.

2.3.1.1 Shuffling

One of the most commonly known shuffling algorithms, proved to output one of all possible permutations, is the Fisher-Yates [FIS53] or Knuth Shuffle[Knu68]. The algorithm is very simple, first described by Fisher and Yates - later translated to a computer algorithm by Donald Knuth. It can be seen in Algorithm 2.1[7]. The seen implementation has complexity $O(n)$.

```
1  input: string[] n
2  begin
3    for i from n−1 downto 1 do
4      j ← random integer such that 0 ≤ j ≤ i
5      exchange n[j] and n[i]
6  end
```

Algorithm 2.1: The Knuth Shuffle.

In words, it will iterate the entire string, switching the current character and a random previous one (starting from the end). The string will be permuted and all possible permutations may be returned. We will not examine the proof, but in previously referenced materials, this has already been shown.

But one thing is the algorithm being solid on paper. There are several potential sources of bias to look out for when implementing. Let us examine some, as this will be very valuable in our later analysis of StegoBlock.

[6] CSPRNG requirements: `https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator#Requirements`

[7] Wikipedia, Knuth Shuffle (Algorithm P): `https://en.wikipedia.org/wiki/FisherYates_shuffle#The_modern_algorithm`

Bad implementation It may seem obvious that an algorithm must be implemented correctly, however a developer can make subtle mistakes in even very simple algorithms. It may look as if results are correct(especially when generating random strings) - when in fact they are not. Notice in Algorithm 2.1 how j is a number drawn from the pool of remaining characters. One could make the mistake of drawing from all characters, introducing a bias[Atw07]. When examining this particular mistake, it may look as if we shuffle more than we should, which intuitively should be better for randomness, but slightly hurt performance. In reality, shuffling more, hurts randomness badly. Notice that the wrong implementation has n^n possible permutations (we iterate over n, and swap any of n). A correct Knuth Shuffle would have $length(n)!$ possible permutations.

Consider running such a faulty algorithm on the array $n = [1, 2, 3]$. It would have 27 possible combinations, while it's a mathematical fact that there should only be 6 possible combinations. Since 27 is not evenly divisible by 6, we cannot even assume that the extra outcomes are evenly distributed - there must be some bias. Paying extreme attention to detail when implementing is critical.

Scaling down state When our shuffle algorithm needs a random number for swapping characters, it needs a PRNG to do so. As we detailed earlier, this is a non trivial operation. Most PRNG's are implemented in such a way that they provide random numbers in some fixed range. Internally, they may have random numbers in the range of 0 to $2^{32} - 1$ or another fixed range. Applications may need random numbers in many other different ranges. Let us again consider our shuffle algorithm. In some setting, it might require some random number in the interval $0 - 15$. It will query some PRNG with an internal range of $0 - 99$. A common way of fitting the result to the requested range, is to apply modulo length of requested range. This can produce a very subtle bias, if the internal state range is not divisible by the requested range. In this particular setting, we will see numbers $0 - 3$ occurring 17% more frequent than $4 - 15$, simply because 16 does not divide evenly with 100[wik].

One possible solution is to use a PRNG with a dynamic internal range, based on the request. Another, much more simple, is to not apply the modulo function. If a number is drawn outside the desired range, request another. Keep doing this until a valid number is returned - even though this could potentially run forever.

Some PRNG's have an internal range of $0 - 1$, but instead return floating points. It is then common to multiply the result by the requested range, and round. Again, this can introduce a bias, because of the finite precision of floating points.

The range of values producible would also be finite. If the number of values in the requested range, does not divide evenly by the floating point - we would again see a bias, similar to the one of PRNG's applying modulo.

Scaling up state A third type of bias, also related to the inner workings of PRNG's occur if it has too few distinct states. If it provide numbers in a range shorter of the requested. A RNG can never be used to securely permute any object into more distinct states, than it has internal states for. Consider again the example RNG that is seeded with 32 bits, e.g. can generate random numbers in the range of 0 to $2^{32} - 1$. This will be enough to exceed the possible permutations of 13 card deck, as $13! < 2^{32} - 1$. A deck of 14 cards will however have more permutations. Typically the RNG will "wrap around" and reuse entropy, resulting in a bias.

2.3.1.2 Well known CSPRNG's

It is impossible to prove that a sequence of numbers are truly random, but possible outcomes should appear with equal probability. NIST provides a series of tests to perform against a RNG, which is a good starting point[AR10]. As a rule of thumb in computer security, one should use existing tested primitives, instead of inventing/implementing their own.

A random number generator is either secure or not. If considered secure, one does not have to worry about the internal workings and potential bias we just iterated. A well known cryptographically secure PRNG is the Blum-Blum-Shub (BBS) PRNG[BS86]. BBS is a stream cipher and given some short input, it will generate a potentially infinite stream of pseudorandom output. BBS comes with a security proof, however now criticized for its impractical performance limitations[SS05].

Another approach is to use AES-CTR, AES in Counter mode. Similar to BSS, it may provide a potentially unlimited number of random bits, as it is a stream cipher. However according to NIST, one will have to reseed after 2^{32} outputted bits, to ensure an adversary has a low advantage of predicting the output. It is like a sponge, we may soak it, squeeze out some random bits, but eventually run dry.

2.3.2 Integrity

When sending messages securely, we will also need to ensure message integrity. Without integrity checks, an adversary may unknowingly, to the recipient, alter a message. The adversary may not be aware what he changes the message to, but his goal may simply be to obfuscate communication. One may falsely believe that a message unreadable by adversaries, for example if encrypted, is also safeguarded against tampering. Consider the theoretically perfect secure one-time-pad scheme[8]. Should an adversary change any bit of the cipher text, some plaintext bit will also change. Without integrity checks, the recipient will have no means of telling.

It is fairly simple to ensure integrity of a message. In fact, we already touched on this subject in Chaffing and Winnowing. Here we described how a MAC function was used to authorize messages, allowing only the recipient to successfully decide which messages are chaff and which are winnow. We calculated a hash of our message, only the intended recipient could recalculate that hash.

A hashed MAC function is a cryptographic primitive that accepts a message and a secret key as input. It is a one-way function, it returns a digest, a fixed size output often much shorter than the input. The digest will change according to the input, even a small input change will affect the digest greatly. See Figure 2.4 where we MD5-hash the values *StegoBlock* and *stegoblock*. Each input results in different digests, but of equal length.

Given hash functions one-way nature, it is not possible to reverse the process and calculate the original message from a digest. It is however possible to have hash collisions, the scenario where two or more different inputs generate the same digest. This is generally bad, as we can then no longer rely on the digest coming from the claimed message. There is then no way to tell if a certain message was altered or not. Depending on the strength of hash function, this is more or less common. An example of a broken or insecure hash function could be MD5, where even a normal household PC may calculate a collision. First examples of limited collisions were reported in 1993 By Den Boer and Bosselaers[dBB93] and in 2005 it was shown how to generate collisions within minutes on a standard notebook by Vastimil Klima[Kli06]. NIST currently (since August 5, 2015) recommends using a minimum of SHA-256 hashing function for any usages[NIS15].

When hashing inputs very similar, for instance *Hello* and *hello*, digests should also be very different. This is known as the *avalanche effect*, and is a result of good randomization. Without the avalanche effect, cryptanalysis would enable predictions of future digests. The Strict Avalanche Criterion states that if one

[8]One Time Pad: `https://en.wikipedia.org/wiki/One-time_pad`

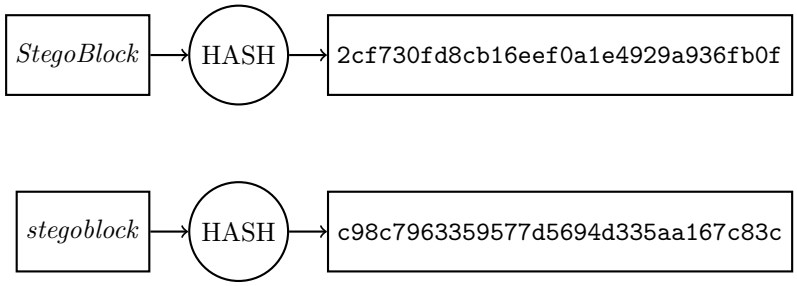


Figure 2.4: Example hash function, MD5

bit changes, every digest bit must change with a 50% probability. The avalanche effect is important in hash functions, but we would also like this trait in our stego block.

2.3.3 Men in the middle

With cryptography we may securely encrypt messages with encryption schemes. We can also ensure their integrity after sending them over an insecure network, by hash functions. But unfortunately, we can still not be safe against sophisticated adversaries. Let us consider an increasingly popular attack vector: The man in the middle. When two parties communicate, they may use strong encryption to ensure privacy. Without proper authorization, for all they know, they could be having a private conversation with the devil.

Let us consider the key exchange protocol by NeedhamSchroeder[NS78]. This is a quite old and tested protocol, ensuring that two parties may arrive at the same secret, over an untrusted network. The revised protocol is part of popular authentication protocol Kerberos, and practically used for devising a shared secret used for either asymmetric or symmetric encryption.

The protocol was initially executed as in Listing 2.2. We will use it as an example of a Man in the middle (MITM) attack. It uses cryptographic primitives like, nonces and public keys. We will not explain these further than the following: A Nonce is a value(integer) that is generated and may only used once. A public key is the publicly available part of a public/private key pair, we already touched on this matter in our section on Public Key Steganography. A public key is not to be kept secret, as opposed to its private key counterpart.

```
1  A → B : {N_A, A}_pk(B)
2  B → A: {N_A, N_B}_pk(A)
```

3 A → B : $\{N_B\}_{pk(B)}$

Listing 2.2: Needham-Schroeder PKE - 1978 version

In Step 1 of the Needham-Schroeder PKE protocol, we have Alice generating and sending a fresh nonce to Bob. Before sending, she encrypts her message with Bob's public key. The Nonce ensures message-freshness (avoids replay-attacks), and encrypting ensures that only Bob can decrypt the message with his private key. In Step 2, Bob replies with the same nonce received from Alice, but also with one he generates himself. His response is encrypted with Alice's public key. Lastly in Step 3, Alice replies Bob with his nonce, also encrypted with his public key.

This dance of nonces ensures to each party, that the other party knows the corresponding private key. How else would the counterpart be able to reply the nonce? After Step 2, Alice must of course compare N_A received from Bob, to N_A she sent. In case of a mismatch, the handshake is terminated. Bob does the same after Step 3. If the handshake completes, they may talk privately.

But consider now the case where an adversary sits in the middle of Alice and Bob. Alice does not realize Eve is malicious, and Eve may use then information obtained to start a conversation with Bob. Bob thinks he speaks to Alice, but unknowingly speaks to Eve as well.

1 A → E : $\{N_A, A\}_{pk(E)}$
2 E → B : $\{N_A, A\}_{pk(b)}$
3 B → E : $\{N_A, N_B\}_{pk(A)}$
4 E → A : $\{N_A, N_B\}pk(A)$
5 E → B : $\{N_B\}_{pk(E)}$
6 E → B : $\{N_B\}_{pk(B)}$

Listing 2.3: Needham-Schroeder PKE - MITM attack

We see that Eve sits between Alice and Bob and exploits the fact that the protocol wrongfully assumes that whoever created N_A is also A, in Step 4. Because it is not checked, Eve exploits A to start a private conversation with Bob, posing as Alice. The MITM attack is a result of not realizing that both parties may be assumed by the adversary. This specific attack on the Needham-Schroeder protocol was discovered and fixed by Lowe[Low95]. The fix is simply to include authorization in Bobs reply in Step 3, which Alice must then verify.

As we are also sending messages over insecure networks with StegoBlock, we must keep this attack form in mind, and protect accordingly. In a similar way, we must include some form of authorization of our emails, to ensure they are in fact from who they claim. Observe Listing 2.4 where we see a similar attack on

the email protocol. Notice that this attack is only possible if Alice trusts Eve, and if Bob believes that Eve is in fact Alice, and that each pair have exchanged stego-keys with each other. Eve must be present in the key exchange. We see how Eve in Step 2 relays an email from Alice, exchanging message S with S' to Bob, where she claims to be Alice.

In the email protocol, the sender is expressed in the message header. Anyone sending an email may claim to be anyone. For a standard email, anyone may set or change the sender header - the recipient can thus never trust this field. Because the attack is so hard to carry out, StegoBlock offers no special remedy, but we will consider the underlying sender-spoofing problem with basic emails. Solving this, would also solve the attack on StegoBlock.

1 $\texttt{A} \rightarrow \texttt{E} : \{S, A\}_{sk(AE)}$
2 $\texttt{E} \rightarrow \texttt{B} : \{S', A\}_{sk(EB)}$

Listing 2.4: Email with StegoBlock

2.3.3.1 Sender Policy Framework

Luckily, email spoofing is an already solved problem. The protocols are available, at least. The Sender Policy Framework (SPF) provides necessary tools. RFC7208 describes the details, and has been experimentally published since 2006[rfc14]. SPF enabled email recipients to verify, if a particular email is indeed from the domain it claims to be. If the SPF check passes, the recipient will know that mail from *alice@doe.com* was in fact sent from doe.com. He will then have to trust doe.com that it was also from Alice, and not some other user at doe.com.

SPF requires setup at the Domain Name System of the domain, only the rightful owner should have access to DNS records of any domain. In a TXT record, the owner may grant access to certain computers, authorized to send email on its behalf. An example could be `toftegaard.it. IN TXT "v=spf1 ip4:2.104.2.59 a -all"`. A receiving server can then verify if the delivering server of an email from toftegaard.it, was in fact the one with IP `2.104.2.59`.

Notice that SPF can only verify the sending server, not the user. Should Alice try to send an email from bob@doe.com, SPF would not catch it. This check is, reasonably, left to the doe.com server.

2.4 Summary

Steganography is a very old art form, dating back to ancient Greece history. Today it is used heavily in its derived watermarking form, in the digital copyright industry.

In steganography, we will usually seek to hide the message m, in a cover-object c, resulting in a stego-object s. We may also have a stego-key k, and some schemes require no cover-object at all. The good steganographic transformation will make an adversary, human or computer, unable to distinguish a set of cover-object from stego-objects. Some cover-objects are better suited for embedding than others, based on how much redundancy they contain. Images typically hold more redundant data than text. The human visual system is quite forgiving, it might not notice small changes to the big picture.

Steganography, similarly to encryption, comes in different forms: Pure, private- and public key. In its pure form, security is in the secrecy of the embed and decode algorithms. No stego-keys are used, allowing easier initialization, but is fair to assume that adversaries over time eventually will learn the algorithms. Pure steganography typically also use no seed or nonce, meaning that there is nothing to introduce entropy between messages. Once adversaries learn the algorithms, they can decode all past and future stego-objects.

We may turn to private key steganography for remedy. In this form, a stego-key introduced. To initialize a protocol, parties must know the algorithms but also a stego-key, in advance. The key essentially allows parties to convert a large secret, the message, into a smaller and more handy secret, they key. Combined with a seed, it introduces entropy between messages. Should an adversary learn one message, he may not be able to learn others. To initialize a private key steganography protocol, stego-keys must be exchanged and we may do so in any way we see fit. For instance with a face-to-face meeting, or digitally with a key exchange protocol.

To avoid key exchange protocols or meetings, we can also use public key steganography - which in the described versions are merely combinations of pure steganography and asymmetric encryption. The example is to encrypt some message, with an asymmetric encryption scheme to arrive at some cipher text. An embedding function may accept any kind of input, so we simply feed it the cipher text.

We learned that private- and public key steganography both comply with Kerckhoffs's principle of only keeping the key a secret. Should the key to some message be stolen, it will not cause trouble for other messages.

Analyzing the security of steganographic schemes is a job for steganalysis. As there are so many different way to hide information digitally, we cannot present a silver bullet for analyzing stego-objects. A steganographic system is already insecure if an attacker is able to prove the existence of a secret message, it is in the nature of steganography to not be detectable. When analyzing objects, one generally look for unusual patterns, anomalies, redundant or invalid data.

We looked at very few cryptographic primitives, simply because we do not need that many for our StegoBlock system. Pseudo random number generators will form our security backbone. PRNG's translate an input seed to a sequence of numbers, that can be perceived as random. The seed deterministically brings the generator to some internal state. This is a crucial property for StegoBlock, as this allows for future decoding, if one knows the seed. The Knuth-Fisher-Yates shuffle algorithm was examined, along with a CSPRNG it becomes a very nice fit for our shuffling needs. It guarantees that its permutations belong to the set of all permutations with equal probability.

Integrity checks of messages ensure that they were not altered since creation. Hash functions allow these kind of checks, by transforming messages of arbitrary length to fixed length (often shorter) digests. Some hash functions are better than others, measured by the difficulty of producing collisions. As of writing, algorithms of at least SHA-256 are recommended for any use.

Even though we have secured a message and its integrity, we learned that a clever adversary, working as a man in the middle, may still cause trouble. In short, if parties blindly trust each other, a malicious client may abuse communication with one client to fool another.

Lastly, we visited the lesser known alternative to steganography and encryption, chaffing and winnowing. None the less of much interest. It was devised in late 90's, were strong encryption scheme export was banned in USA. It was shown how simple authentication, could be used to achieve the same goals as strong encryption, and thus how technologic oppression will be obsoleted by new technology.

Chapter 3

Problem analysis

Ultimately we will handle the problem of allowing people to communicate securely. Securely refers to the condition where no one but the intended recipients may read and understand their messages. The terms private or confidential are synonyms in this topic. To achieve privacy, we need a privacy- and transmission solution as illustrated in Figure 3.1. We have a sender and a receiver. The sender makes his message private with some tool before transferring it. It is then transferred, by some third party - as the sender and receiver are not in proximity. The message will then arrive at the recipient and only he is able to extract the message before reading it.

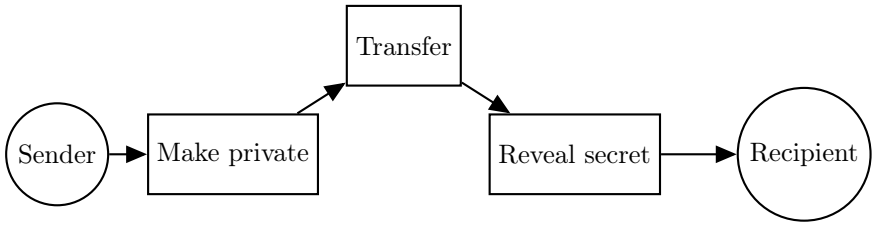


Figure 3.1: Communication process

3.1 Confidentiality

From the theory we just assessed, we learned of 3 possible research areas that may provide secrecy: Encryption, chaffing and winnowing and steganography. Use Figure 3.2 to assist the following walkthrough.

Encryption is the classical choice for privacy or secrecy. The area is extensively researched. Encryption promises to keep messages private, like a digital treasure chest with padlock, for messages instead of jewels and gold. It comes in two forms: Symmetric and asymmetric. In the symmetric form, we operate with a single cryptographic key. As with the padlock on the treasure chest, we must use the same key to lock and unlock. The key is a shared secret between all authorized parties. They must know the key in advance, only with the right key, the message can be decrypted. Asymmetric encryption instead operates with cryptographic key pairs. Keys are mathematically bound to each other, in such a way that one may encrypt and the other decrypt. This is highly convenient, as parties do not need to transfer a secret key, as with symmetric encryption.

For symmetric encryption we may consider options like AES, Blowfish, RC4 and many others. All are good algorithms for encrypting a message and keeping it secure. Popular asymmetric encryption schemes are RSA and Diffie-Hellman Key Exchange, amongst others.

Encryption is, as mentioned, the most used technology for keeping digital message private. We already mentioned solutions like HTTPS Everywhere and Let's Encrypt, that advocates for encrypting all web pages by default. We have also seen how instant messaging platforms move in the same direction, by offering end-to-end encrypted conversations. All these examples use some form of encryption.

To verify or break an encryption scheme, we have cryptanalysis. It is the process of studying or analyzing some cryptographic scheme, with the intend of finding flaws. It covers mathematical analysis to discover algorithmic weaknesses, but also side channel attacks to reveal physical implementation weaknesses. For example, by measuring power consumption or timing results, a clever adversary may reveal the internal operations and decisions of the algorithm. No encryption scheme will become popular without the scrutiny of security experts - effectively performing cryptanalysis.

But we will however seek other options. We quickly rule encryption out, because as advocated in the introductory chapter, we are beginning to see a trend towards encryption regulations. Some governments ban it completely, allow-

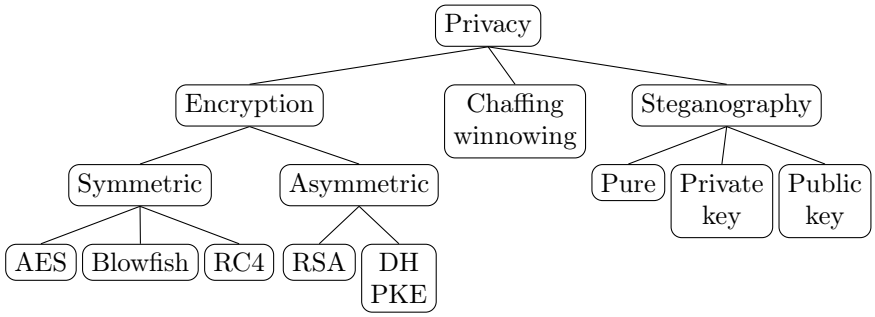


Figure 3.2: Privacy solutions

ing them to keep an eye on people. You might end in jail, simply for using applications with encryption.

Chaffing and winnowing by Rivest is an alternative to encryption, developed in times with similar regulations. Using an authentication mechanism, we may achieve privacy as well. The chaffing process is simple, and obfuscates the "real" packets from bogus. Only the recipient is able to winnow the chaff packets away, as he is the only one capable of correctly authorize packets. We may even delegate the chaffing process to some third party, or a third party may do it without our knowledge.

Only prototypes of chaffing and winnowing systems have been implemented, it has not taken off as a mainstream privacy ensuring solution. This is most likely because encryption regulations seized at the time, and encryption deemed superior. The system is stand alone, it does not extend other specific communication protocols. It follows that both recipient and sender needs special applications to communicate with chaffing and winnowing. If one party, for some reason, is unwilling to install and use it, he simply cannot participate in any communication at all.

Steganography is the last of the major research area of privacy or confidentiality. As we described earlier, it comes in 3 different forms. Pure, private- and public key.

In the pure setting, parties will need to only know the schemes encoding and decoding function. But this is weak, as once an adversary learns them, he may read all past and future messages. Over time, it is likely that he will learn them. Pure steganography violates Kerckhoffs's principle of only keeping keys secret -

not algorithms. Keys are easily changed, algorithms are not.

By introducing a key to the scheme, we arrive at private key steganography. As with symmetric encryption, a key is used for encoding and decoding. The same stego-key, a shared secret, one they need to exchange in advance, as an initialization step.

The theory we found on public key steganography is a combination of asymmetric encryption and pure steganography. As with asymmetric encryption, a key does not need to be transferred securely between parties in advance. The fact that any message may be transferred with pure steganography, and that a message can be any sequence of bits is exploited by first encrypting the message and then encoding the cipher text. Cleverly enough, only the keys and how to obtain they keys must be known in advance.

Pure steganography violates Kerckhoffs's principle, public key steganography does the same, in a way. It also employs encryption, which we previously ruled out due to regulations.

To achieve privacy or confidentiality in our solution, we will explore private key steganography.

Using private key architecture has key exchange implications, but has also shown to benefit another requirement: Plausible deniability. We wish to let users plausibly deny authoring any sent message.

Deniable encryption is plausible deniability implemented in encryption. We already ruled encryption out, but this topic has interesting aspects. It is achieved when an adversary is unable to surely prove the existence of some plaintext within some dataset. For example, the Rubberhose file system[1]. The idea behind is quite simple: Initialize the filesystem by writing random bits to the entire hard drive. Then allocate partitions in such a way that their sectors are randomly distributed on the entire hard drive. Any future write to disk will we encrypted, and thus indistinguishable from the random bits. Without the key to unlock the system, the partition could fill the entire disk as well as nothing at all. There is no way for an adversary to tell without knowing the key. If the user is compelled to give up some key, the adversary will have no means of identifying if it is a wrong key or if there is simply no partition. We can implement plausible deniability in much the same way in steganography.

Assume that some adversary compels both sender and recipient of some stego-object to give up their keys. It may be plausible enough that there is no message

[1]Rubberhose file system: `https://en.wikipedia.org/wiki/Rubberhose_(file_system)`

embedded within the object, but if both parties provide different keys, and if those keys decode different messages - the adversary will know. It is suspicious that the result of the same stego-object is different. As it is also suspicious when a criminal investigator, asks two suspects where the other was on some date, and they answer differently. If parties agree on some fake key in advance, along with the real key, they may confidently disclose the fake key and it will be plausible that their stego-object does not hold a message. This is easier with private key steganography, as parties can freely chose any key, as long as it is the same. The decoding result will be the same.

Our privacy ensuring mechanism will be an implementation of a private key steganography scheme. The field proposes tools for satisfying two sides of the CIA triad: Confidentiality and integrity. Steganography offers secrecy, which causes integrity. If an adversary is unable to detect a message, he cannot compromise its integrity.

Our transport solution offers the last side of the triad, availability. As we mentioned numerous times, there are vast methods for hiding data digitally. Before choosing a specific scheme, we will consider the transportation solution. Steganography may hide everywhere, and we already learned that some cover-objects or vessels are better suited than others. Thus we will yield the best overall result, by considering both in relation to each other.

3.2 Transmission

Our transmission solution must ensure a high level of availability. It is the driving force behind creating a digital solution. The Internet is what connects people today. The Internet has connected people in a way never seen before. It has become ubiquitous, and it is where our final solution must operate. Figure 3.3 shows popular existing communication platform categories. There are so many different services belonging to each. It is fair to say that internet users will communicate with each other, using at least one service belonging to these. We could develop our own standalone system, and we would have full technical control of our system. We would be independent of other services, and have the best technical launch pad.

However, by building a system from scratch, we also start with zero users. We wish for our system to become popular, and if possible we would like to take advantage of other systems success. If we can offer our solution as an extension to an existing service, it might be easier to reach a large user base. There might also be less work, as we can potentially extend existing frameworks and/or user

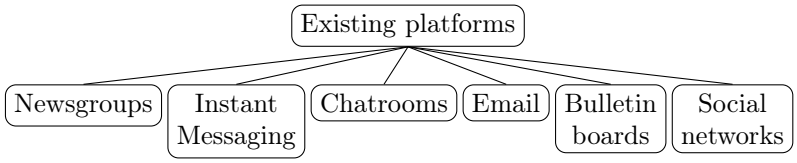


Figure 3.3: Popular Internet communication platforms

interfaces - user interfaces that will already be familiar to existing users.

Newsgroups on usenets is an old forum-like platform. It can be thought of as a forerunner to todays bulletin boards on HTTP. Users may discuss topics in designated groups. In contrast to bulletin boards, usenet is a distributed system - making it hard to take down or manipulate. In theory, anyone may set up a new server and subscribe to other servers news feeds. The decentralized structure is certainly positive. We could choose to hide our communication in existing or new groups. Some groups are even reserved for binary content, where we could upload images or videos with embedded messages.

But despite a steady increase in traffic, usenets are a niche for the tech savvy. It is doubtful that the average internet user even knows of its existence. Using usenets require special software, special in the same way as browser for HTTP. But today, operating systems ship with browsers build in, and it is fair to assume that people know how to operate them. We will look elsewhere, for a popular service we may extend on.

There are many bulletin boards and fora available online, some more popular than others. Users may create threads or posts for others to comment on. They are effectively newsgroups through HTTP. Many offer extended markup, which is great for steganography. Some allow users to post images, and some even require images with optional text. As mentioned, images are great for steganography because of their high redundancy. We could easily imagine a steganographic solution based on posting pictures online, where only the inaugurated would know where and how to extract their messages. However such a solution would quickly be noticed by the operators, if it became popular. We would like to propose a solution that is structured, robust and decentralized. Storing messages secretly in bulletin boards will not provide that.

Instant messaging has become increasingly popular. Applications like ICQ, Windows Messenger and Jabber have all been very popular. Today we have Viber, WhatsApp and more, but the trend seems like a merge between social networks and instant messaging. Twitter offers "Direct Messages" and Facebook has Messenger. While their user bases are huge, and extensions are becoming possible -

they own their platforms and control them completely. These companies profit on their ability to listen in on their users chats. It is unlikely they approve of our intentions, that go straight against theirs. It is also inhibiting that communicating parties must be online simultaneously. We require something more robust.

We will instead extend traditional emails. They are the digital version of traditional mail. They have become so common, that an explanation is not even necessary. Anyone may send an email, anyone can setup an email server. Emails have a distributed architecture and are standardized for interoperability. Even though emails may not be the preferred communication form of many, they most likely have an email address as a fallback solution. Emails are created and viewed by an email client, where we can have extensive control of what is sent out in the other end. There are many different email clients, and anyone may create their own. Emails consist of data and metadata, and support application extensions by special headers. As long as we conform to the protocol requirements, we are free to incorporate steganography as we like.

The distributed architecture, its popularity and extensibility make us chose to develop our solution with email as transportation

3.2.1 Emails

Emails are a very common digital communication form, to many its preferred. Emails can hold rich data and files, there are great contact book functionality available. Most companies integrate emails deeply in their IT setup. We believe that allowing users to keep using email, and offer our solution as an extension, is the best way to ease users in and make it popular. An application that alters emails in such a way, that users may still use emails entirely as they used to, and still send emails to recipients not using our application. Recipients outside the scheme should simply ignore any extra information in emails. We offer our extensions as a superset to ordinary emails. This way, users keep using their existing work flows, their existing contacts they simply need to install an extension.

An emails is obviously just some piece of data, structured to comply with the Internet Message Format, defined by RFC 5322[rfc08]. Email semantics are very similar to how physical mail works. You write a letter and put it in the mailbox. After some time, the recipient receives the letter by the mailman, if he is not home - it will remain in his mailbox. Similarly, emails can be written and handed over to a digital mailman, for later (but typically fast) delivery. The recipient does not need to be online to receive it, in contrast to instant message

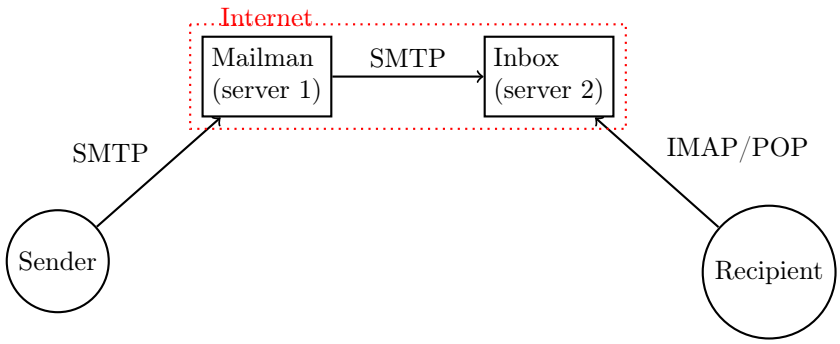


Figure 3.4: Email transfer chain

protocols.

For this thesis, it is considered common knowledge how to compose emails, create email accounts at service providers and sending them. Instead we will briefly look into how emails are handled after they are handed to the "mailman" or service provider. Emails consist of headers and a body. Metadata and data. The body holds the message, headers hold information like sender, recipient, subject and other fields important to the transfer and readability.

Transfer By choosing emails as vessels for our solution, we may draw on a mature infrastructure. To understand how, we will briefly look at how emails are transferred from sender to recipient, and which protocols take part. We will not detail thoroughly on the matter, or go into how protocols are constructed.

When an email is composed, the client hands it to a email service provider through the **SMTP** protocol. In a way, the client says: "Mailman, please deliver this email for me". After the email is delivered from sender to the service provider, it may travel many hops before finally arriving at the recipient. An example can be seen in Figure 3.4. The sender speaks with "server 1" that acts as the mailman. Typically the sender has an account at this server. The recipient may use a different provider, his inbox is at least hosted on a different server, "server 2".

Communication between involved parties follow different protocols. The sender uses SMTP to hand it over to the mailman, the mailman forwards it to the inbox with SMTP as well. The strength of standardized protocols shines through, as we can have decentralized email systems that all communicate together.

SMTP[2] is short for Simple Mail Transfer Protocol, and is in fact simple. The initiating party establishes a connection to target, then transfers the email. Header metadata enables further processing. Notice that the last link between the inbox server and the recipient does not use SMTP. The role of mailman and inbox is acquired by servers. Servers we may assume are always available. The recipient however can be mostly offline, or behind some firewall or NAT[3], using SMTP becomes impossible, as a connection is probably rarely possible to establish. Instead the connection direction is reversed. The recipient checks his mailbox at own will. Since mails are now retrieved, the direction is reversed, SMTP cannot be used. Instead we turn to either **POP** or **IMAP**.

POP[4] is the Post Office Protocol, of which the latest standard is 3. It is simple, emails can be requested and transferred to the client application. IMAP[5], Internet Message Access Protocol works similarly but offers much more complex queries. For instance, email bodies can be requested with out also receiving attached files. This allows for saving bandwidth and download time.

3.3 StegoBlock

As we have now chosen great methods for ensuring confidentiality, plausible deniability and transmission - we can return to decide on the boundaries for our steganographic scheme. In our steganography section we already noted how one might embed secret information in the markup of digital documents. Special markup that will not be rendered, but is still extractable by the initiated. Mutual for all these embedding schemes are that they belong to pure steganography. Once they are revealed, messages will be compromised and authors must devise a new scheme.

But if we do not hide our message, is it then steganography? We will use elements of steganography as an alternative way to achieving confidentiality. We will not hide the fact that there might be a secondary message embedded.

We will present *a system for hiding steganography in plain sight.*

To minimize user interaction, we will generate our own cover-objects. Having users chose their own cover-objects would require steganographic knowledge, incompatible with our vision of attracting all kinds of users. Context-free gram-

[2]RFC SMTP: `https://tools.ietf.org/html/rfc5321`
[3]NAT: `https://da.wikipedia.org/wiki/Network_Address_Translation`
[4]RFC POP3: `https://tools.ietf.org/html/rfc1939`
[5]RFC IMAP: `https://tools.ietf.org/html/rfc3501`

mars is a research field that describes how to generate text mimicking natural language. This would be useful for tricking adversaries to believe our generated text is in fact the primary message, when in fact the embedded message is primary. Context-free grammars in their current state, will not fool humans. It will still look artificial, especially when embedding longer messages. Instead we propose a much simpler solution.

We will generate a text block of finite length, with fixed characters of some alphabet to meet some target distribution. We will use steganography to hide a message in a block of text, hence the name StegoBlock. The block can be inserted in an email as metadata, in the email headers. With a stego-key, block must then be permuted in such a way that reversal is only possible by knowing the key. We will achieve plausible deniability by ensuring that all outgoing emails have our special header with our block. Adversaries will be unable to tell of a block has a message embedded or not - stego-objects will be indistinguishable from cover-objects.

The idea of an isolated block with some secure message, with emails nicely. Of course users must still write some other message in the email- body or subject, but this is tolerable. We can secure the block with steganography, instead of cryptography, thus working around regulations. To realize our solution, we would need to create an application for composing, viewing and sending emails - or at least extend an existing application.

The Thunderbird email client by Mozilla allows extension for just exactly these needs. We will extend Thunderbird with the ability to compose and view secondary, secure messages, embedded in email headers such that even non initiated recipients handle them gracefully.

3.4 Summary

We broke down the problem of allowing people to communicate confidentially, into separate solutions for transmission and securing. Of the areas encryption, steganography and chaffing and winnowing, we deemed steganography as the most promising for our needs. Especially because we ruled the entire encryption area out up front, due to possible regulations in some countries. Of the different steganography forms, we chose to implement a private key scheme. Primarily because the pure form violates Kerckhoffs's principle, because known implementations of public key uses asymmetric encryption, and because a single key is easier to work with plausible deniability.

We chose emails as transport vessel for our steganography. Emails are widely accepted, almost everyone has an email nowadays. They are easily extended and may contain both text, html and binary attachments, making them perfect for steganography. Specifically we chose to implement a block of text, that may or may not contain some secondary message.

Chapter 4

Design

We will now elaborate on the specific component design, needed for StegoBlock. Based on the previously treated theory, we have established the need for plausible deniability, and steganography as an alternative to common encryption. We will now detail more on the specific requirements for a solution meeting those needs, and keep strong adversaries at the gates.

In brief, StegoBlock is a plugin for the Thunderbird email client, it allows users to enter size limited secondary messages through a UI extension. Secondary messages will be appended randomly generated noise, and permuted. The permutation can only be reversed by knowing a stego-key. The secondary message is embedded into seemingly innocent emails, by a special email header. Thunderbird is a mail/messaging application, developed by Mozilla that shares a large code base with Firefox from the same organization. StegoBlock is an extension for this application - we will be using the terms add-on, extension and plugin interchangeably throughout this thesis.

At an early stage, we made the decision to integrate with a common email client. We could also have chosen to create a complete stand alone solution or extend some other existing platform. The motivation behind our choice is an aim for rapid spreading and we firmly believe an email extension will enable just that. We have a strong desire to make it as easy as possible, for users to pick up StegoBlock and start sending secret messages.

We also made the decision to separate our stego-object from the rest of the email. We will integrate our solution by adding our block, as an email header. It is critical that clients without StegoBlock should handle StegoBlock emails gracefully. That means they should simply ignore the relevant headers, and otherwise process the mail as normal. For StegoBlock, we need to transfer two fields: Block and seed. Fortunately, RFC 5322 details application specific headers. Headers that are specific to some application, they may be handled by a specific application and ignored by others. Application specific headers must have the "X-"-prefix.

4.1 Components

It is clear that the most work and challenge will be in designing the steganographic algorithms. However our email client extension also needs several different components, many for allowing easy user input. These must allow users to input, view and configure StegoBlock. Most components are rather simple, thus design description will be brief.

4.1.1 Composing

Email clients typically come with an interface for composing emails. We would need to extend such interface with additional fields for entering secondary messages. This could essentially be a text field. Emails bodies allow HTML, but this is of no importance to our requirements. Since secondary messages are size limited, it would be ideal if the field could display the remaining char count and disable further entering when the limit is met.

4.1.2 Viewer

We must be able to decode and visualize emails with StegoBlock headers. In some way, we must be able to read the headers of the selected email, try to decode it and visualize it to the the user.

4.1.3 Key store

Since the whole solution is based on private key steganography, it is most convenient to the user if his keys can be stored, so he does not need to remember and enter them manually. A store for keys are required, it should however be easily purgeable in case it is quickly needed.

4.1.4 Encoding/decoding

When an email is composed and the user sends, we must be able to hook into the sending process and encode the secondary message. We must also be able to generate a fake message, if none was entered. Previously mentioned components are merely plain user interface logic, our encoding/decoding logic is where we will need steganographic and cryptographic primitives.

We described our wish for plausible deniability earlier. To provide users plausible deniability to any sent message. Emails sent with StegoBlock will always contain a stego-object among the headers. If a secondary message is not chosen by the user - one will be chosen at random. Because of this, users can always claim not embedding a secondary message. The adversary will have no means of distinguishing a real stego-key from a fake. The encoded block will not reveal if any message is embedded or not, which we will later prove with a steganalysis.

For StegoBlock, we will use a PRNG for scrambling the message and appended noise. We rely on the assumption that our chosen PRNG is in fact cryptographically secure. StegoBlock is written in JavaScript - a language designed primarily for browsers. Since every browser may have its own implementation - such an assumption may not hold. Thunderbird, the only place where StegoBlock actually runs, is based on the shared code base between all Mozilla's products (Thunderbird and Firefox). It comes with a promise of being able to generate cryptographically secure random numbers.

The StegoBlock algorithms for encoding and decoding are represented as pseudo code in Algorithm 4.1 and Algorithm 4.3. We detail the functions ENCODE, GENERATENOISE and DECODE. ENCODE depends on Algorithm 4.2 for generating a string of noise adhering to the statistics of FREQUENCY_ALPHABET, with respect to the plaintext. The full application consists of several other files and functions, but these are essential to the steganography of StegoBlock. While the algorithms heavily rely on a random number generator, they are not, and do not rely on any encryption algorithms. If encryption is disallowed, these algorithms will pass.

4.1.5 Encode

```
1  input: string[] plaintext, string seed, string key
2  output: string[]
3  begin
4    if plaintext.length > MAX_PLAINTEXT_LENGTH
5      throw ERR_PT_TOO_LONG
6
7    csprng ← new csprng(seed + key)
8    plaintextLength ← leftPad(plaintext.length or random 3 chars, '
         000')
9    noise ← generateNoise(plaintextLength + plaintext)
10
11   block ← plaintextLength.concat(plaintext).concat(noise)
12   shuffle(csprng, block)
13
14   return block
15 end
```

Algorithm 4.1: Encode.

```
1  input: string[] plaintext
2  output: string[]
3  begin
4    noise ← string[]
5    dict ← new Dictionary
6
7    foreach char ptc in plaintext
8      if dict[ptc] not initialized
9        dict[ptc] = 0
10     dict[ptc]++
11   end
12
13   foreach char c in FREQUENCY_ALPHABET
14     charCount ← round(MAX_BLOCK_LENGTH / 100 * FREQUENCY_ALPHABET[c
           ])
15     charCount ← charCount - dict[c]
16
17     if charCount < 0
18       charCount = 0
19
20     foreach char in charCount
21       noise.push(c)
22     end
23     shuffle(noise)
24
25   return noise
26 end
```

Algorithm 4.2: GenerateNoise.

Let us first examine our ENCODE algorithm. Refer to Algorithm 4.1 for pseudo code, Figure 4.1 for a sequence diagram and Figure 4.2 for a visual representation of an example shuffle of the block. The example shuffle is performed with a considerably smaller max block length than usual, for clarity. Encode accepts a *plaintext* to be encoded, along with a *seed* and *stego-key*. When a user has composed an email and scheduled it for sending, this function will be run. Either the entered StegoBlock message or a randomly generated string, if no message

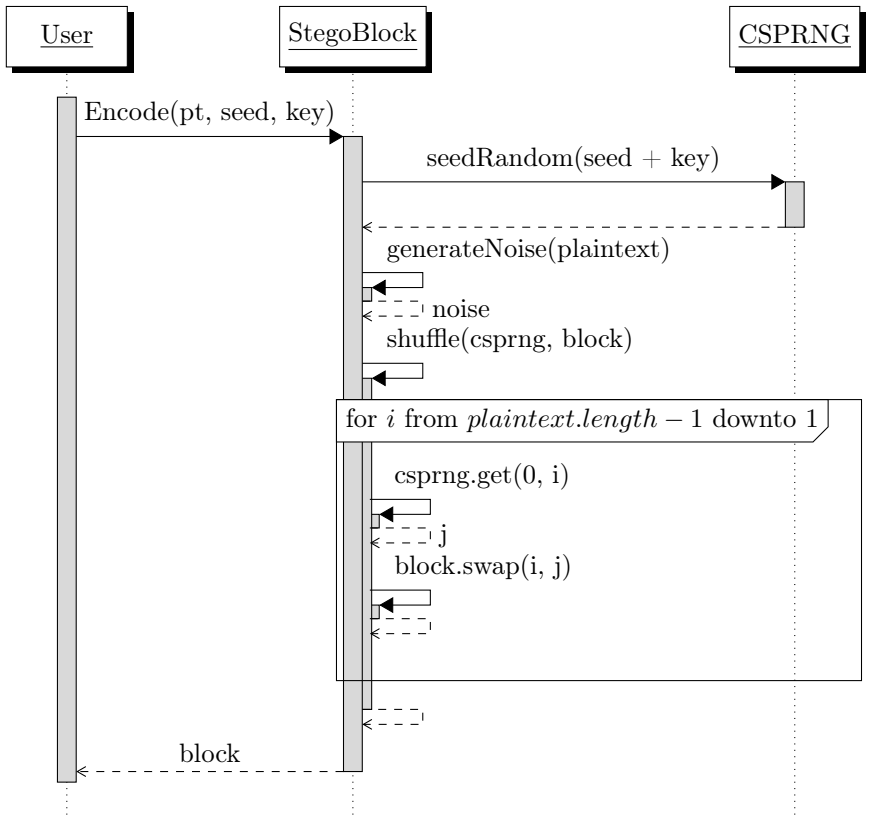


Figure 4.1: Encode, sequence diagram.

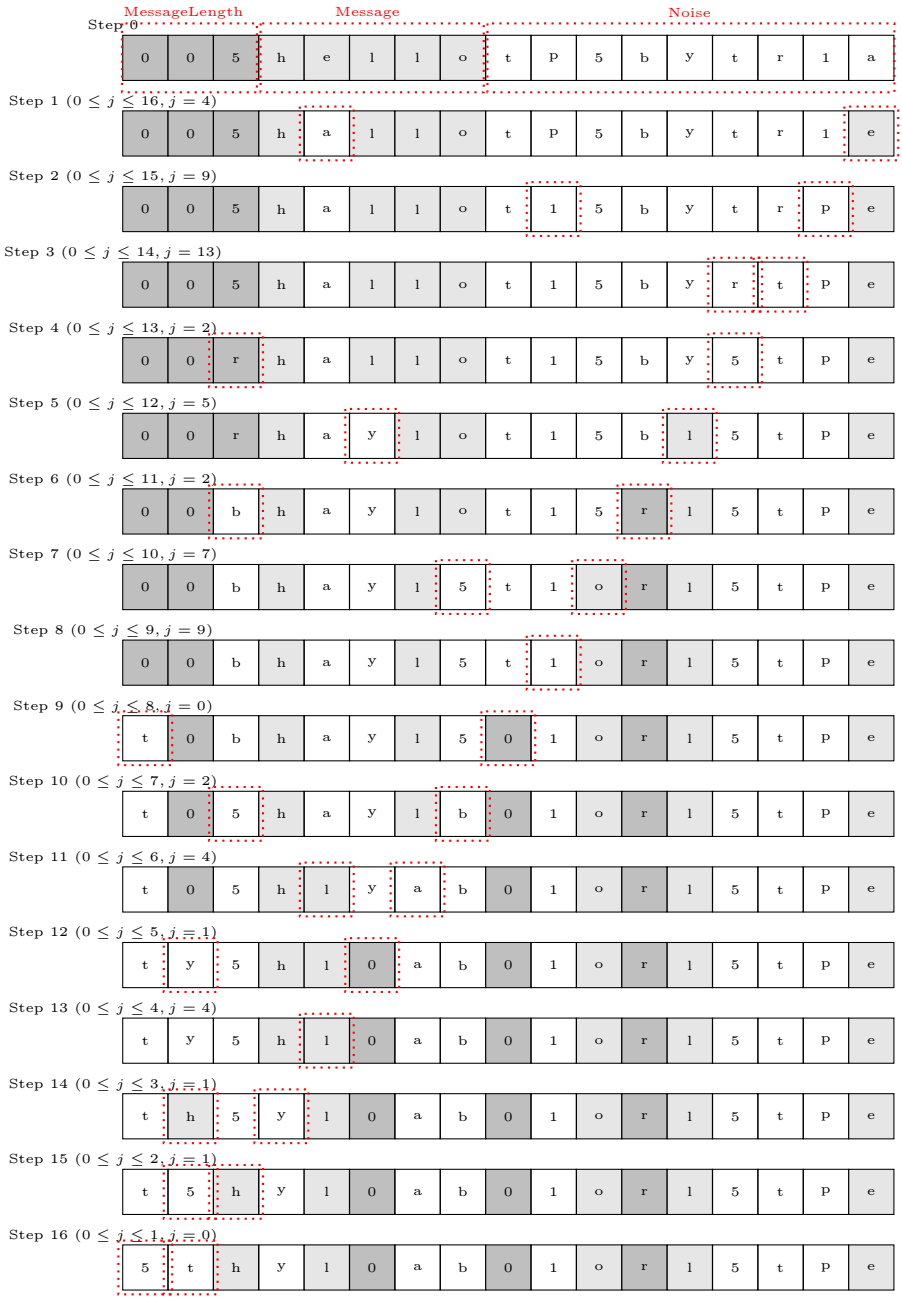


Figure 4.2: Shuffle example run

was written, will be encoded. The seed will be set to the current timestamp, with millisecond precision.

Initializing The seed serves the purpose of introducing entropy over several messages with the same key. Our StegoBlock application features a simple database for storing stego-keys, hereby encouraging key reuse, for the same receiver/sender. When reusing keys, an adversary has an easy, or at least easier, time learning the key, should he identify two or more messages with the same plaintext. Introducing the seed, which is not itself a secret, will add enough entropy to the system to allow key reuse.

The key is used for seeding the Cryptographically Secure Pseudo Random Number Generator (CSPRNG), but is first concatenated with the seed. A stego- or crypto key should always be of sufficient strength.

ENCODE will first check if the provided plaintext length is longer than the allowed. StegoBlock allows messages of maximum 200 characters. If it is longer, an exception will be thrown and execution halted (the email is not sent). Otherwise execution will proceed to initialize a CSPRNG with the seed and key. This ensures a deterministic sequence of "random" integers. It will then left pad the message length with 3 zeros. This is to ensure a 3 digit message length. Next, we generate a string of *noise*. The noise is essentially a string of characters in some alphabet, that adheres to some frequency for each letter. We then concatenate message length, message and noise into one big array, the block. Before returning, the block is shuffled with the Knuth shuffle.

GenerateNoise Refer to Algorithm 4.2 for a textual representation of the algorithm. Before inserting the plaintext into the block, GENERATENOISE generates a string of letters that adhere to some distribution. It accepts the plaintext as argument, as both plaintext, plaintext length and noise combined, must adhere to said distribution. We start off by initializing a new empty array for the noise and a dictionary for fast access to the counts of each inserted character in the noise array. In our StegoBlock application, the distribution of characters should resemble that of a predefined distribution in *FREQUENCY_ALPHABET*.

We will iterate all characters of the plaintext. Each character is looked up in said dictionary. If the value is uninitialized, it will be initialized to 0. The value for the current char is then incremented by one. One sees how the dictionary effectively keeps a count of each character in the plaintext. These counts are used later, when calculating how many of each character should be prepended

the string of noise.

FREQUENCY_ALPHABET is a predefined dictionary of characters and their desired frequency, in the resulting StegoBlock. We will proceed to iterate all its values, cf. For each character, we calculate how many of each there should be in the final StegoBlock of provided size and frequency. The result is kept in the variable charCount. We will retrieve the number of times, we counted the same character in the plaintext, and subtract this from charCount. As charCount may then become less than 0, we set it to 0, if that is the case. This problem and others will be treated later - but if charCount becomes less than 0, it means the plaintext has one or more characters exceeding their max. They will not be able to hide within the block, e.g. maintaining the target distribution. The algorithms generally postpone these checks until last minute, instead of halting, so a full error report can be provided.

We will then add the character of cf, *charCount* amount of times to the noise array. Notice that we shuffle the noise before returning. Otherwise it would be alphabetically ordered. An adversary would otherwise be able to validate a keys correctness. We will detail on this in the later steganalysis.

After concatenating $MessageLength+Message+Noise$ and encoding, checking the character distribution of the block, should result in a perfect distribution matching our target *FREQUENCY_ALPHABET*. If it does not, an error report is shown to the client, and execution halted.

4.1.6 Decode

```
1  input: string[] block, string seed, string key
2  output: string[]
3  begin
4    csprng ← new csprng(seed + key)
5    unshuffle(prng, block)
6    size ← block.remove(0, 3)
7
8    return block.range(3, 3 + size)
9  end
```

Algorithm 4.3: Decode.

```
1  input: CSPRNG[] prng, string block
2  begin
3    indexes ← []
4
5    for i from n−1 downto 1 do
6      indexes.push(prng.get(0, i))
7
8    foreach int i in indexes
9      j ← indexes.pop()
```

```
10      tmp ← block[i]
11
12      block[i] = block[j]
13      block[j] = tmp
14  end
```

Algorithm 4.4: Unshuffle.

Refer to Algorithm 4.3 and 4.4 for pseudo code, and Figure 4.3 for a sequence diagram. When an email is viewed with StegoBlock installed, the function DECODE will run if a StegoBlock email header is present. The seed is extracted and the stego-key retrieved from the database.

We start out by seeding a new CSPRNG with the seed and stego-key concatenated. We are now able to reproduce all the numbers generated by the senders CSPRNG.

To reverse the encoding process, we must generate as many pseudo random numbers as there are characters in the block. For the length of the block, we will query the CSPRNG for a new number, and push it on the *indexes* stack. We need to use the pseudo random numbers in reverse order, but can only generate them in forward order. Observe how we iterate from block length to 0, so we query the PRNG for the same ranges as when encoding. We will need to iterate the block twice, first for generating indexes, then for decoding.

Extracting chars from the block in correct order is now trivial. We iterate the block in forward direction, and pop a value from *indexes* each time. The popped value is the index to swap the current index with.

When the loop has completed, *block* will be back to $MessageLength+Message+Noise$. All that is left is to extract the message. Extracting the length of the message is trivial, as the first 3 letters are reserved for this. We then return the substring or range of *chars*, from index 3 to *size*.

4.1.7 Verification

As mentioned, encoding does not terminate immediately if the target distribution is impossible to meet. Instead a separate check is performed afterwards, allowing for a full report to the user, of all problems encountered. The function CHECKFREQUENCY accepts a single parameter of the type string, which will then be checked against the target distribution $FREQUENCY\_ALPHABET$. We will input the block returned from Encode.

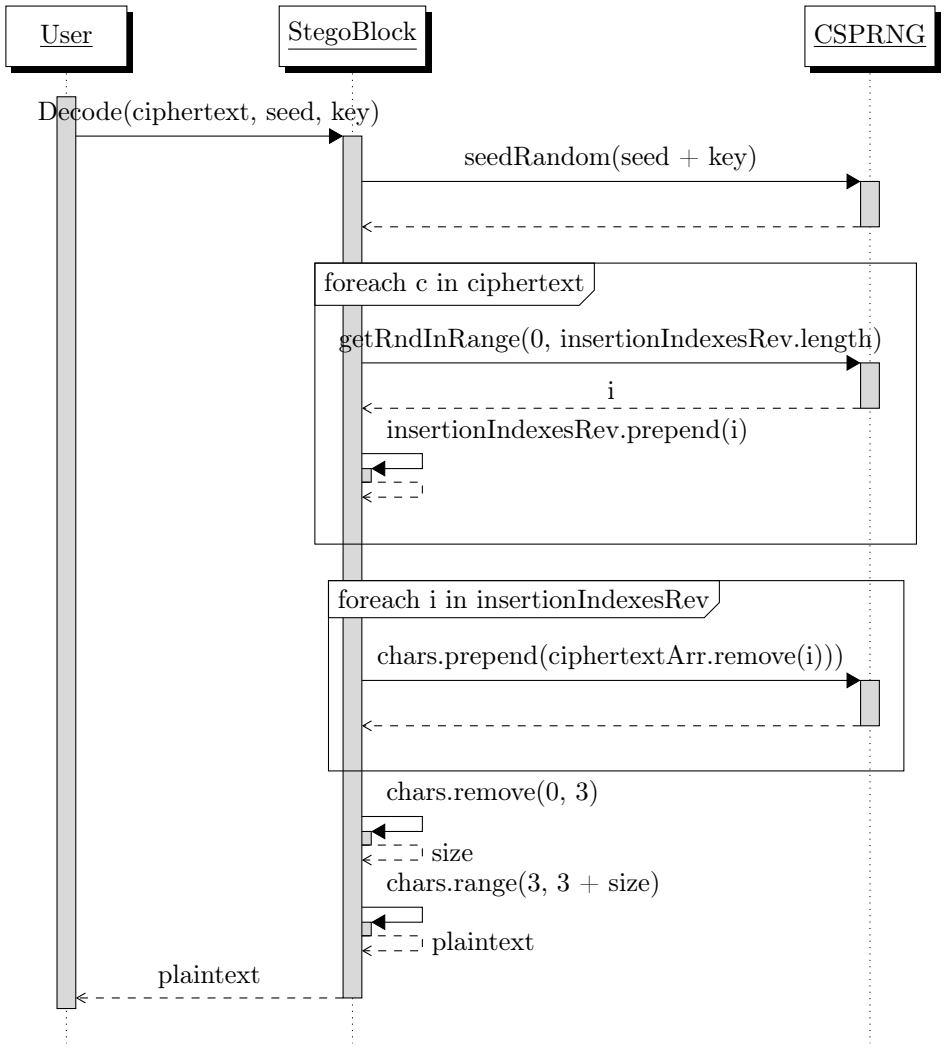


Figure 4.3: Decode, sequence diagram.

We iterate the provided string, and count the number each individual character appears. As before, we keep count in a dictionary d, for fast access. Afterwards we iterate d, and match the count percentages against the ones of the target. Should the frequency be off by more than 1 percent - the character will be flagged as invalid and appended a small report. It could also be the case that the character is not even within the target, this will also be appended the report.

If the final checkFrequency report is not clear, it will be displayed to the user and execution halted. The user will then have the opportunity to change the message into something fitting the distribution. Only blocks with a clear checkFrequency report can be sent.

4.2 Summary

The problems we detailed in our problem analysis will be solved by an email client extension. We wish to ease the use of StegoBlock as much as possible, and since email use is so widespread - this protocol provides a nice starting platform. We detailed on email protocols, and how the use of "X-"-headers allow us to create a solution that works seamlessly with users without StegoBlock installed. This allow us to truly extend on emails, users without StegoBlock will simply only learn the primary message.

We designed the components necessary to implement StegoBlock, as a Thunderbird extension. It must allow Thunderbird users to input secondary messages of fixed length, which will then have appended noise, and be scrambled. Adversaries will be unable to decode the scrambled block, without knowing the key.

We provided detailed designs of algorithms for encoding, generating "noise" and decoding secondary messages in StegoBlock. Our encoding function is able to make the resulting block adhere to a target character distribution, given by some FREQUENCY_ALPHABET. This is done as a measure against statistical steganalysis. Our design implements checks of final blocks. If they fail, execution is halted and no email is sent. Our encode function is designed to accept a stego-key, some seed and a message. The message will be analyzed, and we generate a cover-object, known as "noise", with respect to the message. Before permutation, the block will consist of $MessageLength + Message + Noise$. To permute, we setup for the previously detailed Knuth-Fisher-Yates shuffle.

To implement plausible deniability, we ensure that if no secondary message is written, one will be chosen at random. This is a simple step, but very valuable,

as previously explained. To ensure that randomly chosen messages pass the final block check, we simply generate more noise. But we should handle the first 3 characters individually, as these detail the length of our message. We cannot accept the (rare) scenario where generating noise would randomly generate noise with 3 digits in the valid range. Then users would have gibberish displayed, instead of simply no message. We also cannot simply just have 3 zeroes, it would in fact be much worse. This would mead that a decoded block should always start with 3 digits. An adversary would quickly know if a user disclosed a fake key, as he should always arrive at some block starting with 3 digits. The user should be able to disclose any fake stego-key, to preserve plausible deniability. The cannot be any difference between a wrongly decoded block containing some message, and a correctly decoded block without a message.

To overcome this, we set replace MessageLength with 3 randomly chosen values from our frequency alphabet, and repeat until it is has a non integer value. This allows for easy validation when decoding.

Furthermore we designed our encoding algorithm to not consider if its output adheres to FREQUENCY_ALPHABET during runtime, instead a separate check afterwards will do this. This ensures a more informative error report, allowing the user to fix all problems before resending his mail.

To decode, we provided pseudo-code necessary for reversing the encoding permutation. With this "unshuffle" algorithm, we can effectively separate noise from message. It requires a PRNG in the same initial state as when encoding - but this is easy for users knowing the key. The secret key, appended a publicly available seed will do this.

Chapter 5

Implementation

Based on the design, we will implement StegoBlock as a Thunderbird extension. These are written in JavaScript, and so we do not have the luxuries of type safety, but are limited on availability of cryptographic libraries. For instance, we will resort to a third party library for a seedable PRNG.

Let us first examine the user interfaces implemented for composing, viewing and configuring StegoBlock.

5.1 UI components

In the past, Thunderbird has been very popular and under rapid development. Their website vaguely mentions "millions of users". However interest in the project has declined recently. On July 6, 2012, Mozilla announced that Thunderbird development would no longer be a priority. This was a result of the application not gaining larger user base, despite efforts. It is however still a very feature rich email client.

As mentioned, Thunderbird shares a large part of its code base with Firefox,

and a shared documentation can conveniently be found on Mozilla's website[1]. Unfortunately, documentation is not always complete or up to date. Often developers are left searching email lists or online forums for answers.

Extensions are written in JavaScript for logic, and XUL (very similar to HTML) for design markup. If one is already familiar with JavaScript and HTML, writing Thunderbird extensions will be easy.

Before diving into any specific code of our StegoBlock extension, we will examine the structure of Thunderbird extensions. Our file structure is as shown in Listing 5.1. Thunderbird largely dictates this structure, which ensures nice separation actual content and meta data.

```
stegoblock@toftegaard.it/
 chrome/
  content/
   common.js
   icon.png
   messenger.js
   messenger.xul
   messengercompose.js
   messengercompose.xul
   options.js
   options.xul
   seedrandom.js
   steganography.js
 defaults/
  preferences/
   defaults.js
 chrome.manifest
 install.rdf
```

Listing 5.1: StegoBlock extension file structure

`stegoblock@toftegaard.it/install.rdf` stores all metadata relevant for an extension. Users considering installation, may examine an add-on by the data of this file. In here, we specify extension name, description, version, author, and a unique identifier - which also dictates the name of the root folder. We may also specify supported Thunderbird versions. See Appendix C.1 for our install.rdf file contents.

`stegoblock@toftegaard.it/chrome.manifest` Thunderbird allows extensions to hook themselves into the existing UI. This makes extending existing designs

[1]Thunderbird developer portal: `https://developer.mozilla.org/en-US/Add-ons/Thunderbird`

easy. The existing design is naturally also stored in .xul files. in `chrome.manifest` one may specify which .xul files to extend, and which files will provide the actual extension. By installing another extension *DOM Explorer*, one may point and click the existing UI to reveal which .xul files define them. This will also reveal the entire markup, and allow inserting custom markup. See Appendix C.2 for our chrome.manifest file contents.

`stegoblock@toftegaard.it/defaults/preferences/defaults.js` describes how an extension with to use the build in preference system. StegoBlock uses it for storing stego-keys.

`stegoblock@toftegaard.it/chrome/content/` is the default folder for storing markup and logic. This is where we keep .xul files for injecting our changes to the default UI, and the .js files, with logic for creating blocks. StegoBlock extends 3 different areas of Thunderbird: Compose message window, Read message window and preferences.

`.../common.js` Several areas require much of the same logic. To follow the DRY-principle[2], we will keep such logic in a common file - importable by all other files. Common.js has functionality for interacting with Thunderbird preference store, to store stego-keys. There is also some simple utility functions for merging objects - which is useful when including files in others. The file can be seen in Appendix C.3.

We use common.js to interact with the Mozilla Preference store. This is a convenient store for user specific preferences. This store is not secure. Any stored preferences may be extracted easily as plaintext. StegoBlock runs only on local machines. The passwords are as insecure or secure as any other unencrypted document on the users computer. In the current version of StegoBlock, we rely on users protecting their computer with authorization on operating system level. Most systems also offer system-wide encrypted, which would effectively also encrypt the preference store. It would most definitely be good to encrypt the stored stego-keys, but because of the aforementioned, we can do without in a version 1.

`.../icon.png` is a simple icon for presenting StegoBlock. It was found on `https://www.iconfinder.com/icons/811473`, with a "Free for commercial use" license. Can be seen in Appendix D.1.

[2]DRY: `https://en.wikipedia.org/wiki/Don%27t_repeat_yourself`

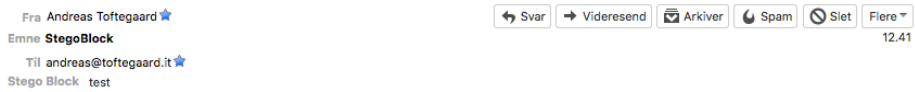


Figure 5.1: StegoBlock view-message-window excerpt

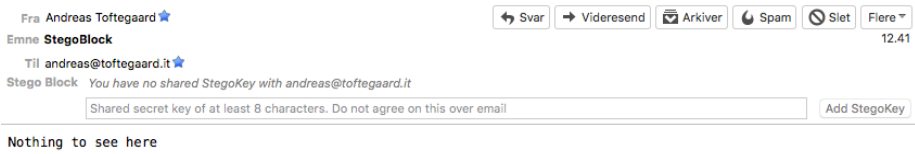


Figure 5.2: StegoBlock view-message-window excerpt

`.../messenger.js/.xul` Changes to the View-message-window are minimal. When users select an email, StegoBlock extension will check for the X-Stegoblock header. If present, it will be checked if a stego-key for the sender is available. If also present, DECODE will run. Otherwise the user will be presented for a text box to enter a stego-key. See Figure 5.1 and 5.2 for UI excerpts demonstrating the functionality, and Appendix D.2 and D.3 for the same in their context.

The block is easily readable and distinguished form the rest of the email. Our .xul file in Appendix C.4 has a single root element. We see that it is defined as `<vbox id="expandedHeadersBox">`, meaning that this element will be inserted in the existing node with `id="expandedHeadersBox"`. We also see how we include the files `seedrandom.js`, `common.js`, `steganography.js` and `messenger.js` with `<script>` tags, as known from HTML. This is how we link logic to our UI. In `messenger.js` shown in Appendix C.5, we will add an event listener to the window, which will notify when the UI is injected and the window fully loaded. Then we can easily check for X-Stegoblock header, extract, decode and show it. The logic for extracting and showing embedded blocks are as described in Algorithm 5.2, which we might call for both the sending and receiving address. We do this to allow StegoBlock extraction of sent mails as well. Thunderbird offers no way to identify if a mail was sent by the client or received, therefore we cannot determine which address/key pair to lookup in our preferences. Since there can never be more than two tries, and since we are likely to receive more mails than we send, we simply first try to lookup the "from" email and then the "to" if it fails.

```
1  input: address, email
2  output: string
3  begin
4    preferences ← getPreferences('addressesAndKeys')
5    address ← normalize(address)
6    key ← preferences[address].key
7    ciphertext ← email.headers.get('X-StegoBlock')
8    date ← email.headers.get('X-SBDate')
```

```
9
10    if ciphertext is not NULL and date is not NULL
11      return decode(ciphertext, date, key)
12
13    return null
14  end
```

Algorithm 5.1: extractStegoHeader.

ElementMap Most our logic files will interact with the UI. For a small application like StegoBlock, implementing design patterns like Model-View-Controller is too much. It takes larger applications to benefit from such patterns. If a logic file interacts with the UI, it will have a special dictionary of elements, known as an ElementMap. Selecting elements for interaction is done by querying the DOM for a unique identifier. This is a slow process, and therefore we may store it in our ElementMap for quick later retrieval.

Silent failing Anything that might cause the algorithms in our messenger window to fail, will be suppressed. In a future version, a facility for error logging would be desirable, but unnecessary for an initial version. If a StegoBlock is unextractable, we choose to not bug users about it.

By embedding our block in the email header, we will be able to quickly extract it, without downloading and parsing the full email body. Email clients typically show header information, before downloading the entire email.

`.../messengercompose.js/.xul` The compose window offers functionality to compose emails. Naturally we will inject custom UI for embedding StegoBlocks. This is the most advanced window of StegoBlock. We will inject out UI as previously explained, and hook in the logic by listening for the "compose-send-init" event, which is fired when the compose window opens. Additionally we will also listen to the "compose-send-message" even, fired when an email is about to be sent.

Our UI will monitor the recipients of the email being composed, and check if a stego-key can be found. If so, a small text area allowing up to 200 characters will be shown. If not, users may enter a new stego-key for the specified address - in the same way as in the view-message-window. See Figure **??** and **??** for UI excerpts demonstrating this functionality. Both examples can be seen in their context in Appendix D.4 and D.5 respectively. Appendix C.7 and C.6 shows the files in their entirety.

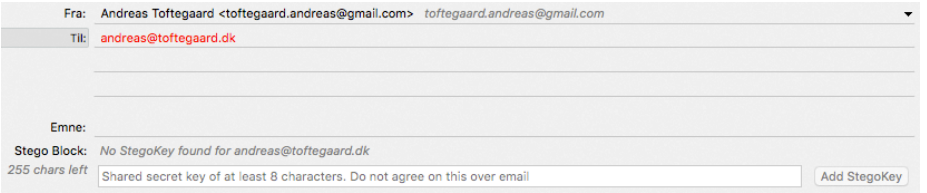


Figure 5.3: StegoBlock compose-message-window excerpt

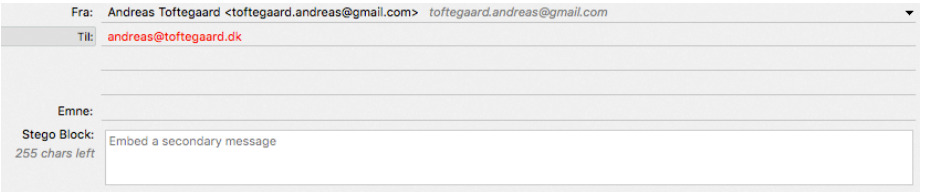


Figure 5.4: StegoBlock compose-message-window excerpt

When the users submits the email for sending, StegoBlock will intercept and inject a StegoBlock header before sending it off. The process is fairly simple and depicted in Algorithm 5.3. Notice that if an exception occurs while trying to send, it will return false and halt email sending. It is not allowed to send an email without a StegoBlock.

```
1  input: email, message, key
2  output: string
3  begin
4    try {
5      date ← Now
6      if key is NULL
7        key ← getRandomAlphanumericOfLength(128)
8      block ← encode(message, date, key)
9
10     email.headers.set('X-StegoBlock', block)
11     email.headers.set('X-SBDate', date)
12   } catch(e) {
13
14     alert('err', e);
15     return false
16   }
17   return true
18 end
```

Algorithm 5.2: injectStegoBlockInMessageHeader.

`.../options.js/.xul` Users need a place to store stego-keys. Thunderbird extensions may store arbitrary information in "Preferences". This is where we store stego-keys. It would be possible to encrypt keys before storing, but

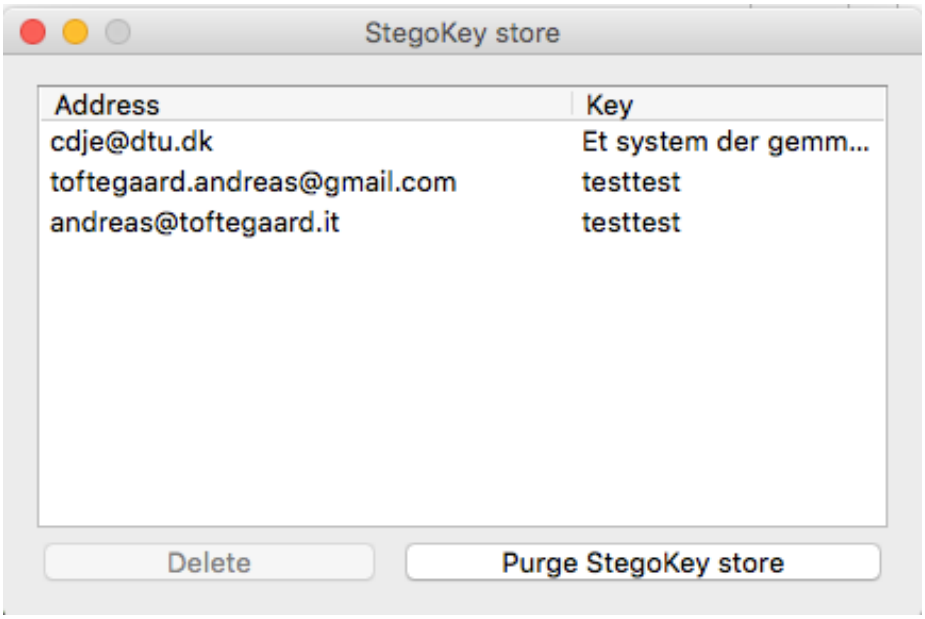


Figure 5.5: StegoBlock options-window excerpt

since Thunderbird is a desktop application - we can rely on the authorization mechanisms build into the operating system. Our options page provide a simple interface for managing keys and addresses. See Figure 5.5 for an excerpt of the options UI and Appendix D.6 for the same window in its context. It offers functionality for deleting one or more selected keys, as well as a button for purging the entire store. Should a possible adversary require access to the users email client - the keystore may be purged quickly, if known in advance.

`.../seedrandom.js` Thunderbird offers a cryptographically secure random number generator, as described earlier. It does not offer a way to seed it, unfortunately. To overcome this limitation, we use an existing open source library by David Bau "seedrandom.js" [3]. It can be seen in its entirety in Appendix C.10. A cryptanalysis has not been made of this library, thus we trust the author completely. Being able to seed a random number generator, allows us to secretly share the future numbers generated with a specific person. This is not an encryption scheme, but simply a tool from our standard cryptographic toolbox.

[3]seedrandom.js: `https://github.com/davidbau/seedrandom`

`.../steganography.js` The file containing all logic for our steganographic calculations. This is where ENCODE and DECODE lives. The full file is available in Appendix C.11. As this file is the core of our application, we will detail it more closely.

The property `maxPlaintextLength` defines the maximum length of a StegoBlock message. In order to allow enough entropy, the maximum message length must be lower than the block length. Therefore we set it to 200, and `blockLength` to 4400. We will detail on our choice for these values, in our steganalysis chapter.

The property `alphabetFrequencies` is the constant we previously referred to as FREQUENCY_ALPHABET. Here it is a dictionary of letters and the frequencies we would like them to appear with in the block. For example we have $e = 8.38191046$, meaning that 8.38191046 percent of the block must be 'e's within a range of `allowedOffset`, which is set to 1 percent. Any character that should be allowed in the block, has to be present in the `alphabetFrequencies`. This ensures it appears with correct frequency, and that function GENERATENOISE also may generate it. If we allowed characters in the block, which were not present in `alphabetFrequencies`, GENERATENOISE would never generate noise with that character - and we would leak the fact that our message contained that specific character.

The remaining functionality is functions ENCODE, DECODE and GENERATENOISE, which are all viewable in their entirety and context in Appendix C.11.

Encode is seen in Figure 5.4. First we reject messages exceeding the maximum allowed length. StegoBlock length are finite, their messages shorter, and thus we must check for this. If length is ok, we proceed by converting the plaintext to an array, it is easier to work with. We then leftpad the message length with zeroes, to ensure a 3 digit format. Notice that if no message was entered, the length will be set to some 3 character value, that is never a digit. This is in contrast to simply setting the length to '000'. If we did that, adversaries could validate a stego-key by either finding some message or a block beginning with three zeroes. We then generate sufficient noise, implementation can be seen in Figure 5.5. We then shuffle our block, with the Knuth-Fisher-Yates shuffle. Lastly we convert all spaces to non-breaking-spaces, explained in the following section (cf. §5.2).

GenerateNoise starts by iterating all characters of the plaintext, and counting how many times each occur. Each count is stored in a dictionary (character, count) for fast access. We then iterate the frequency alphabet. We let charCount be the count of how many times each char must occur in the final block. ptFreq is a count of how many times it already exists in the plaintext, and we can easily deduct the remaining count. We will append the char this many times to the

result. Notice that the result will be ordered until we deliberately shuffle it. If noise needs 3 A's and 2 B's, the result will be 'AAABB'. One could make the already discussed mistake that this is fine, because we permute the entire block, of $MessageLength + Message + Noise$, when in fact it would ruin our plausible deniability.

```
encode: function (plaintext, seed, key) {
  if(plaintext.length > this.maxPlaintextLength)
    throw 'Plaintext too long';

  let plaintextArr = typeof plaintext === 'string' ? plaintext.
      split('') : plaintext; // convert plaintext to string array
  let length = plaintextArr.length.toString();

  if (plaintextArr.length === 0) {
    while (this.isPositiveInteger(length))
      length = this.getRandomString(3);
  }

  let prng = new Math.seedrandom(seed + key); // seed the prng with
       desired key
  let sizeArr = this.leftPad(length, '000').split('');
  let noise = this.generateNoise(sizeArr, plaintextArr); //
      generate noise with correct letter frequencies
  let block = sizeArr.concat(plaintextArr).concat(noise);

  this.shuffle(prng, block);

  return block;
},

shuffle: function (prng, arr) {
  for (let i = arr.length - 1; i > 0; i--) {
    let j = this.getRandomInRange(prng, 0, i);
    let temp = arr[i];

    arr[i] = arr[j];
    arr[j] = temp;
  }
  return arr;
}
```

Listing 5.4: Encode and Shuffle implementation

```
generateNoise: function (sizeArr, plaintextArr) {

  let input = sizeArr.concat(plaintextArr);
  let noise = [];
  let ptDict = {};

  // verify that all chars in plaintext exist in the alphabet.
  // track how many times each char occur.
  for (let i = 0; i < input.length; i++) {

```

```
11      // init bucket if none exists.
12      if (ptDict[input[i]] === undefined)
13        ptDict[input[i]] = 0;
14
15      // increment char count.
16      ptDict[input[i]]++;
17    }
18
19    // run through all chars of the alphabet.
20    for (let x in this.alphabetFrequencies) {
21
22      // calculate the char count given the specified block length
            (4400) and frequency
23      let charCount = Math.round(this.blockLength / 100 * this.
            alphabetFrequencies[x]);
24      let ptFreq = ptDict[x] || 0;
25
26      charCount = charCount - ptFreq; // subtract the char count in
            the plaintext, from the calculated.
27      if (charCount < 0)
28        charCount = 0; // there is already too many of the given char
              , to maintain correct frequency. notify about this later.
29
30      // as the frequency and char count calculated is now with
            respect to the plaintext, push the char onto the noise
31      // array "charCount" times.
32      for (let i = 0; i < charCount; i++)
33        noise.push(x);
34    }
35
36    // shuffle noise, as we would otherwise reveal if some key is
          fake and ruin plausible deniability.
37    this.shuffle(new Math.seedrandom(), noise);
38
39    return noise;
40  }
```

Listing 5.5: Generate Noise implementation

Decode is seen in Figure 5.6. We start by initializing a PRNG to the same seed as we expect the block to be permuted with. We then make the block an array, this is easier to work with. We proceed to unshuffle and block goes from scrambled, back to $MessageLength + Message + Noise$. We then extract the message length, cut out the message part of the block and return it.

```
1  decode: function (block, seed, key) {
2
3    let prng = new Math.seedrandom(seed + key);
4    block = block.split('');
5
6    this.unshuffle(prng, block);
7
8    let sizeStr = block.slice(0, 3).join('');
9
```

```
10    // 3 first chars must be digits to be valid
11    if (!this.isPositiveInteger(sizeStr))
12      return '';
13
14    // parse the size of the plaintext to an int, so we can slice it
          off
15    let size = parseInt(sizeStr);
16
17    // must be valid length
18    if (size < 0 || size > this.maxPlaintextLength)
19      return '';
20
21    return block.slice(3, 3 + size).join('');
22  },
23
24  unshuffle: function (prng, arr) {
25
26    // generate all swapping positions needed, so we may start with
          the last one.
27    let indexes = [];
28    for (let i = arr.length - 1; i > 0; i--)
29      indexes.unshift(this.getRandomInRange(prng, 0, i));
30
31    // reverse knuth-fisher-yates shuffle
32    for (let i = 1; i < arr.length; i++) {
33
34      let j = indexes.shift();
35      let temp = arr[i];
36
37      arr[i] = arr[j];
38      arr[j] = temp;
39    }
40    return arr;
41  }
```

Listing 5.6: Decode and Unshuffle implementation

5.2 No Linear-White-Spaces

When a plaintext is encoded and verified, the initial idea was to simply send the resulting block as it was. It would have the security and character distribution desired. But our choice of embedding it in email headers had issues, requiring us to handle consecutive spaces.

Remember that after filling up the block with message and noise and permuting it, characters will be at random positions. Many, if not most, of these characters will be spaces. A character distribution with spaces of a frequency around 15-17% is considered normal for english. Some of these spaces will be appear next

to each other, which became a problem. We initially observed that Thunderbird would transform multiple adjacent spaces into a single space.

To find a proper explanation, one will have to search email lists or user forums for accurate information, as documentation of the Thunderbird API's are often not properly updated. According to the author of a rewrite of the email header functionality[4], the correct way to insert custom headers, is with the following (JavaScript) code:

```
gMsgCompose.compFields.setHeader('X-StegoBlock', block);
```

Most unfortunately, the `setHeader` method converts multiple adjacent spaces into one. Any leading and trailing spaces will be stripped, any adjacent spaces, regardless of position, will be converted into one. Thunderbird is an open source project, so by inspecting the source code[5], we found that the following regular expression replacement causes us trouble: `fieldValue.replace(/(?:(?:\r\n)?[ \t]+)+/g, " "))`. According to the comments, any Linear-White-Space will be replaced with a single space.

Researching this behavior, brings us to RFC-822, that classifies a Linear-White-Space as any sequence of spaces, horizontal tabs or line breaks that is followed by at least one space or horizontal tab[rfc82]. According to RFC-2047, there cannot be any Linear-White-Spaces in email headers[rfc96], which clearly explains the Thunderbird implementation.

It is unavoidable to have adjacent spaces, so we must handle this obstacle gracefully. Remember that the character order of our block, is crucial to successful extraction. It is the CSPRNG that dictates permutation, and therefore we cannot just "move" away some character in an encoded block, depending on its neighbors. A possible remedy is to replace all spaces with some other character in the standard US-ASCII table, like underscore. We would then need to escape any existing underscores, but that is trivial. However, we will chose another option. Email headers cannot contain non ASCII characters[rfc96], but as we wish to allow users to enter messages in their native language, special characters will be unavoidable. We will therefore exploit that `setHeader` recognizes non ASCII characters and performs a nice conversion according to RFC-2047. After their conversion, each header line will start with the character-set, followed by the then encoded line. This will also nicely handle *line folding*, which is also

[4]Inserting custom headers: `https://groups.google.com/d/msg/mozilla.dev.extensions/s4oFmM8_B28/AxEKZ-SZsnoJ`

[5]Thunderbird normalizeFieldValue source: `https://dxr.mozilla.org/mozilla-central/rev/82d0a583a9a39bf0b0000bccbf6d5c9ec2596bcc/addon-sdk/source/test/addons/e10s-content/lib/httpd.js#4639`

required by RFC-2047. Lines in email headers may not exceed a length of 76 characters, but there may be as many lines as necessary:

The length restrictions are included not only to ease interoperability through internetwork mail gateways, but also to impose a limit on the amount of lookahead a header parser must employ (while looking for a final ?= delimiter) before it can decide whether a token is an encoded-word or something else.

So our solution is very simple. We will encode the StegoBlock header before embedding it, in such a way that there are no spaces. A simple URL encoding will be sufficient. This will ensure that all spaces, tabs and carriage returns are escaped, all linear-white-spaces will be removed. Functions `encodeURIComponent` and `decodeURIComponent` are build into Thunderbird.

We briefly mentioned that `setHeader` handles header folding, but this is not entirely true. When we encoded all white spaces, we encountering other troubles, where no header was set at all. There were no exceptions thrown, simply just no headers. It turns out that Thunderbird is unable to correctly fold headers if they contain no white spaces. To remedy this, we manually split headers in chunks of 76, separated by a space. This allows Thunderbird to fold our header correctly and we must simply remember to remove these spaces again, when decoding.

5.3 Block example

As we saw a visualization of the ENCODE function in Figure 4.2 of how blocks are filled, and since headers are now encoded to eliminate issues with linear-white-spaces, examples of actual headers provide little extra clarity. Appendix A.1 shows the source of an entire email with an embedded StegoBlock. Notice headers **X-Stegoblock** and **X-Sbdate** that store the block and seed respectively. The particular header stores the StegoBlock message "test", using the stego-key "testtest".

5.4 Summary

We implemented the necessary components for StegoBlock as a Thunderbird extension. Implementation was in JavaScript, this is the only option for Thunderbird extensions. We created user interfaces for composing and viewing StegoBlocks, along with a simple interface for administrating stego-keys.

We thoroughly implemented functions Encode, Decode and GenerateNoise as designed. We encountered issues with linear whites paces, that would corrupt out blocks when embedded in email headers. The issue was handled by URL encoding the header value before inserting. This would be needed in future versions anyway, for frequency alphabets with characters outside ASCII.

Chapter 6

Evaluation

The evaluation of our results and StegoBlock will effectively be a steganalysis. We set out to create a private key steganographic solution, with emails as transport. To allow users confidential communication, without using traditional encryption. We did so because, strong encryption has been banned in some countries, and because we believe everyone should have the right to communicate in private. We will now proceed to evaluate our results, as done in a typical steganalysis. The StegoBlock encoding algorithm uses no other cryptographic tool, but a CSPRNG. From this alone, we achieved confidentiality.

6.1 Key exchange

StegoBlock employs private key steganography. Participants must secretly share a stego-key. There exists many different key exchange protocols, but Stegoblock does not offer any. It would be convenient, but one is not offered primarily due to time constraints. Obviously, users should not agree to a key over email, but instead use a different channel. A good option would be for users to meet in person, and agree without eavesdroppers listening in. *Furthermore users should also agree on a fake key.* This strengthens their plausible deniability, as an adversary may interrogate them both, and arrive at the same block decoding

result - if they provide the same key. If they tell different keys, it will surely arise the suspicion of the adversary.

6.2 Encoding

Our encoding function achieves similar results as encryption. It is easy to execute in one direction, hard the other way without also knowing the key. This is the core of encryption, but we can achieve this property with other algorithms as well. StegoBlock has the function ENCODE, which will accept plaintext, key and seed. With a one-way function it will scramble the plaintext in such a way that it will be hard to reverse, without knowing the key and seed. A one-way function, or trap-door function is not supposed to be kept secret. Otherwise it would also violate Kerckhoffs's principle. One can think of a real world example of a one-way function, to understand it better. Anyone can know and understand the process of turning cow into minced meat, it is easy. Turning minced meat into cow, is much harder - regardless of any meat processing knowledge.

We may theoretically choose any FREQUENCY_ALPHABET, StegoBlock comes with the one seen in Appendix E.1. It has 95 different letters. The frequency of each letter is fixed - it will be the same for each run. Our chosen StegoBlock length is 4400. We choose this number for both security and usability reasons, but more on this matter later (cf. §6.3). Since we ensure the same frequency of each letter, and the length is fixed - our block will always be a permutation of the same string. There will never be an occasional overrepresentation of some letter, leaking our message. A StegoBlock message with an abnormal letter distribution will never be sent.

ENCODE will first generate a block in the form of $MessageLength + Message + Noise$, where Noise makes the total block conform to the target letter distribution. Then it will proceed to permute with a CSPRNG. We chose the Knuth shuffle as our permutation function, ensuring that our resulting permutation belongs in the set of all possible permutations. We have already detailed the inner workings or the Knuth shuffle and its ability to chose between the entire set of permutations, which is defined by $maxBlockLength!$, in our case 4400! (factorial), including duplicates. There are extremely many possible permutations, the Knuth shuffle promises perfect randomization, the block always consists of the same 4400 letters, independent from the embedded message. Our CSPRNG used in the Knuth shuffle is seeded with a secret key, and a once time seed. Sending the same message many times will not leak, because of the seed, as this is set to the current timestamp. The seed also ensures an *avalanche* effect. Encoding the same message repeatedly will use different seeds (different

seconds or milliseconds), and the resulting permutation will be much different, even though the message and stego-key is the same.

Based on this, we consider our encoding function secure to our knowledge.

6.3 Block length

We add noise to the block for two reasons: security and usability. If no noise was added, we would have a block in the form of $MessageLength + Message$, resulting in variable block length - depending on message length. We would leak the message length, we would also have a much smaller permutation space. If messages were on average 100 characters, we would have "only" 103! possible permutations (including duplicates). Because an adversary knows all characters are used, and can possibly derive the language, many permutations may be ruled out. For example, a message in english would be unlikely to have more than two of the same letters next to each other. We add noise to increase the permutation space, and to uniform the character use and their frequency. A max block length of 2000, giving 2000! permutations could also be considered secure, but remember that the set of characters are fixed. The user should be able to express most messages with the characters available. Remember that messages that does not conform to the chosen FREQUENCY_ALPHABET will not be send. Increasing block length expands the set of possible messages and introduces the necessary security.

6.4 Message analysis

To ensure that users may send a reasonable amount of different messages, we will analyze a large message corpus, and see how many may be embedded in a StegoBlock of different max lengths. To establish the best foundation for our analysis, we estimate that the best dataset for testing how messages may be embedded, is one of natural language. Compiling such a message corpus can be done in many ways. One possibility could be to data mine Twitter, but tweets are limited to 140 characters, and feature abnormal high use of hashtags and at's. Instead we looked for existing email corpora and found the Enron email corpus[1] by William W. Cohen at Carnegie Mellon University. Later integrity problem fixes and availability by Leslie Kaelbling at MIT. We picked 5000 messages at random, no criteria of any kind. We then filtered away anything but the raw

[1]Enron May 7, 2015: `https://www.cs.cmu.edu/~./enron/`

message. This means mail headers and markings were removed. We then arrived at a large set of messages in natural english language. A message corpus of real emails, written by humans, and of such size should be excellent for further analysis. The average email length was calculated to be 950 characters, but these are all intended as primary messages. It is fair to impose a smaller max length to secondary messages.

Remember that our Total Block Length consists of $MessageLength+Message+Noise$, and that the total letter distribution must adhere to some FREQUENCY_ALPHABET. Before being able to run any tests, we must compile a frequency alphabet. We have compiled this by analyzing the occurrence of each character on a set of randomly chosen emails from our mail corpus. Time wise it would be too big of a computational task to analyze all emails. Basing both tests and frequency alphabet on the same dataset is not considered a problem, as messages was picked at random, from a set of 520.901 emails.

We have tested the parameter Message Acceptance Rate(MAR) by increasing Total Block Length(TBL). We want to know how successful we can embed a set of messages, given some TBL. Furthermore, we have run the tests twice, first for a fixed max message length of 200 and 140 characters, then by a dynamic max message length of 1/4 of the Total Block Length. All tests were run with an Total Block Length in the range 800 - 15000, in steps of 200. We picked a fixed message length of 140 and 200 for the first tests, as we would like to know how much noise is needed to embed small capped messages. It seems people can express most thoughts in a tweet of 140 characters, and thus 200 seems as a reasonable increment, if we need to avoid the contracted wordings of Twitter. *The start index of the capped ranges, is picked at random* for each message - preventing that our samples all start with "Hi, Hello, Dear" and the like.

Figure 6.2 shows how the MAR increases along with TBL and a capped message size of 140 - a tweet length. At TBL 800, we accept 78.70%, and at 15000, we accepted 98.57%. A TBL of 15000 is however very large, and sending such large emails would impact recipient inbox space and bandwidth significantly. If an acceptance rate of 90 is acceptable, we could do with a TBL of 1800 - which is much more acceptable. This is however under the assumption that 140 characters is a good fit, and especially that no message is longer than 140 characters. The MAR is an expression of how well the messages fit into our FREQUENCY_ALPHABET, without considering their length.

Figure 6.1 shows the same chart for a message size of 200 - the reasonable increment to a tweet. At TBL 800, we accepted 34.7%, and at 15000, we accepted 96.8%. If again an acceptance rate of ~90% is acceptable, we could do with a TBL of 4400.

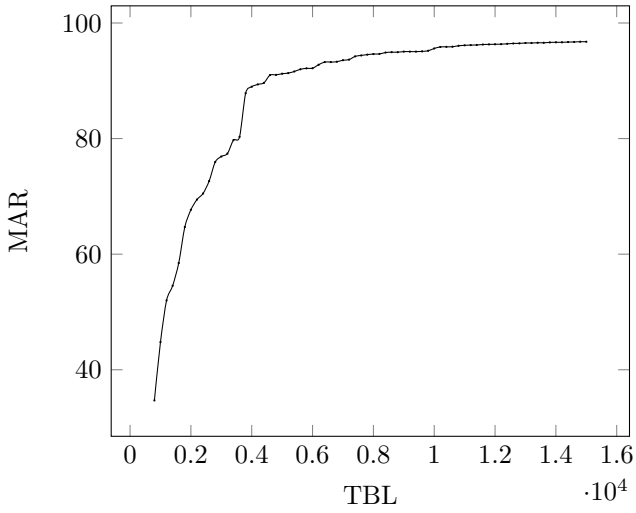


Figure 6.1: TBL vs. MAR - capped message size of 200

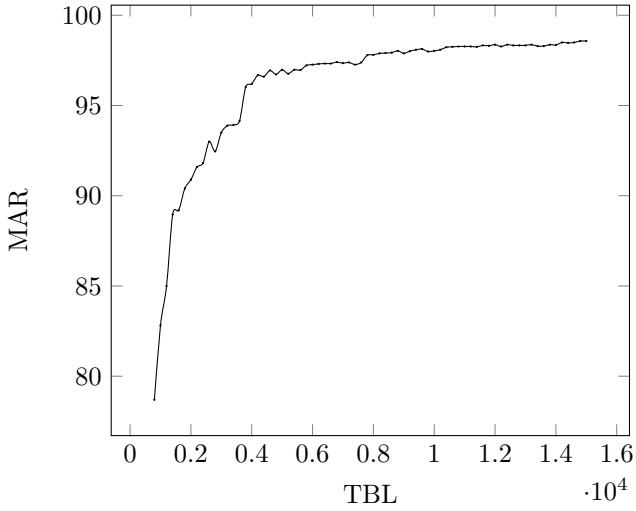


Figure 6.2: TBL vs. MAR - capped message size of 200

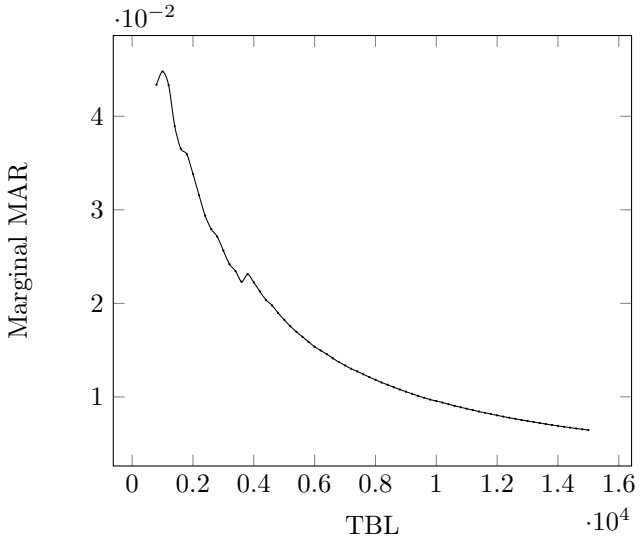


Figure 6.3: TBL vs. Marginal MAR

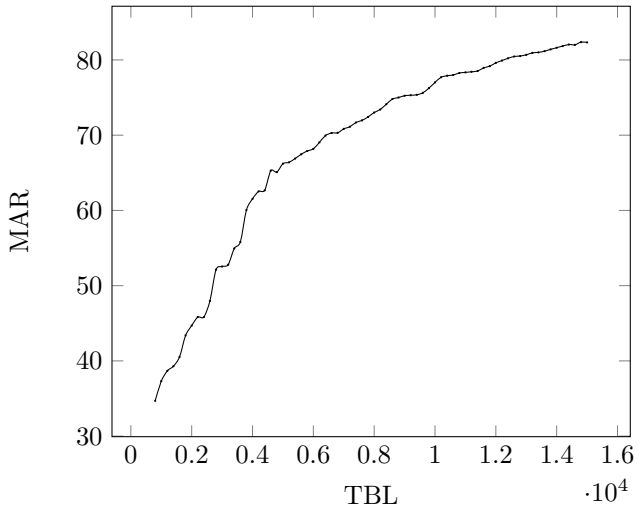


Figure 6.4: TBL vs. MAR - capped message size of $\frac{TBL}{4}$

Figure 6.3 shows MAR vs. TBL again, but with MAR penalized by TBL. The longer the block length, the greater the penalty. There is a contradiction between high Message Acceptance Rate and the Total Block Length. We can't have one without the other, but we would like to minimize TBL to save bandwidth and storage. We observe that highest marginal MAR is at TBL 1000. Here we receive the highest increase in MAR pr TBL increment. TBL 1000 yields a MAR of $44,8\%$. If low TBL is more important than MAR, then this could be an alternative TBL.

Instead of fixing the message size to 140 and 200, we have also experimented with a slightly more dynamic setting: $MaxMessageLength = \frac{TBL}{4}$. Results for MAR are seen in Figure 6.4. A TBL of 4400 would yield a MAR of 62.7%, but this is considering messages of any length, fitted into a TBL with messages of maximum 1100 characters. In other words: If users would like to input any message length, and we would like to add at least 3 quarters noise, for security, they would be 62.7% successfully embedded, given a TBL of 4400. A TBL of 15000 yields 82.33% success, we did not increase TBL enough to reach above 90%. Appendix E.2 shows the entire result set of our tests.

Choosing an optimal max message length, and Total Block Length is difficult, primarily due to the lack of constraints on what optimal means. For StegoBlock, we decided that it was best to allow at least 90% of all messages of maximum 200 characters. TBL must then be set to 4400. This seems like a good tradeoff between MAR and TBL, and should allow a reasonable amount of expressiveness in within the message length. The final size of all StegoBlock headers, with an encoded message of TBL set to 4400 is just around 8 KB, which is considered reasonable.

6.5 Integrity

StegoBlock does not do any integrity checks. In a steganographic setting where information is truly hidden, such a check isn't necessary. We are however not hiding information entirely, we store it in a designated header. Should a StegoBlock be tampered with, it will show up as nonsense. It is easy for an adversary to destroy any StegoBlock message. Techniques include:

1. Delete the email header. Either X-StegoBlock or X-SBDate. Will be easily detected, if the recipient expects some sender to use StegoBlock.
2. Insert or delete an item in the block. The order will be distorted, extraction will be nonsense. This is easily detected, as the extraction algorithm

will identify that the block does not begin with $MessageLength$.

3. Manipulate the block. For instance, it requires no knowledge of the key to change all digits to zeros. This would also change $MessageLength$ to 000 and StegoBlock would assume no message. All zeros would easily be detected, other, more sophisticated tampering, might not be.

An integrity check could easily be implemented. After creating the block and setting the seed, we could run a HMAC function on both header values and the stego-key. The function would be: $hmac(X-StegoBlock+X-SBDate, stego-key)$. The resulting hash value can only be calculated if the stego-key is known. The hash could then be sent in a third header. The recipient could automatically recalculate and verify the hash value. If an adversary manipulated any of the three headers, it would be easily detected - as the hash would no longer match. A SHA-256 digest could be used to calculate a MAC as well: $sha256(X - StegoBlock + X - SBDate + stego - key)$.

But we must be aware that an adversary may then do the same calculation. If he is unable to reach the same MAC value, he will know the disclosed key is fake. Should a block have no message embedded, one will be chosen at random. It is then critical that the stego-key is also chosen at random, such that the user will never be able to tell the real key. An empty block cannot have its integrity validated.

6.6 Permutations and randomness

To randomize the StegoBlock, we chose to implement a Knuth-Fisher-Yates shuffle algorithm as seen in Figure 6.1. As described earlier, this algorithm, if implemented correctly, provably returns a permutation of the set of all possible permutations. For our chosen TBL, this means in the set of all 4400! permutations, an extremely large range. Our shuffle algorithm choice is solid, under the assumption that our PRNG is cryptographically secure.

```
1  function shuffle(prng, arr) {
2    for (let i = arr.length - 1; i > 0; i--) {
3      let j = this.getRandomInRange(prng, 0, i);
4      let temp = arr[i];
5      arr[i] = arr[j];
6      arr[j] = temp;
7    }
8    return arr;
9  }
10
11 function getRandomInRange(prng, min, max) {
```

```
12    min = Math.ceil(min);
13    max = Math.floor(max);
14    return Math.floor(prng() * (max - min + 1)) + min;
15  }
```

Listing 6.1: Implemented Knuth-Fisher-Yates shuffle

We stress that the PRNG we use has not been vetted to be cryptographically secure.

For what we know, it might be - but it has not been checked. This thesis does not aim to prove the security of PRNG's, but we will as a minimum justify that our choice is not completely insecure. Our PRNG is based on the truly random output of RandomSource.getRandomValues() in the Mozilla crypto API.

"The RandomSource.getRandomValues() method lets you get cryptographically random values. The array given as the parameter is filled with random numbers (random in its cryptographic meaning)."[2]

Figure 6.5 shows the results of shuffling the 4 character string $'abcd'$ 600.000 times, with our chosen PRNG and shuffle implementation. Again, this is no proof of security, but we can at least see that there are no obvious bias to a specific permutation. Results show that all possible permutations appear about evenly. There is no way to prove this is actually random, but each permutation should theoretically appear with equal probability.

Our total StegoBlock length is chosen to be 4400 characters. That means 4400! or $3.287612799E + 14122$ possible permutations. There are so many possibilities, that it is impossible to test in any way if we output permutations evenly distributed between all possible permutations. But most importantly, it will also be impossible for an adversary to tell the difference.

6.7 Adversary advantages

We remember the different steganalysis types from earlier (cf. §2.1.4): Stego only, known cover, known message, chosen stego, chosen message and known stego. Since our algorithm is publicly available, and since everyone can execute it, the adversary may effectively try all options.

[2] API RandomSource.getRandomValues(): `https://developer.mozilla.org/en-US/docs/Web/API/RandomSource/getRandomValues`

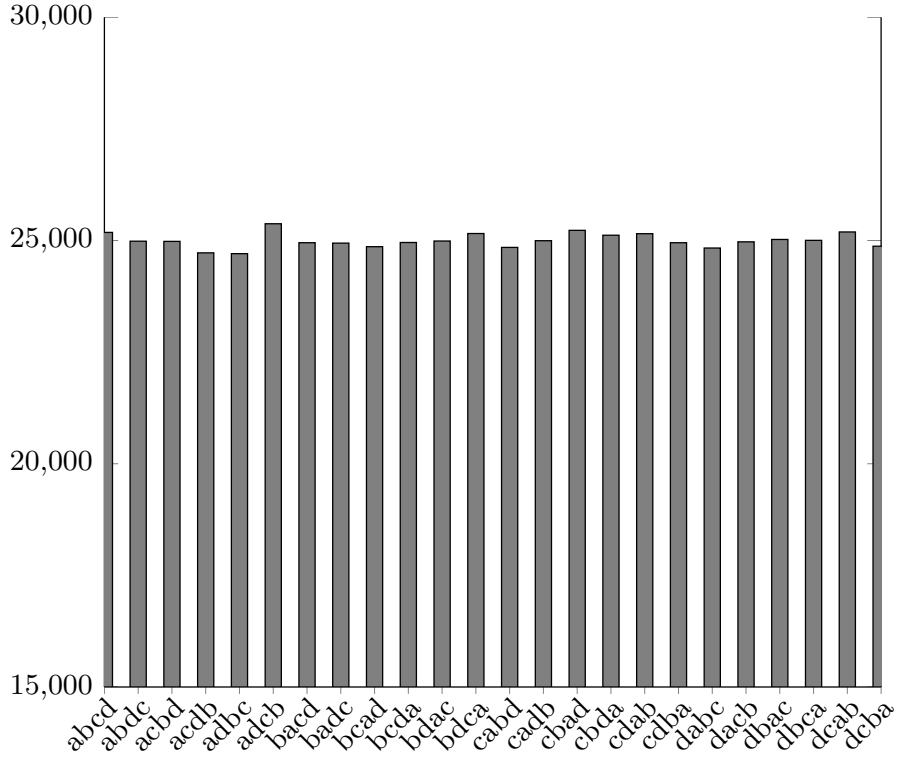


Figure 6.5: Occurrences of shuffle permutations. 4 chars - $6E + 6$ iterations

Our scheme does not require users to input some cover-object, but one could think of the generated noise characters as a cover. As we move from stego only to known stego, we make it increasingly easy to find flaws. The more information we allow the attacker, the more sophisticated his attack vectors become.

In the blind, stego only setting, the adversary has extremely limited chances of success. The block has 4400! different permutations. The block always consists of the same letters, since the target letter distribution will be the same.

Should the adversary for some reason learn which part of the block is cover and which is message, it will become much easier to break the algorithm. The message is still permuted, but in the worst case - he will have to try "only" 200! different permutations to reveal the real message. Extracting the noise would however be difficult, as one would need the internal state of the PRNG, and then it would be simpler to just extract the message itself.

The adversary can operate in the known message setting, if for some reason he acquires a raw StegoBlock email, and learns the message - for instance by looking over his victims shoulder when reading. From knowing these two, he should not be able to tell anything about the next StegoBlock enabled emails from the same sender. The password, or part of it, should not leak in any way. We effectively prevent against this, by appending a seed.

To our application, there is little difference between the chosen stego attack, chosen message and known stego attack. The adversary may install StegoBlock and begin examine algorithms, chose messages and generate stego-objects at will. We argue that our encoding method is solid and that he will have no noteworthy advantages. If our chosen CSPRNG is in fact secure, and we implemented our shuffle algorithm correctly - StegoBlock should be safe against even known stego attacks.

While we are obviously unable to highlight any ourselves, our own best bet on a flaw in the system, is examine the PRNG and in some way learn its internal state from when it executed the permutation. We explicitly stated our trust in our PRNG choice, and that we did not formally examine it. We merely reasoned that its security is plausible, and so this could be the place to look for flaws.

6.8 Summary

We successfully implemented a solution for communicating confidentially over the Internet. We devised a private key steganographic scheme, with encoding

and decoding algorithms. Given some stego-key and seed, we may encode and later decode some message of at most 200 ASCII characters.

Based on a large email corpus, we calculated a frequency alphabet, optimized for english language. Based on this alphabet, we decided on max message lengths of 200 characters and a total block length of 4400. This allow us to successfully encode ~90% of the messages we sampled. We expect this to be a good tradeoff between MAR and TBL.

Different possible attacks on block integrity was presented. Because we are not hiding the block entirely, but storing it in a header, we must pay attention to block integrity. StegoBlock does not implement any check, but it has been thoroughly explained how one should be implemented. A simple SHA-256 hash of the block, key and seed will do as a MAC, and can be sent a long in a separate header. It is critical to ensure plausible deniability that block with randomly chosen messages are also encoded with random keys.

We did not use a formally proven CSPRNG. One was not available, and we consider it outside the scope of this thesis, to prove one. We assume the one we use is secure, but first established that doing so is fair. By shuffling a deck of 4 values 600.000 times, all different results should appear with equal probability. While it is impossible to validate it that was actually the case, a small test hinted in that direction.

Based on the tests we performed, and our analysis of our encoding function, we believe that our solution is in fact secure. We also believe we met our goals of user friendliness with our interface. It requires little to none getting started with StegoBlock, as the UI comes with build in help. Users can keep communicating with their existing contacts, the ones that like can install StegoBlock and gain an extra, subliminal communication channel in their emails.

Chapter 7

Conclusion

We have learned that online communication is under heavy pressure, in the form of eavesdropping and possible regulations. Intelligence agencies pose a large threat, as they have resources necessary for wiretapping backbones of the Internet. We have learned how they can build massive data centers for storing the vast data amount, form these taps. Collected data is analyzed, structured, categorized and made available for later search. From Edward Snowden's leaks we have seen screenshots of the NSA application X-Keyscore that allows analysts to pick up information on people, with simple queries like name or email address. It is most plausible that other major intelligence agencies possess similar tools. Furthermore we learned that analysts are trained for looking after people displaying suspicious behavior, for example in the form of using encryption. We also notice how Turkey and Pakistan impose bans on several applications, using strong encryption to secure the communication between parties. We saw a clear need for an alternative confidentiality preserving communication platform, working around the use of encryption. Surely these type of applications will also be used by terrorists, preventing intelligence agencies to eavesdrop. But we have seen how these agencies monitors not only targeted suspects - but entire countries. Banning applications allowing secure communication will not remove terrorists, it will instead make them find other ways. We would also like freedom fighters to communicate securely within repressive governments, and to quote Gerald Seymour: *One man's terrorist is another man's freedom fighter.*

To solve the problem, we went with a steganographic approach, as this and cryptography are the major research topics for ensuring confidentiality. Like pedantic article by Ron rivest, Chaffing and winnowing, steganography allows to work around encryption, to solve the same goals. We had 3 overall goals:

1. Offer message confidentiality, integrity and availability, even against a strong adversary limited only by cryptography.
2. Provide users plausible deniability to any message from the system.
3. No encryption allowed.

Confidentiality, integrity and availability are key components in online communication, often referred to as the CIA triad. Since they are most often achieved by cryptography and encryption, we first explored which cryptographic components we could not do without, and which we could omit in favor of steganography. Encryption schemes were instantly ruled out, but simple hash functions would take us a long way.

Plausible deniability, referring to the condition, where a person can, plausibly and legally, deny any knowledge of, or association with some specific reality, in a way that they have deliberately provided beforehand was thought in from the beginning. People could easily be threatened into giving up their keys to secure communication, users would not be secure enough. We needed to make sure that no adversary could verify if a some key is valid or fake, allowing the suspected user to plausibly provide any key.

To meet each goal, we ended up with designing and implementing StegoBlock, an extension for email client Thunderbird. Building on top of traditional emails, let us utilize an already existing and widespread communication medium. Users should have a low entry barrier, as interface and usage would change minimally from what they already know and use. By extending emails in "X-"-headers, we created a solution that still works for email clients not StegoBlock enabled. These clients will simply ignore said headers, only with StegoBlock installed, additional processing occurs.

StegoBlock implements the idea of embedding a small message in a larger, randomly generated cover object, that is then permuted, only reversible with knowledge of some stego-key. In detail, we would generate what we called noise, and append it to the secret message and its length. We would then do a Knuth-shuffle on the entire block, but with a CSPRNG seeded with the stego-key and a seed. Only by knowing the key, one could reverse the permutation and extract the secret message. Reversal without key is proved extremely hard, as the

number of possible permutations equals the factorial of the total block length. We wrote our own implementation of the simple Knuth-shuffle along with a reverse-Knuth-shuffle as well. Provided with a RNG in the same state, the algorithms can successfully encode and decode a message. We did not use a formally validated CSPRNG, but settled for assuming one. We did however carefully examine which properties are necessary for a CSPRNG and check for obvious bias in our chosen third party implementation.

To accommodate plausible deniability, we made sure that every email sent with StegoBlock installed, would embed some message. If users do not write one, a randomly generated one will be used instead. There will always be a block in emails from a StegoBlock user, but the user may always argue that no secondary message was added. There is much value in plausible deniability. If users can convince the adversary that they are not hiding anything in a particular message, the adversary may not even bother decoding the message. With traditional crypto systems, it is also possible to keep messages secure, but crude criminals may force key disclosure with threats or violence. StegoBlock prevents against these unfortunate cases.

We designed StegoBlock to have a storage for stego-keys and to have a simple user interface. The design was reasoned thoroughly about in our steganalysis. We established how a message length limit is a necessity, but reasoned thoroughly about said limit, and the total block length. By analyzing a large email corpus, we established a foundation on real, human written emails, for evaluating how successful embedding message would be. We ensured that our method of scrambling the block would permute it in such a way, that the outcome would be one in every possible permutation - and in general we sought to remove every possible shortcut to extracting the secret message. In particular, we made an effort to block the usage of statistical analysis to reveal patterns in the block, by ensuring that all blocks follow the same target distribution of letters. All blocks consist of the same amount of the same characters, we use the same FREQUENCY_ALPHABET for all blocks and we can embed roughly 90% of all <= 200 character english messages.

Our steganalysis showed that the recommended settings for block and message length, results in a very hard reversal process, if the stego-key is unknown. We iterated possible attacks on block integrity, but which would still not affect message security in the form of enable message decoding. In particular, we saw how an adversary might alter an encoded block to look like no message was embedded. This attack and the alike, were shown preventable by simple integrity checks. Implementation was described and made easy for further development. Our shuffle algorithm and its promise of results in the entire range of all possible permutations are formally proved in existing work. We briefly validated that along with our chosen random number generator, it did not have any obvious

bias. The security claim we gave of keeping even an adversary of Dolev-Yao type strength at the gates is met, to the best of our knowledge.

With StegoBlock and the theory and steganalysis it is based on, we consider it proven that it can indeed be used to hide users secondary communication. We strongly believe that users can display plausible deniability, especially if they agree on a fake stego-key, besides the real one. Should the adversary interrogate both communicating parties and they both reveal the same fake stego-key, their denial is even more plausible, as the adversary arrives at the same decoding result. Our solution should allow everyone to communicate in private, legally, even if encryption is considered illegal. Users may even plausibly deny that they communicated in private.

7.1 Future work

Probably the most obvious area for improvement in the current application, is the stego-key store. This is currently not secured good enough. Today, its security is based solely on the security mechanisms of the operating system. A new version of StegoBlock must surely remedy this situation. Should an adversary gain access to the current key store, he will obviously learn all stego-keys and be able to decode all blocks formed by the user in question.

Implementing an integrity check is top priority as well. We already iterated possible attacks that can all be remedied by calculating a simple hash and validating it. We could use a SHA-256 digest of header values `XStegoBlock` + `XSBDate` + stego-key and store it in a new header, `X-SBIntegrity`. The receiver could then easily recalculate the digest, and discard blocks mismatching integrity values. Providing a MAC may however ruin plausible deniability. Should an adversary compel users to give up their fake keys, he can easily tell if keys are fake, by doing the same computation. We should in all aspects ensure that an empty block looks and behaves the same way, as a filled block with a fake key. A possibility could be to encode randomly chosen messages with randomly chosen keys - which our implementation already does.

StegoBlock currently has limited functionality for multiple recipients. An email with a StegoBlock can be sent to as many recipients as needed, but the block will be the same for all recipients. This means all recipients must share a key for all to decode the message. We imagine a future version to better support multiple recipients. One possibility could be to allow multiple keys for decoding messages from the same sender. We must remember that an adversary would expect all recipients to disclose keys decoding a block to the same message. This

might require the entire recipient group to agree on the same fake key, unless a more clever scheme can be discovered.

Future work may also be, to formally verify if our PRNG choice is in fact cryptographically secure. The workload for such an exercise could amount to a thesis in itself. It is however critical to the system, because of the hard dependance on the cryptographic primitive.

We would like to offer a dynamic FREQUENCY_ALPHABET. In a future StegoBlock version, users should be able to pick their own target character distribution. This would allow them to blend their StegoBlock into the distribution of their native language. A letter frequency analysis of the StegoBlock would return the same distribution as the message itself. We would also have the added benefit of allowing the characters of the users native language. The initial version only allows the default ASCII character set. Following the though stream of allowing dynamic distribution targets, we could imagine users being able to select books, emails, newspapers, tweets, basically any text on their own - then StegoBlock would analyze the character frequency of those inputs - and adjust the FREQUENCY_ALPHABET accordingly.

Lastly we also see a future version to have some form of logging mechanism. Currently errors will be suppressed, unless if happening when sending emails - where they will not be logged either. Debugging is near impossible, if not reproducible in a development environment. A full fledged logging mechanism, with option notifying developers would be a great addition.

Appendix A

Header example

```
Return-Path: <toftegaard.andreas@gmail.com>
Received: from MacBook-Pro-2.local (x1-6-a0-63-91-fe-bf-82.cpe.webspeed.dk.
    [2.104.2.59])
        by smtp.gmail.com with ESMTPSA id 89sm1327821lja.16.2016.11.10.11.05.34
        for <toftegaard.andreas@gmail.com>
        (version=TLS1_2 cipher=ECDHE-RSA-AES128-GCM-SHA256 bits=128/128);
        Thu, 10 Nov 2016 11:05:35 -0800 (PST)
From: Andreas Toftegaard <toftegaard.andreas@gmail.com>
Subject: StegoBlock
To: Andreas Toftegaard <toftegaard.andreas@gmail.com>
X-Stegoblock: a.%20MttOvon%20iyEncoc.%20to0ryinrrupeos%3EltEsntittlnt%20.%20o
 %20emaathzhirUr%2Fnsciai%20pOaa%20tpi%20ob1pewlelvayIgotht%20loh1dsCrrla%20o
 aeagd._%20r%20%20yahr%20c%20%20%3Af%20%20m%20teroesecnEc%20vLbssodrDo%20%201
 %20%20%20esaween%20aho%20mlikdMnl%20tl%20bmoeo%25%20ttoilecgn%20mhesseg%20do
 0aFP%2Cw%20iwiEyeersn5%20lldaebHmms)dsAei%20%40e%20a%20hv%20%3E%20sssn%2F%20
 D%20rein%20a%20ae%20eioEiFicT%20ieuaFetriyh%20d%20%20%3Ev9e%20n%20iiv%20hPsT
 nsaso%20tae%20en6saIconfdtahTohh%20tmaudQar%26c%20sye%20ueCkotHt%20xTod0im%2
 0sert%20%3F%20ittlnkek%20oLlrNjd%3E%20eo%20afi0tu%20plithoetCN%20%20s%20%20e
 bcctyowe.wr%20oo%20%20R%40r%3D_eniesepa%20hml%20i%20hmttinal%20dasoe%20ho4i%
 20idc-%20%3C%20e%2Ca%20Ysheon(%3Diywn%20%2Fr0%20%20gu%40sm8t%20sM%20kCatvt%2
 0a0a%20ilPtaNtif%3DwegE%20InesMoi%20Aab%3Dhz%20teli_hDr%20mTea1%20ccol.e%20I
 %20ok%2C%20m.%20s%20t%202nftht%20%20Tb%20%20%20%20fstaeq%20a%20at'ia%20na%3A
 snalIltg%20%2CofsgemlapfreEpozmEneibmhee%20-fcuoreepChlj%3AueeNSnreobrb%20%2
 Bhkm%20i20uCnp%20%20rdNg%20tLcrei%20obroanrwywfoo%20skohcsi%20Nr%2C%5Bydi%3C
 _ye0%20dsnnit0a3revuP%20iebrre%20%3Diehamn%20auaal%20%20%20e%200%20sf%20ntnt
 ne%3AkOenrmaoeru%20-se%20pnoSo%20%20%20yoEnh%20%20ct%20%20Go%203aKnn%20%3Cpd
 paoe%20%20%20%2Cmxiane%3Daasapup%2CafeJKele9n%20%20et7aleatt%20emeAecn%20txe
 ieery.shsmm%20u%2Fv%20s%2Ca%20i%20ioDeHssmaest%20oe-lier%20r%20o)an%20%20net
 Ma%09uoCsylae%20r%20Ou%20ea.u%20%20lvsLare%3Fo4nheoeegtmwi%20niT%20te%20hlM1
 npea%20t%20nryi%20i.sTa%20ruiurTe%20sn%09rf%20nrlObsz*%20sd)aenrnTcit%20%20i
 .%3Dnlpeaoe7BusaooE1uio%20s%20oaaFs1d%2Fain6%3Dpaefnmeias%20eho%20-ethtaaLak
 nn%20tchoeidslr%20%20d%3Arurcath%20ftinurs%20%20%22%201%20lopotUimtRar%20%20
 cd%2Fldf-rtotonhiSunkOieuuraegfsghae%2Canel%20ea%2CynZtEoosens%20i%20ffdp%20
 %20doot%2Cuidr%20etMuftl%20%20-toei%20%20At%2Cenid%20so%20stono%24.%20wmib%2
```

```
 0dnnt%2FRlo%20pee%20%40e.AhneMeau%20whe%20ohn%40ato0haef%20%20%20o%20sldceif
 ~h%20dwosr5%20ep0c.sntCoewuoetp.lirh%20oeaeg%20u%20tr%20u%20Rd%2Cl%20sOkirei
 %20BrluYd%20%20a%2B%201eltc%20lsoRa%20orasmto%20hiefdr%20(_vetom%20yaa8eeaIm
 wiiDer_ilre%20f%20c%20eishr0ivleGenbB%2Ft%207plEcnt%09osaite%20heAm%20er-cse
 anrih4ng2ra%20Do-aya%2Ct%20%20ile%20lwhDr.n%20Sedtn%3Eed%20%20h%20enm%2C4iid
 p%3ErmeJnwblf%20%20ehHoo7-anashnnineh%200ht%20%20trthdWuiprmr%20isWedn-gdr%2
 0nfpmeCefunrd%20aiceee(c%207nooaaoacSir_%20%20be0Clsybr%20aa%20knrs%20EosMe%
 20edlhws%3E.hdhwiito%20p%20e%3Etmrrdat2d%20E%20di%20upenm%20%20s%2C5%2Cbcriu
 %209unfi%20saeed%20%20ocP.%20.p%20deiss%20n%20mipnsat%20sctag%20etpc%20oh%09
 aeEni%20%20tEaIie%20%20eebdk%2Cornis%3Dl%20%20eeg%3AseDth%20s%20%3DtJrnhg%20
 lroPCr%20i'lyidVee%20%20%20cGihraohiaintl%20sganu%2C0%20t%20hnnt%20%20tpiSeI
 %20%20%20webp%20%20art%2C0%20dedt.%20e%20hrcCw%20Fv%20%20nue%20an%20%20sa%20
 %20ovt2nusoar%20Cc1tint-Gs2e%20egenbpr2n_inp6rhtdeog%20v%20P%20yetaPnistd0eo
 rhMa-TMImnnmOir'g%20na%20wvt%20'm%3F%20%2F%20%20Gaai1etoopntlnt%20ath%20icho
 %20o%20R2incbccrf%2F_%20s%206%20%20Tu%3Auaio%3DeiC%20t%20%20q%20pDkoi%3Dfi_u
 aurorErvoMruiAniynfIb%20mnnoegmau%20%26tn%20sle%20t%3DC.pe%20ht.ne%20H%20v%2
 0Mc%20otalarocsettsmaeEs1rmto%20uiordykt)s%20%20ado%20%20FeeuSennTcdayythne%
 20nlsidiataoXtska%09sorec.y%20Nah%20osrWr%2Fe%203edeaecoC%20mootiaiei%20Dnrr
 Reale!%20r%20yioa%20%20'irheomneo2ili%20ath%20vtwtdo%20Jb%20sd-rs%20e.t'rn%2
 0gtem%20%20%3Diwb%20yocw3%20pnoed%20t%20eihadleh%20W%2Cs%20nrtiUyl%20a5%20o%
 20o%20ttnHt2il%3ANn2%20os%20ggee.e%20tadsag%09%20stNla%20n%20oqrredTnTe%20nd
 KaeT%20etdesenbfp%20oh0co%20Pe9cn%20%20%20pp4e.lat%20M%20%20%20ittnT%20e%20%
 20%20rs%20W%20hs6%20-%20Sue%20%3Dr%20e.e%20frrAg%20ce%20%2CGt%20mt%20ot%20at
 ShAne%2Fot%20yfoertau8noir%209%20h3a5tyatalV%20m%20rp%20eutsid%2CAc%20p2Ui%2
 0drdsr%20to.%20u%20%20%2Csogdey%20NeOfrofeahjC%20%20r%20g%20t%20nneBn2ofeht%
 3FLaiRrn%23erlsribmng%3Coesn%202tso.rn0tiTrnO%20Paprnvc%20sai%20-tg%2Cee%20s
 %20ctB%20sp%20m%20s8%20%20%20ih3tiE1theh%20rl2sl1n-'ps%20ss%20bln_2%20n%2C%2
 0%2Cn'brmasi(lxicc%20rg%20etoNhop%20heis%20h%20T%20leoac%20Ia%20%09twneoeeyI
 ogsrbegtw%20ced%20%20la%20lss-teli%3Dsabow%20xo%2CuRarde.t%20%3D%20%40%20nh%
 20meor%20%20%20oduwmiSsoiw%20en.na%3D%20hhnti%20%20ho0%20%2ClA%20nsnartEnoo%
 20%3Dn%20n%20r%20n1%3D0tyeowo.oaln%2FmSsdswAo%3A0mmoafefU%20%20%20ln%20eeo(W
 srDimdcnelsa%20e%20_sp%20ior%20.ithIoleoacaeusPdhli%20uS%20At%2Chr%20%2Frven
 o0%209e%20%09%20%20oTwParon%20%20X%20.fi%09a%20huvo%20uDnh%20h0%20s%201%20aC
 Rc%20%20snkina%20m%20wL%3E0%7CinrtlqnioD%20orSae%20leNmoEdvleao%20nekofo%20U
 gHmoy%20le%20tpa%20nlopen%20ssdneiS0cnipye%20%3EhteoTbeu%20rtdhAt1%20capgski
 -%402it%20da%20r%20enso%20f%20ee.t5ht%200ohtu%3Fcd%20ve%2F%20t%20-esr%20nate
 enebS%20oa%20oi7tE%20umtr%3Dds%20ec9waearR%3Aela%20u%20n%20%20*i%201%20ui%3E
 yaa%20dK%20iOIil%20c%20%20lg2ye%3Ba%3DlICo4tdnirs1witglveh%2F%2Cci%20H%22ctn
 a*s%20n%20h0leettt%20hseh3ewc2xfde%20gc%20mvotno%20orreaea_gW%20eteorf-ultty
 %20%2F%20wnhm%20aShoia%3D%20cua4n%20dA%20lrey.kgpn.schd%20%2F3n%20oourl%20%2
 CBm%20g%20aS%20Wh%3At%20p%2CaFeEtrogtLysdyt%20uo%20niy9ato%20tc.r%20%20pasno
 aeeca%20Set%20afheuieluipoan%20o%20ow%2CS%20rat6itgo%20no%20%20b%20%20i2L%20
 kh%20eyOh%20tnk.%20toebsEod%20%20%20ii%20gtuSita%20%20ru%20d%20erGc%20aelyll
 nolpn%20Fwh5%20.-rna-%20%20ont%3FolSs%20%3DdvfitEttnruar%20vTere%20s%3B)eeoh
 ejnotlI'uo%20%20tt%20oraremwttr%20ya%20Rb%20%20%20Ep%20%20facput%20%20wgBh%2
 0oie%24%20esf%20%2F%20vis%20tycs%20ei%2FruyncBsoe%20cic%20ae.eolH3weeica%3De
 %20.%20.e%20s7%3EfrirIpDi%20.%20eegamu'-dleraodaeo0Pro%20dc%20Niaoveu%20%20e
 eise2%201tyo%2CcfeaT%20fe%201CTeut%20s!uE%20a%20rr%20nc%40tb%5D%20xoFf8%20st
 t%20%20RJ%20%20nnllnVr%20rtrcnaeiasiilnA%20o%20t%20seln%20gt%201rcSrcis%20us
 otrmaeh%20%20iarVry%261%20p%20se%20te%2FtsceerwskEmpi%20kcfoe%20seealrfIlh%2
 01tl%20ee%20B%20gdoodnrst_nn%20ieelsntsl-%20ninenigt0%20t3%20nexetn%20vrai2%
 20.dentgdstfr%20gdE1yNo%20eC%20C%3Flttlt%3A%3D1A%20o%20nor%20eip%20c%20pmrua
 m3s.yIdieethfetf%20osawui.o3%20g8%2Ccaeeoie%20.gta%20egiaet0_%3D%20aocc%20%2
 0t%20OtGfed%20Osovet-%20)%20ryco%20%2C%20%20a%20sietsy0%20ueniioani%20w%20ni
 chiy%20mtsc5%20r%20f%20des%20trdrn%5Cims%20%20%20Cto.%3BMost%20hoes-en%20%20
 nto_%20M0ntYrtynNyPwl.%20gbdse%20%20wase%20%202-%20%200iJae0%20%20ygawmT-0tt
 tr%20-Eatbu%3Doatta1AsurlAiruccmrdi%20vkri%201aa%20io-r%20s%3Fb
X-Sbdate: Thu Nov 10 2016 20:05:34 GMT+0100 (CET)
Message-ID: <d496c8e5-8d87-fc4c-86e8-e2d8fcaad665@gmail.com>
Date: Thu, 10 Nov 2016 20:05:34 +0100
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:45.0)
 Gecko/20100101 Thunderbird/45.4.0
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8; format=flowed
Content-Transfer-Encoding: 7bit
```

```
99
100 Nothing to see here
```

Listing A.1: Source code of email with StegoBlock

Appendix B

Installation

For development, unzip and place the provided source in the Thunderbird extensions folder:

Windows %APPDATA%/Thunderbird/Profiles/<Profile Name>/extensions

Linux ~/.thunderbird/<Profile Name>/extensions/

MAC ~/Library/Thunderbird/Profiles/<Profile Name>/extensions/

StegoBlock can then be selected and installed from the Extensions menu within Thunderbird.

For simple end user installation, use the Extensions menu within Thunderbird to select the stegoblock.xpi file directly. Go to the Extensions tab, click the gears icon, then select "Install extension from a file...".

StegoBlock has been tested on Thunderbird 45.4.0, running on a Macbook Pro with OS X El Capitan (10.11.6).

Appendix C

StegoBlock extension files

```
<?xml version="1.0"?>

<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:em="
    http://www.mozilla.org/2004/em-rdf#">

  <Description about="urn:mozilla:install-manifest">
    <em:id>stegoblock@toftegaard.it</em:id>
    <em:name>Stego Block</em:name>
    <em:description>Encrypted secondary messages in emails.</em:
        description>

<!--The Stego Block extension allows for embedding small secondary
    and encrypted messages in email headers, known as a Stego Block
    . If no message is added, a randomly generated on will be
    inserted. Adversaries cannot destinguish between Stego Blocks
    containing real messages and ones containing random tekst. This
     provides the sender with plausible deniability of
    communicating in secret with any recipient.

Stego Blocks are encrypted with 256 bit AES.-->

    <em:version>1.0.1</em:version>
    <em:creator>Andreas Toftegaard</em:creator>
    <em:optionsURL>chrome://stegoblock/content/options.xul</em:
        optionsURL>
    <em:iconURL>chrome://stegoblock/content/icon.png</em:iconURL>

    <em:targetApplication>
```

```
      <Description>
        <em:id>{3550f703-e582-4d05-9a08-453d09bdfdc6}</em:id>
        <em:minVersion>1.5.0.*</em:minVersion>
        <em:maxVersion>51.0</em:maxVersion>
        <em:type>2</em:type>
      </Description>
    </em:targetApplication>

  </Description>
</RDF>
```

Listing C.1: StegoBlock extension install.rdf file

```
content stegoblock  chrome/content/
overlay chrome://messenger/content/messenger.xul  chrome://
    stegoblock/content/messenger.xul
overlay chrome://messenger/content/messengercompose/
    messengercompose.xul  chrome://stegoblock/content/
    messengercompose.xul
```

Listing C.2: StegoBlock extension chrome.manifest file

```
1  var SBCommon = function () {
2
3    // gets the StegoBlock extension preferences.
4    var initPreferences = function (obj) {
5
6      if (obj.prefs)
7        return;
8
9      obj.prefs = Components.classes['@mozilla.org/preferences-
           service;1']
10       .getService(Components.interfaces.nsIPrefService)
11       .getBranch('stegoblock.');
12   };
13
14   return {
15
16     // shortcut for the extensions preferences
17     prefs: null,
18
19     // stores callbacks for the preference observer
20     observeCallbacks: {},
21
22     // convenience utilities
23     utils: {
24
25       // native JS implementation for extending objects. somewhat
             similar to jQuery.extend()
26       extend: function extend () {
27
28         if (typeof(arguments[0]) === undefiend)
29           arguments[0] = {};
30
31         for (let i = 1; i < arguments.length; i++) {
```

```
32            for (let key in arguments[i]) {
33              if (arguments[i].hasOwnProperty(key))
34                arguments[0][key] = arguments[i][key];
35            }
36          }
37
38          return arguments[0];
39        }
40      },
41
42      // register a callback that gets fired when preferences change
43      observeCharPreferences: function (id, callback) {
44
45        if (this.prefs === null) {
46
47          let that = this;
48          let observingObject = {
49
50            observe: function (subject, topic, data) {
51
52              if (topic !== 'nsPref:changed')
53                return;
54
55              for (let callbackId in that.observeCallbacks)
56                that.observeCallbacks[callbackId](JSON.parse(that.
                      prefs.getCharPref(data)));
57            }
58          };
59
60          initPreferences(this);
61
62          this.prefs.QueryInterface(Components.interfaces.
                nsIPrefBranch2);
63          this.prefs.addObserver('', observingObject, false);
64        }
65
66        if (this.observeCallbacks[id] === undefined)
67          this.observeCallbacks[id] = callback;
68      },
69
70      // get specific char preference as an object
71      getCharPref: function (key) {
72
73        initPreferences(this);
74
75        return JSON.parse(this.prefs.getCharPref(key));
76      },
77
78      // set specific char preference with an object. object gets
            stored serialied.
79      setCharPref: function (key, value) {
80
81        initPreferences(this);
82
83        this.prefs.setCharPref(key, JSON.stringify(value));
```

```
84      },
85
86      // unregister a previously registered callback for preference
            change
87      forget: function (id) {
88
89        delete this.observeCallbacks[id];
90      }
91    };
92  };
93
94  // extend the global variable with common functionality, for easy
        access
95  SBCommon.utils.extend(window.SBCommon, SBCommon());
```

Listing C.3: StegoBlock extension common.js file

```
1   <?xml version="1.0"?>
2   <overlay id="stegoblockMessenger" xmlns="http://www.mozilla.org/
        keymaster/gatekeeper/there.is.only.xul">
3
4     <script type="application/javascript" src="chrome://stegoblock/
          content/seedrandom.js"/>
5     <script type="application/javascript" src="chrome://stegoblock/
          content/common.js"/>
6     <script type="application/javascript" src="chrome://stegoblock/
          content/steganography.js"/>
7     <script type="application/javascript" src="chrome://stegoblock/
          content/messenger.js"/>
8
9     <vbox id="expandedHeadersBox">
10      <hbox id="stegoblock-content-box" collapsed="true">
11
12        <grid flex="1" id="stegoblock-grid">
13          <columns id="stegoblock-columns">
14            <column id="stegoblock-header-column" minwidth="0"></
                  column>
15            <column id="stegoblock-content-column" flex="1"></column>
16          </columns>
17
18          <rows id="stegoblock-rows">
19            <row>
20              <label id="stegoblock-header" value="Stego Block" class
                    ="headerName"/>
21              <hbox>
22                <description id="stegoblock-content" flex="1"> </
                      description>
23                <vbox id="stegoblock-disabled-box" collapsed="true"
                      flex="1">
24                  <hbox>
25                    <label id="stegoblock-disabled-label" flex="1"
                          style="color: grey; font-style: italic"/>
26                  </hbox>
27                  <hbox id="stegoblock-add-key-box">
```

```
28                    <textbox id="stegoblock-add-key" placeholder="
                          Shared secret key of at least 8 characters.
                          Do not agree on this over email"  flex="1"
                          onkeyup="sb.validateKey()"/>
29                    <button id="stegoblock-add-button" oncommand="sb.
                          addKey()" style="text-align: center" disabled
                          ="true">Add StegoKey</button>
30                  </hbox>
31                </vbox>
32              </hbox>
33            </row>
34          </rows>
35        </grid>
36
37      </hbox>
38    </vbox>
39
40  </overlay>
```

Listing C.4: StegoBlock extension messenger.xul file

```
1   const sbCommon = window.SBCommon();
2   const sbStego = window.SBStego();
3
4   var sb = {
5
6     // storage for the sender of a selected email
7     sender: null,
8
9     // gets an element by id, from the map or DOM, if not already in
          the map
10    elementMap: function (id) {
11
12      if (this.map === undefined)
13        this.map = {};
14
15      if (this.map[id] === undefined)
16        this.map[id] = document.getElementById(id);
17
18      return this.map[id];
19    },
20
21    // add listener messagepane loading
22    startup: function (event) {
23
24      let messagepane = this.elementMap('messagepane');
25      let _this = this;
26
27      messagepane.addEventListener('load', function (event) {
28
29        _this.handleMessageSelection();
30      }, true);
31    },
32
```

```
33    // when a message is selected, headers are checked for a
          StegoBlock.
34    // if one is present, it will be tried shown to the user.
35    handleMessageSelection: function () {
36
37      let enumerator = gFolderDisplay.selectedMessages;
38      let _this = this;
39
40      // iterate over all selected emails
41      for (let msgHdr in fixIterator(enumerator, Ci.nsIMsgDBHdr)) {
42
43        // extract all headers as MIME messages
44        MsgHdrToMimeMessage(msgHdr, null, function (aMsgHdr, aMimeMsg
              ) {
45
46          try {
47
48            // trial and error. first "from" then "to" - ensures that
                   StegoBlocks in
49            // sent mails can also be read. not very elegant, but
                   apparently there is
50            // no way to distinguish if a mail is in a "sent" folder.
51            if (!_this.extractStegoHeader(aMimeMsg.headers.from.
                   toString().trim(), aMimeMsg))
52              _this.extractStegoHeader(aMimeMsg.headers.to.toString()
                     .trim(), aMimeMsg);
53
54          } catch (err) {
55
56          }
57        }, true, { examineEncryptedParts: true });
58      }
59    },
60
61    extractStegoHeader: function (sender, aMimeMsg) {
62
63      let cont = this.elementMap('stegoblock-content');
64      let contentBox = this.elementMap('stegoblock-content-box');
65      let disabledBox = this.elementMap('stegoblock-disabled-box');
66      let disabledLabel = this.elementMap('stegoblock-disabled-label'
            );
67      let prefs = sbCommon.getCharPref('addressesAndKeys');
68      let addressRegEx = /<(.*)>/;
69
70      contentBox.collapsed = true;
71      disabledBox.collapsed = true;
72      cont.collapsed = false;
73      contentBox.collapsed = true;
74      cont.childNodes[0].nodeValue = ''; // hacky way to set value of
             a description node
75
76      // handle "name <email>" format
77      if (sender.indexOf('<') > 0) {
78
79        sender = addressRegEx.exec(sender)[1];
```

```
80        this.sender = sender;
81      }
82
83      // find matching StegoKey for sender
84      let key;
85      for (let i = 0; i < prefs.length; i++) {
86
87        if (prefs[i].addr === sender)
88          key = prefs[i].key;
89      }
90
91      // extract header
92      let ciphertext = aMimeMsg.get('X-StegoBlock').toString();
93      let date = aMimeMsg.get('X-SBDate').toString();
94
95      // remove folding spaces
96      ciphertext = ciphertext.replace(new RegExp(' ', 'g'), '');
97      ciphertext = decodeURIComponent(ciphertext);
98
99      // do not show any StegoBlock UI if email does not contain a
            StegoBlock
100     if (ciphertext.length === 0) {
101
102       contentBox.collapsed = true;
103       return false;
104     }
105
106     // there is a StegoBlock, but no matching StegoKey. show UI for
            adding one.
107     if (key === undefined) {
108
109       contentBox.collapsed = false;
110       disabledBox.collapsed = false;
111       cont.collapsed = true;
112       disabledLabel.value = 'You have no shared StegoKey with ' +
              sender;
113       return false;
114     }
115
116     // show the StegoBlock
117     var plaintext;
118     try {
119
120       plaintext = sbStego.decode(ciphertext, date, key);
121     } catch (e) {
122
123       contentBox.collapsed = false;
124       cont.childNodes[0].nodeValue = e;
125     }
126
127     // strip away any random right padding (if message is less than
            maxMessageLength)
128     //plaintext = plaintext.substring(0, plaintext.lastIndexOf
            ('///'));
129
```

```
130       contentBox.collapsed = false;
131       cont.childNodes[0].nodeValue = plaintext;
132
133       return true;
134     },
135
136     // fired on keyup when trying to add a new StegoKey
137     // validates if the key meets basic requirements, like length
138     validateKey: function () {
139
140       let value = this.elementMap('stegoblock-add-key').value;
141       let button = this.elementMap('stegoblock-add-button');
142
143       if (value === undefined || value.length < 8) {
144
145         button.disabled = true;
146         return;
147       }
148
149       button.disabled = false;
150     },
151
152     // adds a new (valid) StegoKey to the preferences
153     addKey: function () {
154
155       let textbox = this.elementMap('stegoblock-add-key');
156       let key = textbox.value;
157       let prefs = sbCommon.getCharPref('addressesAndKeys');
158
159       prefs.push({ addr: this.sender, key: key });
160
161       sbCommon.setCharPref('addressesAndKeys', prefs);
162       this.handleMessageSelection();
163
164       textbox.value = '';
165     }
166   };
167
168   window.addEventListener('load', function (event) {
169
170     sb.startup(event);
171   }, false);
```

Listing C.5: StegoBlock extension messenger.js file

```
1  <?xml version="1.0"?>
2  <overlay id="stegoblockMessengercompose" xmlns="http://www.mozilla.
       org/keymaster/gatekeeper/there.is.only.xul">
3
4    <script type="application/javascript" src="chrome://stegoblock/
         content/seedrandom.js"/>
5    <script type="application/javascript" src="chrome://stegoblock/
         content/common.js"/>
6    <script type="application/javascript" src="chrome://stegoblock/
         content/steganography.js"/>
```

```
7    <script type="application/javascript" src="chrome://stegoblock/
         content/messengercompose.js"/>
8
9    <vbox id="addresses-box">
10     <hbox id="stegoblock-message-box">
11       <vbox style="width: 11.5em;" pack="end">
12         <label id="stegoblock-message-label" value="Stego Block:"
               control="stegoblock-content" style="text-align: right"/
               >
13         <label id="stegoblock-message-length" flex="1" style="color
               : grey; font-style: italic; text-align: right"/>
14       </vbox>
15
16       <textbox multiline="true" id="stegoblock-textbox" flex="1"
             placeholder="Embed a secondary message"
17         name="stegoblock.message.body" minheight="50" onkeyup="sb.
               ui.setRemainingCharCount()"/>
18
19       <vbox id="stegoblock-disabled-box" collapsed="true" flex="1">
20         <hbox>
21           <label id="stegoblock-disabled-label" flex="1" style="
                 color: grey; font-style: italic"/>
22         </hbox>
23         <hbox id="stegoblock-add-key-box">
24           <textbox id="stegoblock-add-key" placeholder="Shared
                 secret key of at least 8 characters. Do not agree on
                 this over email" flex="1" onkeyup="sb.ui.validateKey
                 ()"/>
25           <button id="stegoblock-add-button" oncommand="sb.ui.
                 addKey()" style="text-align: center">Add StegoKey</
                 button>
26         </hbox>
27       </vbox>
28     </hbox>
29   </vbox>
30
31 </overlay>
```

Listing C.6: StegoBlock extension messengercompose.xul file

```
1  const sbCommon = window.SBCommon();
2  const sbStego = window.SBStego();
3
4  var sb = {
5
6    ui: {
7
8      // service for prompting users
9      promptservice: Components.classes['@mozilla.org/embedcomp/
           prompt-service;1'].getService(Components.interfaces.
           nsIPromptService),
10
11     // maximum StegoBlock message length
12     maxMessageLength: sbStego.maxPlaintextLength,
13
```

```
14      // regexp for extracting textboxes with email recipient
            addresses
15      addressNodeRegEx: /addressCol2/,
16
17      // regexp for extracting email from "name <email>" format
18      addressRegEx: /<(.*)>/,
19
20      // storage for key of recipient
21      key: null,
22
23      // storage for recipient of the message
24      recipient: null,
25
26      // gets an element by id, from the map or DOM, if not already
            in the map
27      elementMap: function (id) {
28
29        if (this.map === undefined)
30          this.map = {};
31
32        if (this.map[id] === undefined)
33          this.map[id] = document.getElementById(id);
34
35        return this.map[id];
36      },
37
38      // fired when compose window is ready
39      NotifyComposeFieldsReady: function () {
40
41        let label = this.elementMap('stegoblock-message-length');
42        document.getElementById('stegoblock-textbox').value = '';
43        label.value = this.maxMessageLength + ' chars left';
44
45        this.elementMap('stegoblock-textbox').addEventListener('
              keydown', this.validateLength, true);
46
47        this.observeRecipientsByPolling();
48      },
49
50      // observes the recipients of an email by polling.
51      observeRecipientsByPolling: function () {
52
53        let _this = this;
54        setInterval(function () {
55
56          let els = document.getElementsByTagName('*'); // get fresh
                collection each iteration
57          let addresses = _this.getRecipients(els);
58
59          if (addresses.length > 1)
60            _this.disable('toomany');
61          else if (addresses[0] !== undefined)
62            _this.validateRecipientAndKey(addresses[0]);
63          else
```

```
64            _this.enable(null); // in case of no recipient, just show
                  the textarea
65
66        }, 500);
67      },
68
69      // extracts recipients (email addresses) from a collection of
            DOM nodes
70      getRecipients: function (elementsCollection) {
71
72        let addresses = [];
73        for (let element in elementsCollection) {
74
75          if (this.addressNodeRegEx.test(elementsCollection[element].
                id)) {
76
77            let val = document.getElementById(elementsCollection[
                  element].id).value.trim();
78            if (val.length > 0)
79              addresses.push(val);
80          }
81        }
82        return addresses;
83      },
84
85      // validates of there is a key for a single recipient.
86      // maintains UI accordingly, by disabling or enabling textarea.
87      validateRecipientAndKey: function (recipient) {
88
89        let prefs = sbCommon.getCharPref('addressesAndKeys');
90
91        // handle "name <email>"" format
92        if (recipient.indexOf('<') > 0)
93          recipient = this.addressRegEx.exec(recipient)[1];
94
95        // check if key is known for recipient
96        let foundKey = false;
97        for (let i = 0; i < prefs.length; i++) {
98          if (prefs[i].addr && (prefs[i].addr === recipient))
99            foundKey = prefs[i].key;
100       }
101
102       if (foundKey)
103         this.enable(foundKey);
104       else
105         this.disable('nokey', recipient);
106     },
107
108     // disables the StegoBlock textarea for a specified reason.
109     disable: function (reason, extra) {
110
111       let label = this.elementMap('stegoblock-disabled-label');
112       let box = this.elementMap('stegoblock-disabled-box');
113       let textbox = this.elementMap('stegoblock-textbox');
114       let addbox = this.elementMap('stegoblock-add-key-box');
```

```
115         let reasonText;
116
117         switch (reason) {
118
119           case 'toomany': {
120
121             reasonText = 'Stego Block only supports one recipient';
122             addbox.collapsed = true;
123
124             break;
125           }
126           case 'nokey': {
127
128             this.recipient = extra;
129             reasonText = 'No StegoKey found for ' + extra;
130             addbox.collapsed = false;
131             this.validateKey();
132
133             break;
134           }
135         }
136
137         this.key = null;
138         label.value = reasonText;
139         box.collapsed = false;
140         textbox.collapsed = true;
141       },
142
143       // enables a previously disabled StegoBlock textarea
144       enable: function (key) {
145
146         let box = this.elementMap('stegoblock-disabled-box');
147         let textbox = this.elementMap('stegoblock-textbox');
148
149         this.key = key;
150         box.collapsed = true;
151         textbox.collapsed = false;
152       },
153
154       // fired on keyup when trying to add a new StegoKey
155       // validates if the key meets basic requirements, like length
156       validateKey: function () {
157
158         let value = this.elementMap('stegoblock-add-key').value;
159         let button = this.elementMap('stegoblock-add-button');
160
161         if (value === undefined || value.length < 8) {
162
163           button.disabled = true;
164           return;
165         }
166
167         button.disabled = false;
168       },
169
```

```
170     // fired on keydown of the StegoBlock textarea. ensures message
            length does
171     // not exceed maxMessageLength.
172     validateLength: function (event) {
173
174       let textboxValue = sb.ui.elementMap('stegoblock-textbox').
              value;
175       let remaining = sb.ui.maxMessageLength - textboxValue.length;
176       let keyCode = event.keyCode;
177
178       if (remaining <= 0 && event.keyCode !== 8 && event.keyCode
              !== 46) {
179
180         event.preventDefault();
181         return false;
182       }
183       return true;
184     },
185
186     // maintains a counter for remaining characters in the
            StegoBlock textarea
187     setRemainingCharCount: function (event) {
188
189       let label = this.elementMap('stegoblock-message-length');
190       let textbox = this.elementMap('stegoblock-textbox');
191
192       if (textbox.value.length > this.maxMessageLength) // prevents
              pasting of long texts
193         textbox.value = textbox.value.substring(0, this.
              maxMessageLength);
194
195       let remaining = this.maxMessageLength - textbox.value.length;
196       label.value = (remaining === 1 ? (remaining + ' char left') :
              (remaining + ' chars left'));
197     },
198
199     // adds a new (valid) StegoKey to the preferences
200     addKey: function () {
201
202       let textbox = this.elementMap('stegoblock-add-key');
203       let key = textbox.value;
204       let prefs = sbCommon.getCharPref('addressesAndKeys');
205
206       prefs.push({ addr: this.recipient, key: key });
207       sbCommon.setCharPref('addressesAndKeys', prefs);
208       textbox.value = '';
209     }
210   },
211
212   // fired after user clicks Send. injects the StegoBlock message
          in the email header
213   injectStegoBlockInMessageHeader: function (event) {
214
215     try {
216
```

```
217         let plaintext = document.getElementById('stegoblock-textbox')
                .value || '';
218         let date = (new Date()).toString();
219         let key = sb.ui.key;
220
221         // ensure a random key, if no message provided
222         if (plaintext.length === 0 || !key)
223           key = sbStego.getRandomString(128);
224
225         // hide!
226         let block = sbStego.encode(plaintext, date, key);
227         let check = sbStego.checkFrequency(block);
228
229         // block will contain adjacent spaces. those will be squashed
                by
230         // https://dxr.mozilla.org/mozilla-central/rev/82
                d0a583a9a39bf0b0000bccbf6d5c9ec2596bcc/addon-sdk/source/
                test/addons/e10s-content/lib/httpd.js#4639
231         // which is a normalization function that all headers go
                through. we cannot reverse
232         // this transformation, and must therefore transform spaces.
233         block = encodeURIComponent(block.join(''));
234
235         // check if block is valid
236         if (check.notInAlphabet.length > 0 || check.
                outsideFrequencyBounds.length > 0) {
237
238           let str = 'StegoBlock did not pass the character frequency
                  check and ' +
239             'your message was NOT send. ' +
240             'Either because you use invalid characters or too many of
                   some.\r\n\r\n' +
241             'Invalid characters:\r\n';
242
243           for (let x in check.notInAlphabet)
244             str += check.notInAlphabet[x] + ' ';
245
246           str += '\r\n\r\nCharacters used too many times:\r\n';
247
248           for (let x in check.outsideFrequencyBounds)
249             str += check.outsideFrequencyBounds[x] + ' ';
250
251           sb.ui.promptservice.alert(window, 'Fatal error', str);
252
253           event.preventDefault();
254           return false;
255         }
256
257         // fold headers, as lines cannot exceed 78 chars
258         block = sb.fold(block);
259
260         gMsgCompose.compFields.setHeader('X-StegoBlock', block);
261         gMsgCompose.compFields.setHeader('X-SBDate', date);
262
263         return true;
```

```
264
265       } catch (e) {
266
267         // it is crucial to cancel all emails without a StegoBlock,
              to preserve plausible deniability.
268         sb.ui.promptservice.alert(window, 'Fatal error', 'An
              unrecoverable error occured during StegoBlock generation.
               ' +
269           'To preserve "Plausible deniability", it is crucial that
                all outgoing emails contain a StegoBlock. Your email '
                +
270           'has been cancelled.');
271
272         // prevent from bubbling, cancelling sending.
273         event.preventDefault();
274         return false;
275       }
276     },
277
278     // adds spaces in a string by an interval. used for folding
          header
279     fold: function (str) {
280
281       let ret = [];
282       let len;
283       let n = 63;
284
285       for (let i = 0, len = str.length; i < len; i += n) {
286
287         if (i === n)
288           n += 13;
289
290         ret.push(str.substr(i, n));
291       }
292
293       return ret.join(' ');
294     }
295   };
296
297   window.addEventListener('compose-send-message', sb.
          injectStegoBlockInMessageHeader, true);
298   window.addEventListener('compose-window-init', function () {
299
300     gMsgCompose.RegisterStateListener(sb.ui);
301   }, true);
```

Listing C.7: StegoBlock extension messengercompose.js file

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
3
4  <prefwindow id="stegoblock-prefs" title="Stego Block Options" xmlns
       ="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul
       ">
5
```

```
6    <script type="application/javascript" src="chrome://stegoblock/
         content/json2.js"/>
7    <script type="application/javascript" src="chrome://stegoblock/
         content/common.js"/>
8    <script type="application/javascript" src="chrome://stegoblock/
         content/options.js"/>
9
10   <prefpane id="stegoblock-stettings-pane" label="StegoKey store">
11     <preferences>
12       <preference id="pref_symbol" name="stegoblock.symbol" type="
             string"/>
13     </preferences>
14
15     <listbox id="stegoblock-address-key-list" rows="10" width="400"
           seltype="multiple"
16       onselect="sb.onlistselect(this.selectedItems)">
17       <listhead>
18         <listheader label="Address" width="250"/>
19         <listheader label="Key" width="150"/>
20       </listhead>
21       <listcols>
22         <listcol/>
23         <listcol flex="1"/>
24       </listcols>
25     </listbox>
26
27     <hbox flex="1">
28       <button id="stegoblock-delete-key" flex="2" style="text-align
             :center" oncommand="sb.onDelete()" disabled="true">Delete
             </button>
29       <button id="stegoblock-purge" flex="2" oncommand="sb.onPurge
             ()" style="text-align:center">Purge StegoKey store</
             button>
30     </hbox>
31
32   </prefpane>
33
34 </prefwindow>
```

Listing C.8: StegoBlock extension options.xul file

```
1  const sbCommon = window.SBCommon();
2
3  var sb = {
4
5    // the selected StegoKeys
6    selectedPrefIndexes: [],
7
8    // service for prompting users
9    promptservice: Components.classes['@mozilla.org/embedcomp/prompt-
         service;1'].getService(Components.interfaces.nsIPromptService
         ),
10
11   // initializes a list with all stored StegoKeys.
12   // keys are not stored encrypted, it is considered unnecessary
```

```
13    init: function () {
14
15      let list = document.getElementById('stegoblock-address-key-list
            ');
16      let purgebutton = document.getElementById('stegoblock-purge');
17      let prefs = sbCommon.getCharPref('addressesAndKeys');
18
19      // remove any previously added list items in the list
20      for (let i = list.getRowCount() -1; i >= 0; i--)
21        list.removeItemAt(i);
22
23      // add list items with each StegoKey
24      for (let i = 0; i < prefs.length; i++) {
25
26        let row = document.createElement('listitem');
27        let cell = document.createElement('listcell');
28
29        row.setAttribute('value', i);
30        cell.setAttribute('label', prefs[i].addr);
31        row.appendChild(cell);
32
33        cell = document.createElement('listcell');
34        cell.setAttribute('label',  prefs[i].key );
35        row.appendChild(cell);
36
37        list.appendChild(row);
38      }
39
40      purgebutton.disabled = prefs.length === 0;
41    },
42
43    // fired when one or more items are selected in the list.
44    // maintains an array of selected StegoKeys
45    onlistselect: function (items) {
46
47      this.selectedPrefIndexes = [];
48      let button = document.getElementById('stegoblock-delete-key');
49
50      for (let item in items) {
51
52        try {
53
54          this.selectedPrefIndexes.push(parseInt(items[item].
                getAttribute('value')));
55        } catch(e) {
56
57        }
58      }
59
60      if (this.selectedPrefIndexes.length > 0)
61        button.disabled = false;
62      else
63        button.disabled = true;
64    },
65
```

```
66    // fired when Delete button is clicked. deletes the selected
67    // StegoKeys if user confirms.
68    onDelete: function () {
69
70      let text = this.selectedPrefIndexes.length > 1 ? 'Are you sure
            you want to delete these StegoKeys? This action cannot be
            undone.' : 'Are you sure you want to delete this StegoKey?
            This action cannot be undone.';
71
72      if (this.promptservice.confirm(window, 'Confirm deletion', text
            )) {
73
74        let prefs = sbCommon.getCharPref('addressesAndKeys');
75
76        for (let i = 0; i < this.selectedPrefIndexes.length; i++)
77          prefs.splice(this.selectedPrefIndexes[i], 1);
78
79        sbCommon.setCharPref('addressesAndKeys', prefs);
80        this.init();
81      }
82    },
83
84    // fired when Purge button is clicked. deletes all stored
          StegoKeys.
85    onPurge: function () {
86
87      if (this.promptservice.confirm(window, 'Confirm purge', 'Are
            you sure you want to delete all stored StegoKeys? This
            action cannot be undone.')) {
88
89        sbCommon.setCharPref('addressesAndKeys', []);
90        this.init();
91      }
92    }
93  };
94
95  window.addEventListener('load', function () {
96    sb.init();
97  }, false);
```

Listing C.9: StegoBlock extension options.js file

```
1
2  seedrandom.js
3  =============
4
5  Seeded random number generator for Javascript.
6
7  version 2.3.10
8  Author: David Bau
9  Date: 2014 Sep 20
10
11 Can be used as a plain script, a node.js module or an AMD module.
12
13 Script tag usage
```

```
14 ----------------
15
16 <script src=//cdnjs.cloudflare.com/ajax/libs/seedrandom/2.3.10/
       seedrandom.min.js>
17 </script>
18
19 // Sets Math.random to a PRNG initialized using the given explicit
       seed.
20 Math.seedrandom('hello.');
21 console.log(Math.random());          // Always 0.9282578795792454
22 console.log(Math.random());          // Always 0.3752569768646784
23
24 // Sets Math.random to an ARC4-based PRNG that is autoseeded using
       the
25 // current time, dom state, and other accumulated local entropy.
26 // The generated seed string is returned.
27 Math.seedrandom();
28 console.log(Math.random());          // Reasonably unpredictable.
29
30 // Seeds using the given explicit seed mixed with accumulated
       entropy.
31 Math.seedrandom('added entropy.', { entropy: true });
32 console.log(Math.random());          // As unpredictable as added
       entropy.
33
34 // Use "new" to create a local prng without altering Math.random.
35 var myrng = new Math.seedrandom('hello.');
36 console.log(myrng());                // Always 0.9282578795792454
37
38
39 Node.js usage
40 -------------
41
42 npm install seedrandom
43
44 // Local PRNG: does not affect Math.random.
45 var seedrandom = require('seedrandom');
46 var rng = seedrandom('hello.');
47 console.log(rng());                  // Always 0.9282578795792454
48
49 // Autoseeded ARC4-based PRNG.
50 rng = seedrandom();
51 console.log(rng());                  // Reasonably unpredictable.
52
53 // Global PRNG: set Math.random.
54 seedrandom('hello.', { global: true });
55 console.log(Math.random());          // Always 0.9282578795792454
56
57 // Mixing accumulated entropy.
58 rng = seedrandom('added entropy.', { entropy: true });
59 console.log(rng());                  // As unpredictable as added
       entropy.
60
61
62 Require.js usage
```

```
63  ----------------
64
65  Similar to node.js usage:
66
67  bower install seedrandom
68
69  require(['seedrandom'], function(seedrandom) {
70    var rng = seedrandom('hello.');
71    console.log(rng());                  // Always 0.9282578795792454
72  });
73
74
75  Network seeding
76  ---------------
77
78  <script src=//cdnjs.cloudflare.com/ajax/libs/seedrandom/2.3.10/
        seedrandom.min.js>
79  </script>
80
81  <!-- Seeds using urandom bits from a server. -->
82  <script src=//jsonlib.appspot.com/urandom?callback=Math.seedrandom
        ">
83  </script>
84
85  <!-- Seeds mixing in random.org bits -->
86  <script>
87  (function(x, u, s){
88    try {
89      // Make a synchronous request to random.org.
90      x.open('GET', u, false);
91      x.send();
92      s = unescape(x.response.trim().replace(/^|\s/g, '%'));
93    } finally {
94      // Seed with the response, or autoseed on failure.
95      Math.seedrandom(s, !!s);
96    }
97  })(new XMLHttpRequest, 'https://www.random.org/integers/' +
98    '?num=256&min=0&max=255&col=1&base=16&format=plain&rnd=new');
99  </script>
100
101 Reseeding using user input
102 --------------------------
103
104 var seed = Math.seedrandom();        // Use prng with an automatic
        seed.
105 document.write(Math.random());       // Pretty much unpredictable x
        .
106
107 var rng = new Math.seedrandom(seed); // A new prng with the same
        seed.
108 document.write(rng());               // Repeat the 'unpredictable'
        x.
109
110 function reseed(event, count) {      // Define a custom entropy
        collector.
```

```
111     var t = [];
112     function w(e) {
113       t.push([e.pageX, e.pageY, +new Date]);
114       if (t.length &lt; count) { return; }
115       document.removeEventListener(event, w);
116       Math.seedrandom(t, { entropy: true });
117     }
118     document.addEventListener(event, w);
119   }
120   reseed('mousemove', 100);            // Reseed after 100 mouse
          moves.
121
122   The "pass" option can be used to get both the prng and the seed.
123   The following returns both an autoseeded prng and the seed as an
          object,
124   without mutating Math.random:
125
126   var obj = Math.seedrandom(null, { pass: function(prng, seed) {
127     return { random: prng, seed: seed };
128   }});
129
130
131   Version notes
132   -------------
133
134   The random number sequence is the same as version 1.0 for string
          seeds.
135   * Version 2.0 changed the sequence for non-string seeds.
136   * Version 2.1 speeds seeding and uses window.crypto to autoseed if
          present.
137   * Version 2.2 alters non-crypto autoseeding to sweep up entropy
          from plugins.
138   * Version 2.3 adds support for "new", module loading, and a null
          seed arg.
139   * Version 2.3.1 adds a build environment, module packaging, and
          tests.
140   * Version 2.3.4 fixes bugs on IE8, and switches to MIT license.
141   * Version 2.3.6 adds a readable options object argument.
142   * Version 2.3.10 adds support for node.js crypto (contributed by
          ctd1500).
143
144   The standard ARC4 key scheduler cycles short keys, which means that
145   seedrandom('ab') is equivalent to seedrandom('abab') and 'ababab'.
146   Therefore it is a good idea to add a terminator to avoid trivial
147   equivalences on short string seeds, e.g., Math.seedrandom(str + '\0
          ').
148   Starting with version 2.0, a terminator is added automatically for
149   non-string seeds, so seeding with the number 111 is the same as
          seeding
150   with '111\0'.
151
152   When seedrandom() is called with zero args or a null seed, it uses
          a
153   seed drawn from the browser crypto object if present.  If there is
          no
```

```
154 crypto support, seedrandom() uses the current time, the native rng,
155 and a walk of several DOM objects to collect a few bits of entropy.
156
157 Each time the one- or two-argument forms of seedrandom are called,
158 entropy from the passed seed is accumulated in a pool to help
        generate
159 future seeds for the zero- and two-argument forms of seedrandom.
160
161 On speed - This javascript implementation of Math.random() is
        several
162 times slower than the built-in Math.random() because it is not
        native
163 code, but that is typically fast enough.  Some details (timings on
164 Chrome 25 on a 2010 vintage macbook):
165
166 * seeded Math.random()          - avg less than 0.0002 milliseconds
         per call
167 * seedrandom('explicit.')       - avg less than 0.2 milliseconds
        per call
168 * seedrandom('explicit.', true) - avg less than 0.2 milliseconds
        per call
169 * seedrandom() with crypto      - avg less than 0.2 milliseconds
        per call
170
171 Autoseeding without crypto is somewhat slower, about 20-30
        milliseconds on
172 a 2012 windows 7 1.5ghz i5 laptop, as seen on Firefox 19, IE 10,
        and Opera.
173 Seeded rng calls themselves are fast across these browsers, with
        slowest
174 numbers on Opera at about 0.0005 ms per seeded Math.random().
175
176
177 LICENSE (MIT)
178 -------------
179
180 Copyright 2014 David Bau.
181
182 Permission is hereby granted, free of charge, to any person
        obtaining
183 a copy of this software and associated documentation files (the
184 "Software"), to deal in the Software without restriction, including
185 without limitation the rights to use, copy, modify, merge, publish,
186 distribute, sublicense, and/or sell copies of the Software, and to
187 permit persons to whom the Software is furnished to do so, subject
        to
188 the following conditions:
189
190 The above copyright notice and this permission notice shall be
191 included in all copies or substantial portions of the Software.
192
193 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
194 EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
195 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
        NONINFRINGEMENT.
```

```
196 IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
        ANY
197 CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT
        ,
198 TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
199 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
200
201 */
202
203 /**
204  * All code is in an anonymous closure to keep the global namespace
         clean.
205  */
206 (function (
207     global, pool, math, width, chunks, digits, module, define,
          rngname) {
208
209 //
210 // The following constants are related to IEEE 754 limits.
211 //
212 var startdenom = math.pow(width, chunks),
213     significance = math.pow(2, digits),
214     overflow = significance * 2,
215     mask = width - 1,
216     nodecrypto;
217
218 //
219 // seedrandom()
220 // This is the seedrandom function described above.
221 //
222 var impl = math['seed' + rngname] = function(seed, options,
        callback) {
223   var key = [];
224   options = (options == true) ? { entropy: true } : (options || {})
          ;
225
226   // Flatten the seed string or build one from local entropy if
          needed.
227   var shortseed = mixkey(flatten(
228     options.entropy ? [seed, tostring(pool)] :
229     (seed == null) ? autoseed() : seed, 3), key);
230
231   // Use the seed to initialize an ARC4 generator.
232   var arc4 = new ARC4(key);
233
234   // Mix the randomness into accumulated entropy.
235   mixkey(tostring(arc4.S), pool);
236
237   // Calling convention: what to return as a function of prng, seed
          , is_math.
238   return (options.pass || callback ||
239       // If called as a method of Math (Math.seedrandom()), mutate
            Math.random
240       // because that is how seedrandom.js has worked since v1.0.
            Otherwise,
```

```
241       // it is a newer calling convention, so return the prng
              directly.
242       function(prng, seed, is_math_call) {
243         if (is_math_call) { math[rngname] = prng; return seed; }
244         else return prng;
245       })(
246
247   // This function returns a random double in [0, 1) that contains
248   // randomness in every bit of the mantissa of the IEEE 754 value.
249   function() {
250     var n = arc4.g(chunks),             // Start with a numerator n
              < 2 ^ 48
251         d = startdenom,                 //   and denominator d = 2
                ^ 48.
252         x = 0;                          //   and no 'extra last
                byte'.
253     while (n < significance) {          // Fill up all significant
            digits by
254       n = (n + x) * width;              //   shifting numerator and
255       d *= width;                       //   denominator and
              generating a
256       x = arc4.g(1);                    //   new least-significant-
              byte.
257     }
258     while (n >= overflow) {             // To avoid rounding up,
            before adding
259       n /= 2;                           //   last byte, shift
              everything
260       d /= 2;                           //   right using integer
              math until
261       x >>>= 1;                         //   we have exactly the
              desired bits.
262     }
263     return (n + x) / d;                 // Form the number within
            [0, 1).
264   }, shortseed, 'global' in options ? options.global : (this ==
        math));
265 };
266
267 //
268 // ARC4
269 //
270 // An ARC4 implementation.  The constructor takes a key in the form
        of
271 // an array of at most (width) integers that should be 0 <= x < (
      width).
272 //
273 // The g(count) method returns a pseudorandom integer that
      concatenates
274 // the next (count) outputs from ARC4.  Its return value is a
      number x
275 // that is in the range 0 <= x < (width ^ count).
276 //
277 /** @constructor */
278 function ARC4(key) {
```

```
279     var t, keylen = key.length,
280         me = this, i = 0, j = me.i = me.j = 0, s = me.S = [];
281
282     // The empty key [] is treated as [0].
283     if (!keylen) { key = [keylen++]; }
284
285     // Set up S using the standard key scheduling algorithm.
286     while (i < width) {
287       s[i] = i++;
288     }
289     for (i = 0; i < width; i++) {
290       s[i] = s[j = mask & (j + key[i % keylen] + (t = s[i]))];
291       s[j] = t;
292     }
293
294     // The "g" method returns the next (count) outputs as one number.
295     (me.g = function(count) {
296       // Using instance members instead of closure state nearly
              doubles speed.
297       var t, r = 0,
298           i = me.i, j = me.j, s = me.S;
299       while (count--) {
300         t = s[i = mask & (i + 1)];
301         r = r * width + s[mask & ((s[i] = s[j = mask & (j + t)]) + (s
              [j] = t))];
302       }
303       me.i = i; me.j = j;
304       return r;
305       // For robust unpredictability, the function call below
              automatically
306       // discards an initial batch of values.  This is called RC4-
              drop[256].
307       // See http://google.com/search?q=rsa+fluhrer+response&btnI
308     })(width);
309   }
310
311   //
312   // flatten()
313   // Converts an object tree to nested arrays of strings.
314   //
315   function flatten(obj, depth) {
316     var result = [], typ = (typeof obj), prop;
317     if (depth && typ == 'object') {
318       for (prop in obj) {
319         try { result.push(flatten(obj[prop], depth - 1)); } catch (e)
              {}
320       }
321     }
322     return (result.length ? result : typ == 'string' ? obj : obj + '
          \0');
323   }
324
325   //
326   // mixkey()
327   // Mixes a string seed into a key that is an array of integers, and
```

```
328 // returns a shortened string seed that is equivalent to the result
        key.
329 //
330 function mixkey(seed, key) {
331   var stringseed = seed + '', smear, j = 0;
332   while (j < stringseed.length) {
333     key[mask & j] =
334       mask & ((smear ^= key[mask & j] * 19) + stringseed.charCodeAt
              (j++));
335   }
336   return tostring(key);
337 }
338
339 //
340 // autoseed()
341 // Returns an object for autoseeding, using window.crypto if
        available.
342 //
343 /** @param {Uint8Array|Navigator=} seed */
344 function autoseed(seed) {
345   try {
346     if (nodecrypto) return tostring(nodecrypto.randomBytes(width));
347     global.crypto.getRandomValues(seed = new Uint8Array(width));
348     return tostring(seed);
349   } catch (e) {
350     return [+new Date, global, (seed = global.navigator) && seed.
            plugins,
351       global.screen, tostring(pool)];
352   }
353 }
354
355 //
356 // tostring()
357 // Converts an array of charcodes to a string
358 //
359 function tostring(a) {
360   return String.fromCharCode.apply(0, a);
361 }
362
363 //
364 // When seedrandom.js is loaded, we immediately mix a few bits
365 // from the built-in RNG into the entropy pool.  Because we do
366 // not want to interfere with deterministic PRNG state later,
367 // seedrandom will not call math.random on its own again after
368 // initialization.
369 //
370 mixkey(math[rngname](), pool);
371
372 //
373 // Nodejs and AMD support: export the implementation as a module
        using
374 // either convention.
375 //
376 if (module && module.exports) {
377   module.exports = impl;
```

```
    try {
      // When in node.js, try using crypto package for autoseeding.
      nodecrypto = require('crypto');
    } catch (ex) {}
  } else if (define && define.amd) {
    define(function() { return impl; });
  }

  //
  // Node.js native crypto support.
  //

  // End anonymous scope, and pass initial values.
  })(
    this,    // global window object
    [],      // pool: entropy pool starts empty
    Math,    // math: package containing random, pow, and seedrandom
    256,     // width: each RC4 output is 0 <= x < 256
    6,       // chunks: at least six RC4 outputs for each double
    52,      // digits: there are 52 significant digits in a double
    (typeof module) == 'object' && module,    // present in node.js
    (typeof define) == 'function' && define,  // present with an AMD
        loader
    'random'// rngname: name for Math.random and Math.seedrandom
  );
```

Listing C.10: StegoBlock extension seedrandom.js file - by David Bau

```
var SBStego = function () {

  return {

    maxPlaintextLength: 200,
    blockLength: 4400,

    alphabetFrequencies: {

      ' ': 16.06718960,
      'e': 8.38191046,
      't': 5.97449455,
      'o': 5.49426190,
      'a': 5.49365722,
      'n': 5.17089898,
      'i': 4.87451515,
      'r': 4.55353236,
      's': 4.31688330,
      'l': 2.93379732,
      'h': 2.70875299,
      'd': 2.40453403,
      'c': 2.26601057,
      'u': 1.97602092,
      'm': 1.76507724,
      'p': 1.50065145,
      'f': 1.34908232,
      'y': 1.34689517,
```

```
28          'g': 1.32540969,
29          '.': 1.14563926,
30          'w': 1.13791993,
31          'b': 0.92085225,
32          ',': 0.83979924,
33          'O': 0.83385535,
34          'v': 0.74238124,
35          '-': 0.70177754,
36          'E': 0.68850029,
37          '=': 0.64724045,
38          'k': 0.58342728,
39          'T': 0.56770557,
40          '2': 0.51566439,
41          'C': 0.51247374,
42          '/': 0.47558818,
43          'S': 0.47345250,
44          '1': 0.43507454,
45          'A': 0.43493302,
46          'I': 0.42157858,
47          '_': 0.36992336,
48          'M': 0.36024846,
49          'N': 0.33621560,
50          'P': 0.32306700,
51          'U': 0.32287402,
52          'D': 0.31618393,
53          'R': 0.30635465,
54          '>': 0.27271121,
55          ':': 0.26795096,
56          '3': 0.26488896,
57          '\'': 0.22384783,
58          'B': 0.21430158,
59          'H': 0.21057057,
60          'L': 0.20983724,
61          'F': 0.19583951,
62          '\t': 0.19488746,
63          '@': 0.19023013,
64          '5': 0.18643479,
65          '9': 0.18505817,
66          'W': 0.18344998,
67          'x': 0.18092833,
68          '?': 0.18050377,
69          'G': 0.17274584,
70          '4': 0.16012472,
71          '7': 0.15750015,
72          'U': 0.14626852,
73          '8': 0.14083925,
74          '6': 0.13537139,
75          'J': 0.13009651,
76          ')': 0.12981347,
77          '(': 0.12127074,
78          '<': 0.10048000,
79          'q': 0.09605425,
80          'j': 0.09571974,
81          'K': 0.08672672,
82          'z': 0.08664953,
```

```
83        'V': 0.08546590,
84        'Y': 0.07774656,
85        ';': 0.06578159,
86        '*': 0.06518978,
87        '&': 0.05738038,
88        '$': 0.05343066,
89        '"': 0.05126925,
90        '!': 0.04618735,
91        'X': 0.04392301,
92        '+': 0.03535455,
93        'Z': 0.02887031,
94        'Q': 0.02826563,
95        '|': 0.02237320,
96        '~': 0.02202583,
97        ']': 0.01722698,
98        '[': 0.01717552,
99        '%': 0.01478253,
100       '\\': 0.01220941,
101       '#': 0.01201643,
102       '`': 0.00562225,
103       '{': 0.00015439,
104       '}': 0.00014152
105     },
106
107     generateNoise: function (sizeArr, plaintextArr) {
108
109       let input = sizeArr.concat(plaintextArr);
110       let noise = [];
111       let ptDict = {};
112
113       // verify that all chars in plaintext exist in the alphabet.
114       // track how many times each char occur.
115       for (let i = 0; i < input.length; i++) {
116
117         // init bucket if none exists.
118         if (ptDict[input[i]] === undefined)
119           ptDict[input[i]] = 0;
120
121         // increment char count.
122         ptDict[input[i]]++;
123       }
124
125       // run through all chars of the alphabet.
126       for (let x in this.alphabetFrequencies) {
127
128         // calculate the char count given the specified block
              length (4400) and frequency
129         let charCount = Math.round(this.blockLength / 100 * this.
              alphabetFrequencies[x]);
130         let ptFreq = ptDict[x] || 0;
131
132         charCount = charCount - ptFreq; // subtract the char count
              in the plaintext, from the calculated.
133         if (charCount < 0)
```

```
134             charCount = 0; // there is already too many of the given
                    char, to maintain correct frequency. notify about
                    this later.
135
136           // as the frequency and char count calculated is now with
                  respect to the plaintext, push the char onto the noise
137           // array "charCount" times.
138           for (let i = 0; i < charCount; i++)
139             noise.push(x);
140         }
141
142         // shuffle noise, as we would otherwise reveal if some key is
                fake and ruin plausible deniability.
143         this.shuffle(new Math.seedrandom(), noise);
144
145         return noise;
146       },
147
148       encode: function (plaintext, seed, key) {
149
150         if(plaintext.length > this.maxPlaintextLength)
151           throw 'Plaintext too long';
152
153         let plaintextArr = typeof plaintext === 'string' ? plaintext.
                split('') : plaintext; // convert plaintext to string
                array
154         let length = plaintextArr.length.toString();
155
156         if (plaintextArr.length === 0) {
157
158           while (this.isPositiveInteger(length))
159             length = this.getRandomString(3);
160         }
161
162         let prng = new Math.seedrandom(seed + key); // seed the prng
                with desired key
163         let sizeArr = this.leftPad(length, '000').split('');
164         let noise = this.generateNoise(sizeArr, plaintextArr); //
                generate noise with correct letter frequencies
165         let block = sizeArr.concat(plaintextArr).concat(noise);
166
167         this.shuffle(prng, block);
168
169         return block;
170       },
171
172       decode: function (block, seed, key) {
173
174         let prng = new Math.seedrandom(seed + key);
175         block = block.split('');
176
177         this.unshuffle(prng, block);
178
179         let sizeStr = block.slice(0, 3).join('');
180
```

```
181       // 3 first chars must be digits to be valid
182       if (!this.isPositiveInteger(sizeStr))
183         return '';
184
185       // parse the size of the plaintext to an int, so we can slice
              it off
186       let size = parseInt(sizeStr);
187
188       // must be valid length
189       if (size < 0 || size > this.maxPlaintextLength)
190         return '';
191
192       return block.slice(3, 3 + size).join('');
193     },
194
195     // knuth-fisher-yates shuffle
196     shuffle: function (prng, arr) {
197
198       for (let i = arr.length - 1; i > 0; i--) {
199
200         let j = this.getRandomInRange(prng, 0, i);
201         let temp = arr[i];
202
203         arr[i] = arr[j];
204         arr[j] = temp;
205       }
206
207       return arr;
208     },
209
210     // reverse knuth-fisher-yates shuffle. only works if prng is in
           same state as when shuffled.
211     unshuffle: function (prng, arr) {
212
213       // generate all swapping positions needed, so we may start
            with the last one.
214       let indexes = [];
215       for (let i = arr.length - 1; i > 0; i--)
216         indexes.unshift(this.getRandomInRange(prng, 0, i));
217
218       // reverse knuth-fisher-yates shuffle
219       for (let i = 1; i < arr.length; i++) {
220
221         let j = indexes.shift();
222         let temp = arr[i];
223
224         arr[i] = arr[j];
225         arr[j] = temp;
226       }
227       return arr;
228     },
229
230     // checks if a string has correct frequency of each char,
           according to alphabetFrequencies.
231     checkFrequency: function (string) {
```

```
232
233         let dict = {};
234         let ret = {
235
236           notInAlphabet: [],
237           outsideFrequencyBounds: []
238         };
239
240         for (let i = 0; i < string.length; i++) {
241
242           if (dict[string[i]] === undefined)
243             dict[string[i]] = 0;
244
245           dict[string[i]]++;
246         }
247
248         let frequencies = [];
249         let sortedKeys = Object.keys(dict).sort();
250
251         for (let k in sortedKeys) {
252
253           let charCount = Math.round(this.blockLength / 100 * this.
                alphabetFrequencies[sortedKeys[k]]);
254           let isInAlphabet = this.alphabetFrequencies[sortedKeys[k]]
                !== undefined;
255           let isFrequencyWithinBounds = isInAlphabet && charCount ===
                 dict[sortedKeys[k]];
256
257           if (!isInAlphabet)
258             ret.notInAlphabet.push(sortedKeys[k]);
259           if (!isFrequencyWithinBounds)
260             ret.outsideFrequencyBounds.push(sortedKeys[k]);
261         }
262
263         return ret;
264       },
265
266       // returns the next char of a plaintext array or noise, if the
             first is empty.
267       getChar: function (plaintext, noise) {
268
269         if (plaintext.length > 0)
270           return plaintext.shift();
271
272         return noise.shift();
273       },
274
275       // checks if some input is a positive integer. from: http://
             stackoverflow.com/a/10835227
276       isPositiveInteger: function (input) {
277         return 0 === input % (!isNaN(parseFloat(input)) && 0 <= ~~
               input);
278       },
279
```

```
280     // returns a random int in the specified range (including),
            using the provided function.
281     getRandomInRange: function (prng, min, max) {
282
283       min = Math.ceil(min);
284       max = Math.floor(max);
285
286       return Math.floor(prng() * (max - min + 1)) + min;
287     },
288
289     // left pads some string with some other string
290     leftPad: function (text, pad) {
291
292       if (typeof text === 'undefined')
293         return pad;
294
295       return (pad + text).substring(text.length, text.length + pad.
              length);
296     },
297
298     // generates random string of given length. only alpha numeric
            chars.
299     getRandomString: function (length, prng) {
300
301       let text = '';
302       let possible = Object.keys(this.alphabetFrequencies).join('')
              ;
303
304       if (!prng)
305         prng = new Math.seedrandom();
306
307       for (let i = 0; i < length; i++)
308         text += possible.charAt(Math.floor(prng() * possible.length
              ));
309
310       return text;
311     }
312   };
313 };
314
315 // extend the global variable with common functionality, for easy
        access
316 window.SBCommon.utils.extend(window.SBStego, SBStego());
```

Listing C.11: StegoBlock extension steganography.js file

Appendix D

StegoBlock extension images and screenshots

Figure D.1: StegoBlock icon

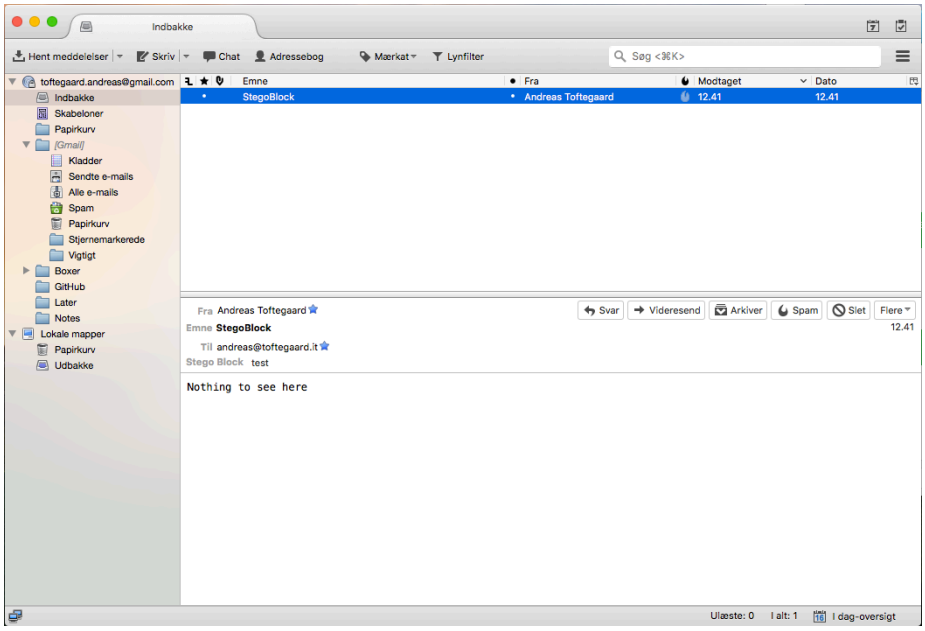


Figure D.2: StegoBlock view-message-window - stego-key found

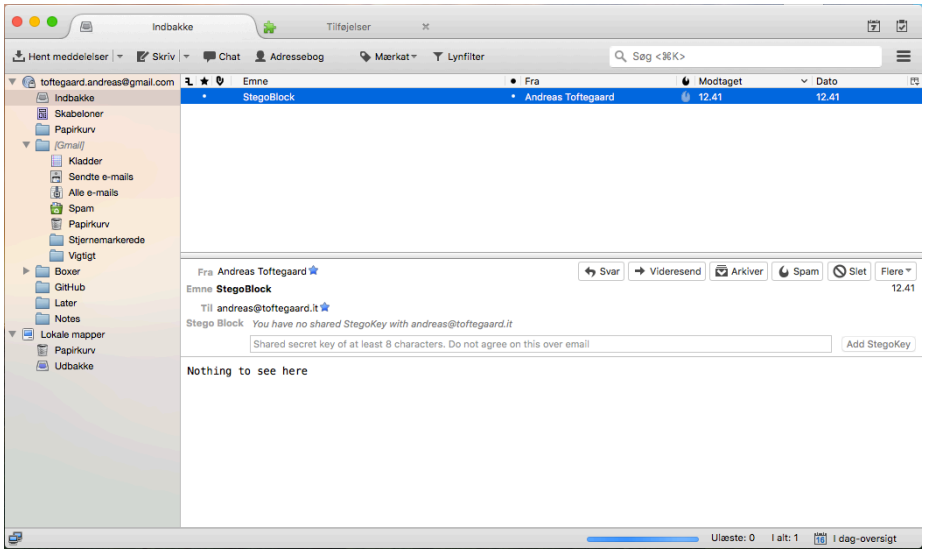


Figure D.3: StegoBlock view-message-window - no stego-key found

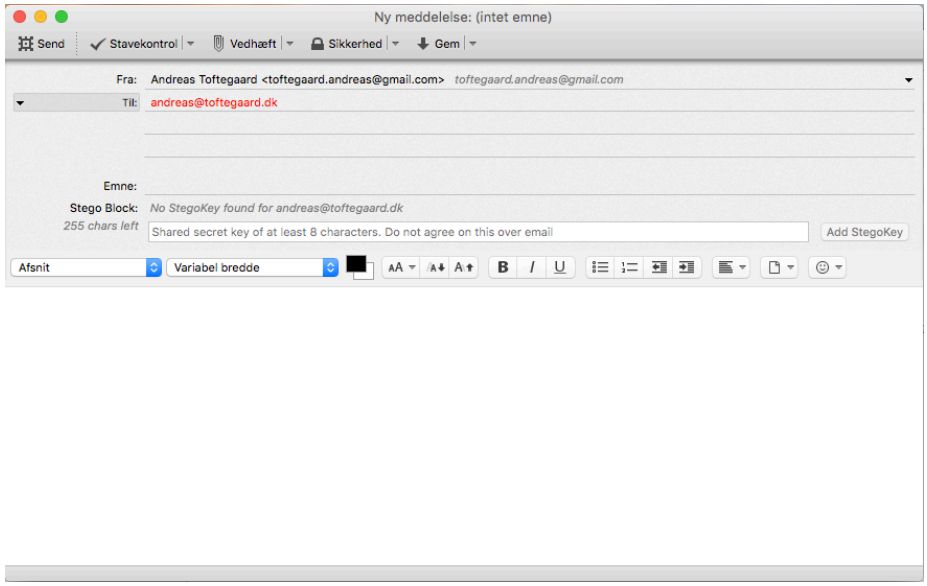


Figure D.4: StegoBlock compose-message-window - stego-key found

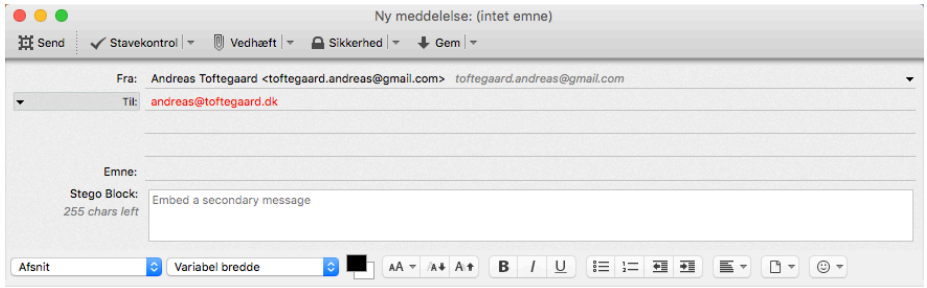


Figure D.5: StegoBlock compose-message-window - no stego-key found

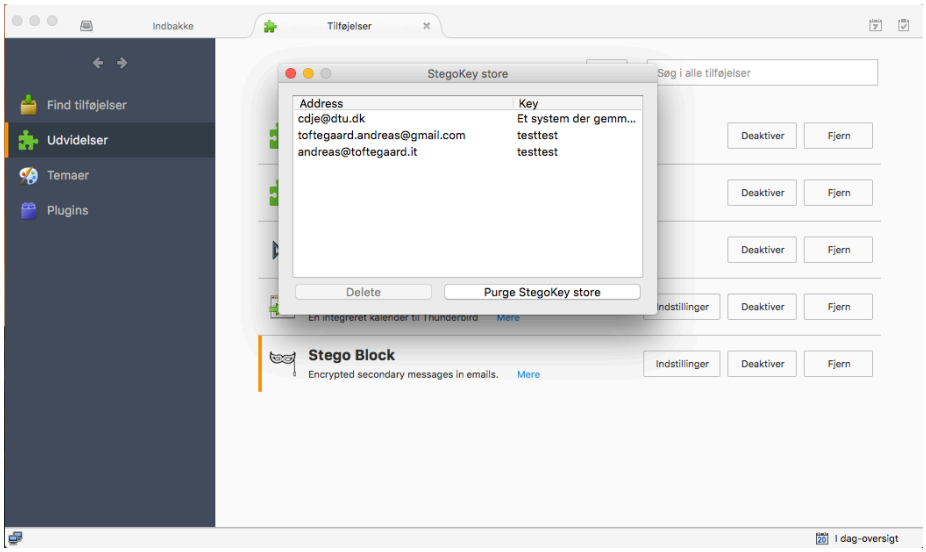


Figure D.6: StegoBlock options-window

Appendix E

Total Block Length analysis results

SPACE	16.06718960	/	0.47558818	6	0.13537139
e	8.38191046	S	0.47345250	J	0.13009651
t	5.97449455	1	0.43507454	)	0.12981347
o	5.49426190	A	0.43493302	(	0.12127074
a	5.49365722	I	0.42157858	<	0.10048000
n	5.17089898	_	0.36992336	q	0.09605425
i	4.87451515	M	0.36024846	j	0.09571974
r	4.55353236	N	0.33621560	K	0.08672672
s	4.31688330	P	0.32306700	z	0.08664953
l	2.93379732	O	0.32287402	V	0.08546590
h	2.70875299	D	0.31618393	Y	0.07774656
d	2.40453403	R	0.30635465	;	0.06578159
c	2.26601057	>	0.27271121	*	0.06518978
u	1.97602092	:	0.26795096	&	0.05738038
m	1.76507724	3	0.26488896	$	0.05343066
p	1.50065145	’	0.22384783	"	0.05126925
f	1.34908232	B	0.21430158	!	0.04618735
y	1.34689517	H	0.21057057	X	0.04392301
g	1.32540969	L	0.20983724	+	0.03535455
.	1.14563926	F	0.19583951	Z	0.02887031
w	1.13791993	TAB	0.19488746	Q	0.02826563

b	0.92085225	@	0.19023013	|	0.02237320
,	0.83979924	5	0.18643479	∼	0.02202583
0	0.83385535	9	0.18505817	]	0.01722698
v	0.74238124	W	0.18344998	[	0.01717552
-	0.70177754	x	0.18092833	%	0.01478253
E	0.68850029	?	0.18050377	\	0.01220941
=	0.64724045	G	0.17274584	#	0.01201643
k	0.58342728	4	0.16012472	‘	0.00562225
T	0.56770557	7	0.15750015	{	0.00015439
2	0.51566439	U	0.14626852	}	0.00014152
C	0.51247374	8	0.14083925		

Table E.1: StegoBlock FREQUENCY_ALPHABET

TBL	MAR 140	MAR 200	Penalized	MAR $\frac{1}{4}$
800	78,69643935	34,69510968	0,043368887	34,69510968
1000	82,82035808	44,79774603	0,044797746	37,31133025
1200	84,99295916	52,00241497	0,043335346	38,67981485
1400	88,95594448	54,53813645	0,038955812	39,29120547
1600	89,1973446	58,4624673	0,036539042	40,51881666
1800	90,4043452	64,70114711	0,035945082	43,38901187
2000	90,88714544	67,6796136	0,033839807	44,69712216
2200	91,59122913	69,45059368	0,031568452	45,82410948
2400	91,81251257	70,49708191	0,029373784	45,82410948
2600	92,9993965	72,65043268	0,027942474	47,97746025
2800	92,45624623	75,95089555	0,02712532	52,12316361
3000	93,50231342	76,89675991	0,025632253	52,54578386
3200	93,88453028	77,39987925	0,024187462	52,78728114
3400	93,92476363	79,73435299	0,02345128	54,94063192
3600	94,14604707	80,29784665	0,022304957	55,80599718
3800	96,01689801	87,84463675	0,02311701	60,05232441
4000	96,1979481	88,97162407	0,022242906	61,54155766
4200	96,68074834	89,37411954	0,021279552	62,52767156
4400	96,60028163	89,6357416	0,020371759	62,66854498
4600	96,94226514	90,98410143	0,019779152	65,28476555
4800	96,72098169	91,02435098	0,018963406	65,08351781
5000	96,98249849	91,22559871	0,01824512	66,21050513
5200	96,76121505	91,32622258	0,017562735	66,39162809
5400	96,98249849	91,60796941	0,016964439	66,89474743
5600	96,96238181	91,99034011	0,016426846	67,45824109
5800	97,22389861	92,1513383	0,015888162	67,90098611
6000	97,26413197	92,17146307	0,015361911	68,18273294
6200	97,30436532	92,77520628	0,014963743	69,04809821

6400	97,324482	93,23807607	0,014568449	69,95371302
6600	97,324482	93,23807607	0,014126981	70,29583417
6800	97,4049487	93,29845039	0,01372036	70,31595895
7000	97,34459867	93,56007245	0,013365725	70,83920306
7200	97,38483203	93,66069632	0,01300843	71,12094989
7400	97,26413197	94,22418998	0,012732999	71,68444355
7600	97,38483203	94,38518817	0,012419104	71,96619038
7800	97,78716556	94,52606158	0,012118726	72,42906017
8000	97,80728224	94,62668545	0,011828336	73,01267861
8200	97,88774894	94,64681022	0,011542294	73,41517408
8400	97,90786562	94,88830751	0,011296227	74,11954116
8600	97,9279823	94,94868183	0,011040544	74,80378346
8800	98,02856568	94,94868183	0,010789623	75,00503119
9000	97,88774894	95,0493057	0,010561034	75,24652848
9200	98,008449	95,0493057	0,010331446	75,3069028
9400	98,08891571	95,0493057	0,010111628	75,36727712
9600	98,12914906	95,08955524	0,009905162	75,62889917
9800	97,98833233	95,19017911	0,009713284	76,25276716
10000	98,02856568	95,59267458	0,009559267	77,01750855
10200	98,08891571	95,85429664	0,00939748	77,70175086
10400	98,22973245	95,87442141	0,009218694	77,90299859
10600	98,24984912	95,89454619	0,009046655	78,00362246
10800	98,2699658	96,05554438	0,008894032	78,26524452
11000	98,2699658	96,13604347	0,00873964	78,34574361
11200	98,2699658	96,17629302	0,008587169	78,4262427
11400	98,24984912	96,19641779	0,008438282	78,52686657
11600	98,33031583	96,27691688	0,008299734	78,94948682
11800	98,31019916	96,29704166	0,008160766	79,17085933
12000	98,37054919	96,31716643	0,008026431	79,61360435
12200	98,2699658	96,33729121	0,007896499	79,91547595
12400	98,37054919	96,39766553	0,007774005	80,21734755
12600	98,33031583	96,45803985	0,0076554	80,45884484
12800	98,33031583	96,47816462	0,007537357	80,51921916
13000	98,33031583	96,53853894	0,007426041	80,68021735
13200	98,37054919	96,53853894	0,007313526	80,9418394
13400	98,29008248	96,57878849	0,007207372	81,00221373
13600	98,29008248	96,57878849	0,007101382	81,16321191
13800	98,37054919	96,63916281	0,007002838	81,4047092
14000	98,35043251	96,65928758	0,006904235	81,62608171
14200	98,49124925	96,65928758	0,006806992	81,84745422
14400	98,47113257	96,69953713	0,006715246	82,04870195
14600	98,49124925	96,7196619	0,006624634	82,0084524
14800	98,57171595	96,75991145	0,006537832	82,37069833
15000	98,57171595	96,75991145	0,006450661	82,33044878

Table E.2: Test results for MAR vs. TBL

Bibliography

[AA96] R. Anderson and R. Anderson. Stretching the limits of steganography. 1996.

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. *Proceedings of the Acm Conference on Computer and Communications Security*, 2015-:5–17, 2015.

[AR10] James Nechvatal Miles Smid Elaine Barker Stefan Leigh Mark Levenson Mark Vangel David Banks Alan Heckert James Dray San Vo Andrew Rukhin, Juan Soto. A statistical test suite for random and pseudorandom number generators for cryptographic applications. 2010. Accessed: 2016-10-07 13:33.

[Atw07] Jeff Atwood. The danger of naïveté. 2007. Accessed: 2016-10-05 13:12.

[BS86] M. Blum and M. Shub. A simple unpredictable pseudo-random number generator. *Siam Journal on Computing*, 15(2):364–83, 364–383, 1986.

[BZ14] C. Blackwell and H. Zhu. *Cyberpatterns: Unifying Design Patterns with Security and Attack Patterns*. Springer International Publishing, 2014.

[Cac04] C Cachin. An information-theoretic model for steganography. *Information and Computation*, 192(1):41–56, 2004.

[CDJCB05] Giacomo Cancelli, Gwenaël Doërr, Ingemar J. Cox, and Mauro Barni. Detection of ś1 lsb steganography based on the amplitude of histogram local extrema. 2005.

[Coh06] Electronic Frontier Foundation Cindy Cohn. At&ts role in dragnet surveillance of millions of its customers. 2006.

[daw96] Randomness and the netscape browser. `https://people.eecs.berkeley.edu/~daw/papers/ddj-netscape.html`, 1996. Accessed: 2016-09-13 13:22.

[dBB93] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of md5. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 293–304. Springer Berlin Heidelberg, 1993.

[DH76] W DIFFIE and ME HELLMAN. New directions in cryptography. *Ieee Transactions on Information Theory*, 22(6):644–654, 1976.

[DSM+04] Onkar Dabeer, Kenneth Sullivan, Upamanyu Madhow, Shivakumar Chandrasekaran, and B. S. Manjunath. Detection of hiding in the least significant bit. 2004.

[DY83] Dolev and Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[FIS53] R.A. FISHER. *STATISTICAL TABLES FOR BIOLOGICAL, AGRICULTURAL AND MEDICAL RESEARCH*. 1953.

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–83, January 1883.

[Kli06] Vlastimil Klima. Tunnels in hash functions: Md5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006. `http://eprint.iacr.org/2006/105`.

[Knu68] D. E. Knuth. *The art of computer programming*. Addison-Wesley., 1968.

[KP00] S. Katzenbeisser and F.A.P. Petitcolas. *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.

[NIS15] NIST. Nist's policy on hash functions. 2015. Accessed: 2016-10-08 10:30.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[oST08] National Institute of Standards and Technology. Guide to general server security, 2008. `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf`.

[PHGW16] Christoph Ponikwar, Hans-Joachim Hof, Smriti Gopinath, and Lars Wischhof. Beyond the dolev-yao model: Realistic application-specific attacker models for applications using vehicular communication. 2016.

[rfc82] Standard for the format of arpa internet text messages. `https://tools.ietf.org/html/rfc822`, 1982. Accessed: 2016-09-16 11:32.

[rfc96] Mime (multipurpose internet mail extensions) part two: Message header extensions for non-ascii text. `https://tools.ietf.org/html/rfc2047`, 1996. Accessed: 2016-09-16 11:33.

[rfc08] Internet message format. `https://tools.ietf.org/html/rfc5322`, 2008. Accessed: 2016-10-31 10:13.

[rfc14] Sender policy framework (spf) for authorizing use of domains in email, version 1. `https://tools.ietf.org/html/rfc7208`, 2014. Accessed: 2016-10-12 10:24.

[Riv98] Ronald L. Rivest. Chaffing and winnowing: Confidentiality without encryption. 1998.

[SC84] G. J. Simmons and D. Chaum. The prisoner's problem and the subliminal channel. 1984.

[SS05] A Sidorenko and B Schoemnakers. Concrete security of the blum-blum-shub pseudorandom generator. *Lecture Notes in Computer Science*, 3796:355–375, 2005.

[was] U.s., british intelligence mining data from nine u.s. internet companies in broad secret program. `https://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html`. Accessed: 2016-09-09 09:24.

[Way09] Peter Wayner. Disappearing cryptography. *Disappearing Cryptography*, 2009.

[wik] Wikipedia - fisher-yates shuffle - modulo bias. Accessed: 2016-10-05 14:35.

[Yao82] A. C. Yao. Theory and applications of trapdoor functions. *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, 80–91, 1982.

[ZDN] Prism: Here's how the nsa wiretapped the internet. `http://www.zdnet.com/article/prism-heres-how-the-nsa-wiretapped-the-internet`. Accessed: 2016-09-09 10:27.