Investigating the Security Level of Encryption Methods in the Deep Web

Philip Nassr

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2017

Investigating the Security Level of Encryption Methods in the Deep Web

Submitted by: Philip Nassr

Abstract

Browsing safely on the Web is a major issue nowadays. There have been multiple solutions to maximise security using modern encryption methods such as RSA and SHA-256. HTTPS is basic, yet the most widely-known and popular solution for establishing safe client-server connection. Most websites have HTTPS support, but web servers do not always ensure that clients connect securely.

The thesis discusses the security issues in the clear and the deep web, with a contribution to an open-source project that facilitates safer web browsing. It allows users to automatically establish a secure connection with a web server without explicitly having to state it through an *https://* tag. The dissertation also represents a potential solution for greater efficiency and process automation.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank all the people, who helped me in completing my final year project. Dr. John Power, my personal tuTor, was extremely helpful with his guidance and mentoring. Special thanks to the Electronic Frontier Foundation, the Tor project, and all the contributors of “HTTPS Everywhere”, thanks to which I had the opportunity to work on the project and get some very useful pieces of advice.

Chapter 1

Introduction

1.1 Problem Description

1.1.1 General Information

The Web has been around for many years. A common misconception is that the Web is the Internet - the Web is actually one of the very few features the Internet offers. It is a shared network space available to the public, where documents and pages are identified by a Uniform Resource Locator (URL). Its primary usage is for resource sharing, communication and collaboration, information gathering, and entertainment [55].

With all the fancy features provided by the Internet, security has always been an issue. To begin with, one should always consider the type of information shared. Sharing sensitive information such as home address or bank account number poses personal security at risk. However, online payment and delivery services have been present long enough and indeed require these sensitive details. To make such actions as safe as possible, several encryption methods are used to avoid data exposure or theft. Of course there have been multiple cases of account thefts that became part of the underground economy of cyber criminals [33]. Details on how to minimise the risk will be discussed further. Anonymous browsing is another form of security. Anonymity is when a user cannot be directly identified by a third party when surfing the Web. The IP address and user requests are encrypted and if there is a man in the middle, be it an eavesdropper or hacker, work needs to be done to decrypt them. This can be achieved via proxy servers, Virtual Private Networking (VPN) and web browsers such as Tor (The Onion Router). The project's main focus will be on how the Tor browser and other I2P networks work, what makes them secure and the ethical issues that surround them.

1.1.2 Encryption

It is sensible to start with a basic introduction to encryption, as this is the key to data security. Encryption refers to the process of encoding information in such a way that it cannot be intercepted by a third party unless it has a special key to decrypt it. Cryptography, on the other hand, is the study of these encoding techniques. There are two known types of encryption: symmetric and public.

Symmetric Key Encryption

This is the type of encryption, where the same key is used for both encrypting and decrypting a message. Due to that, the key's safety is a top priority so that the encoded messages do not get revealed. Advanced Encryption Scheme (AES) is a popular example of such an algorithm [1, 28]. The diagram below illustrates the method straightforward.

$$\text{Plaintext} \rightarrow \text{Encryption with key } k \rightarrow \text{Ciphertext}$$

$$\text{Ciphertext} \rightarrow \text{Decryption with key } k \rightarrow \text{Plaintext}$$

Public Key Encryption

Two keys are used for this technique: public and private. The public key can be given to anyone, but the private must be kept for yourself. It means that a pair of people can exchange public keys and communicate safely even if a man in the middle intercepts them. This is because both public and private keys are mathematically related - it allows the participants to communicate their public keys in the open initially, and then securely after the initial exchange using each individual's private key (p. 3) [28]. Imagine there are two participants, Alice and Bob, that would like to exchange a message securely, and Eve, who is an eavesdropper and wants to reveal their encoded information [28]. Their communication is illustrated in Figure 1.1.

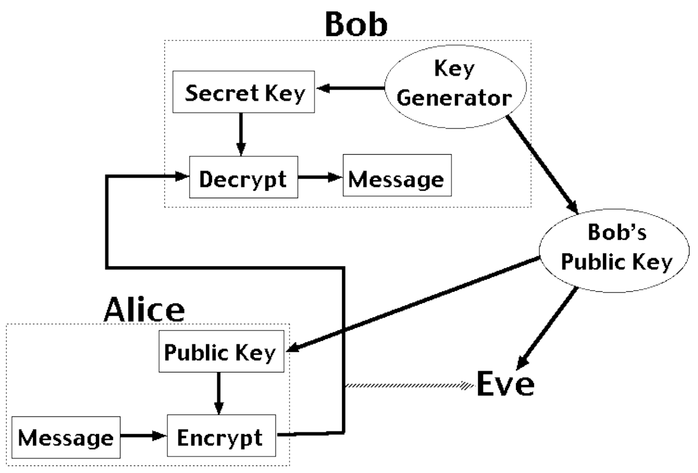


Figure 1.1: Public Key Communication [28]

Bob generates a pair of keys and shares his public one with to everybody, including Eve, but only he knows the private one. Thus, anyone can send him an encrypted message using the public key, but only Bob will be able to decrypt it using the secret (private) key. Since Eve does not know Bob's private key, she cannot easily reveal their communication, but with passive listening and consistency, she could use Bob's public key to decrypt Alice's messages using her own private key. This is one of the major public encryption bottlenecks that could be solved using digital signatures and certificates (will be discussed further).

1.1.3 Into the Deep Web

The Internet used on a daily basis is referred to as the clear/surface web. It is commonly accessed by the public for social networking, watching videos, reading news and much more. The Deep Web is anything that cannot be indexed by a search engine. In order to understand it better, I will give a popular example. If you imagine the Internet as an iceberg in the middle of the ocean, what you see above the water surface is the clear web. Everything below the water level, no matter how deep it goes, is known as the deep web. It is a subset of the Internet that is not accessible by major search engines such as Google and Bing. This is due to the Internet's nature of being too large to be completely covered by the engines and for security reasons. For example, databases for most of the web pages contain sensitive and confidential data, that cannot and should not be found by a search engine - this part of a website is the Deep Web (p. 2) [36]. *"With its myriad databases and hidden content, this deep Web is an important yet largely unexplored frontier for information search (p. 96) [31]."* Figure 1.2 shows a conceptual representation of the deep web site `bn.com/`.

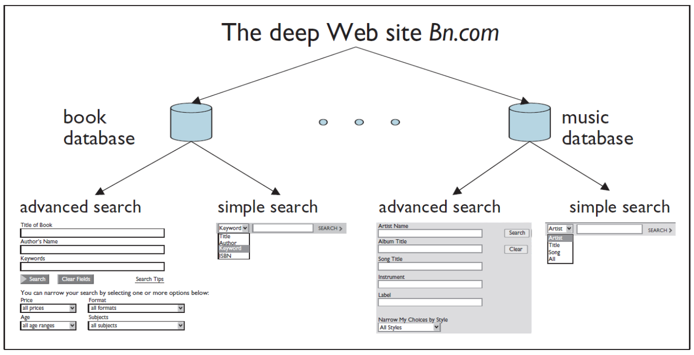


Figure 1.2: Site, databases, and interface (p. 97) [31]

As illustrated, the website provides several web databases (book and music given as examples), accessed through multiple query interfaces, including simple and advanced search. In addition, this proves that just because it is deep does not mean it is illegal. Most of the content is accessible but is kept to private via encryption. This is very useful when storing sensitive data. For example, when sharing your bank account details, its password-protected bits, such as card number, CVV, and address, are stored in the deep web. Due to the nature of the content, a lot of users surfing "below the water level" would prefer to stay anonymous. As mentioned, there are several methods to achieve anonymity with TOR browser being the most popular one [26].

According to the official Tor website, the Onion Router is an Internet networking protocol designed to anonymize the data relayed across it. It was originally developed by the U.S. Department of Defense for the purpose of protecting government communications, but it is now an open source volunteer-run project (p. 3) [36]. While browsing, a user cannot be located by an IP address, neither can data collectors like Google Ads perform traffic analysis on the user's internet habits [32]. It runs through computer servers of thousands of volunteers (relays) spread across the world. If a request for a page is given at a specific geographical position (e.g. Bath, UK), the page's server will see the request to be given from completely different location (e.g. New Zealand). Tor strips away part of the packet's header, which could be used to learn things about the sender such as the operating system from which the message was sent. The rest of the addressing information is encrypted via a packet wrapper. Figure 1.3 illustrates how Tor works in general.

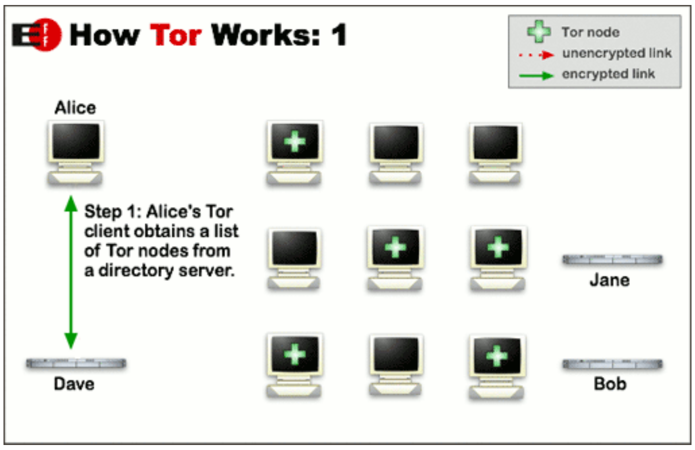

How Tor Works: 1
Tor node
unencrypted link
encrypted link
Alice
Step 1: Alice's Tor client obtains a list of Tor nodes from a directory server.
Jane
Dave
Bob

Figure 1.3: Tor Illustrated [26]

Each relay decrypts only enough of the data packet wrapper to know which relay the data came from and which one to send it to next. The relay then rewraps the package in a new wrapper and sends it on. Apart from government officials, individuals use the browser to prevent websites from tracking their location. Journalists, on the other hand, use it to publish more sensitive media and communicate safely.

> *The variety of people who use Tor is actually part of what makes it so secure. Tor hides you among the other users on the network, so the more populous and diverse the user base for Tor is, the more your anonymity will be protected [38].*

1.1.4 The Dark Web

It is generally considered the 'Deep' and the 'Dark' web to mean the same thing, whereas in fact, it does not. The Dark Web is a subset of the Deep Web and is restricted in addition to being non-indexed. It is the part of the Internet that cannot be accessed using a mainstream browser such as Google Chrome or Firefox. A proxy software such as Tor, I2P, and Freenet, is required. Figure 1.4 illustrates the different 'layers' of the Internet.

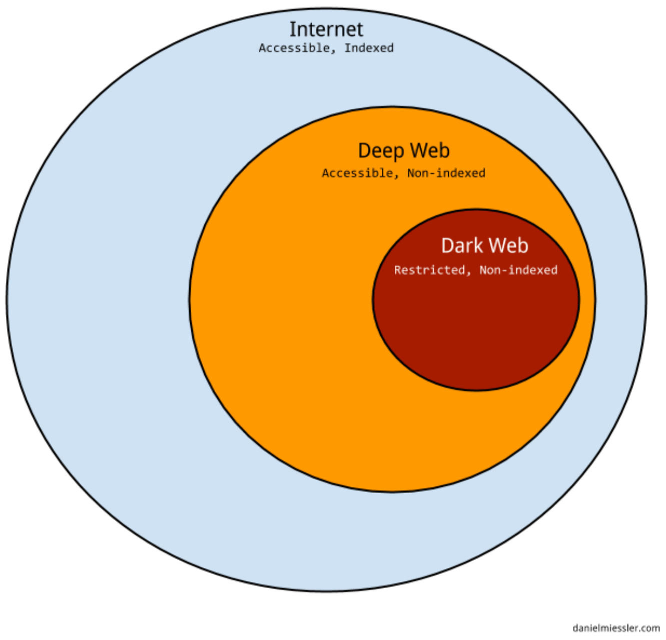


Figure 1.4: The Internet 'layers' [46]

The Dark Web is usually associated with criminal activities like drug dealing, arms supply, counterfeit documents, pedophilia, and CP (Child Pornography). Due to its nature of anonymity, such actions are easily carried out. However, legitimate uses are also present - it is much larger and more diverse than these illegal activities. According to Zach Brooke, the illegal services make up only about 1.5% of Tor's total traffic (p. 24) [32]. The key thing here is the concept of privacy - people are much more vulnerable in the clear web, where it is full of means to be tracked down and monitored by third parties. Examples include Flash player, cookies, IP addresses, and Geolocation. Everyone has the equal right to keep his/her identity hidden on the Internet for security reasons. It is a great medium for journalists, activists, and victims of domestic abuse, that would have consequences if they share anything in regards to it (p. 1223) [38]. This diversity of audience forms the so-called DWSN (Dark Web Social Networking). It is an equivalent of Facebook or Twitter, with excessive freedom due to anonymity (although not perfect). According to Rath, even *"Tor's executive director is working with victims of domestic abuse who need to communicate without being tracked by their abusers. Tor is also used by Chinese dissidents who can't access sites like Twitter. And it became a valuable tool during the Arab Spring. (p. 1223) [38]"*

The stated facts in the previous and the upcoming paragraph are summarized by Gehl in his book 'Power/Freedom on the Dark Web' [38]. The idea of social networking in the dark web, however, is not quite the same as what people are used to on social websites. Facebook, for example, is associated with people's identity, friends, messages, likes, and comments. Socialising on the dark side of the Internet is much more about freedom of speech without revealing your true identity. The DWSN's rules and regulations actually state that no one should reveal any personal information about themselves. While this facilitates the freedom of speech, it is technically a requirement that everyone must adhere to. It might sound like a paradox, but the DWSN does have its own TOS (Terms of Service) and privacy policy. This is what part of the 'Hacker's Manifesto' (i.e. Privacy Policy) in regards to identity states:

> *This is our world now - the world of the electron and the switch, the beauty of the baud We exist without skin color, without nationality, without religious bias and you call us criminals. Yes, I am a criminal. My crime is that of curiosity. My crime is that of judging people by what they say and think, not what they look like. (p. 1225) [38]*

The admin continues:

> *Tell me who you ARE not WHO you are. [DWSN] isn't all about anonymity, it's about soul. It's about putting a piece of yourself out there for the world to see that you otherwise would have been too hesitant to allow others to witness in a traditional setting. (p. 1225) [38]*

The admin's speech asserts that people should be judged by what they do instead of their age, gender, appearance, or sexual orientation. Just because somebody has decided not to reveal their true identity does not mean the person is a criminal. This is a statement I completely agree with. It is not a coincidence that *"Don't judge a book by its cover"* is my favourite English idiom.

1.2 Aims and Objectives

My project is research-based and the main focus is on the literature survey and references. My plans for implementation changed due course a couple of times. In the beginning, I wanted to do an improvement of an existing encryption algorithm, but it was out of my scope of knowledge and resources. Then I decided to do a real-time security measurement programme for Tor, but it turned out that there is an extension called "NoScript" that does it. Finally, I decided to work on "HTTPS Everywhere", an open source project between The Tor project and Electronic Frontier Foundation (EFF). Some of the aims and objectives below are outdated - they correspond to the research carried out, but not to the implementation and testing. This was my original line of thought in case you are interested.

Aims

1. Investigate the work behind the Deep Web and how it operates.

2. Research the encryption methods used within, what makes them so secure, gaps and areas for improvement.
3. Discuss the ethical issues around the 'Hidden part of the Internet'. - which bit of the deep web is considered illegal, how to browse safely without imposing law enforcement.
4. Implement a real-time programme that would measure the level of security while browsing in Tor.

Objectives

1. Discuss and research the Tor Project - history, aims, current state. (Aim 1)
2. Distinguish between the 'deep' and the 'dark' web - a common misconception. (Aim 3)
3. Relate and discuss case studies - history of cyber security, the Enigma Machine. (Aim 1)
4. Investigate the legal and ethical issues with anonymous browsing. (Aim 2)
5. Compare and differentiate the encryption methods for anonymity - public vs. private key encryption, cryptanalysis, algorithm efficiency and complexity. (Aim 4)
6. Investigate the cyber attacks and tricks performed in a technical aspect - SQL Injection, authentication bypassing. (Aim 2)
7. Evaluate the results from the attacks and based on them, measure the level of browser security and think of ways of improvement. (Aim 4)
8. Examine the use of cryptocurrency (e.g. Bitcoins) - how safe they are, is it worth having a Bitcoin wallet, currency mining. (Aim 3)
9. Experiment with modern encryption - SSL, HTTPS, Firewalls. (Aim 4)
10. Experiment with attacks and think of additional improvements to the real-time programme. (Aim 4)

Chapter 2

Literature Survey

2.1 Ancient History to WWII

The Dark Web topic is very broad and the focus will be on its technical aspects. This includes the encryption methods used by Tor, their pros, and cons and how could they be potentially improved. Much research needs to be carried out and the plan is to start with some background history and a glimpse into the secure communication methods in the past. Research began from classic encryption techniques used years BC to modern ones applied nowadays.

2.1.1 The Caesar Cipher

Also known as the shift cipher, it is one of the simplest encryption methods. It is named after Julius Caeser, who used it frequently to communicate with his allies. Each character in a message is substituted with a letter that is n positions down the alphabet. The number of possible encryptions using this cipher is *n-1*, where n is the number of characters in an alphabet - for example, the English alphabet has 26 letters, so the number of possible key encodings would be 25 (26 as a key would result in the original text). It is defined by the equations

$$E_n(x) = (x + s)\ mod\ 26$$
$$D_n(x) = (x - s)\ mod\ 26$$

where $E_n(x)$ is the encryption function and $D_n(x)$ the decryption respectively - x is each character in the plaintext and s is the number of shifts. For example, if we have the plaintext *"Hello, my name is John Doe"* and s=4, the ciphertext would be *"Lipps qc rwqi Nslr Hsi"*. This is one of the least secure encryption methods since it is very easy to guess the key. Even brute forcing is not really time-consuming and could happen in a matter of milliseconds if one has the technology to speed up the process [3]. Therefore, stronger encryption techniques had to be invented.

2.1.2 The One Time Pad

This is an enhanced version of the Caesar cipher. It has an element of randomness, which makes it immune to frequency analysis. In a message, every character is shifted a random amount. The result is a key with the same length as the original plaintext and random frequency distribution. A character can have 25 possible keys using the English alphabet, but the keyspace grows exponentially for each character in the message. It is defined by the formula N^m, where N is the number of letters in the alphabet and m is the number of characters in the message. So if we take the word 'spring' as an example, there will be 26^6 possibilities for message encryption. Every character is shifted a random amount, which is why the cipher is called 'One Time Pad' - the key cannot be used more than once due to the element of randomness. This method was a giant breakthrough for encryption and kept a really high-security level until the era of modern computers, which managed to break it. The technique's main downside is that the key size is identical to the length of the original message. Moreover, the key can be easily exploited unless a true random algorithm is used for its generation [13].

The encryption methods discussed above are quite easy to break, so much more have been developed throughout human histories, such as the Playfair, Transposition, and Four-Square cipher. They all have their strengths and bottlenecks, but the more interesting one that made a revolution in the area of data security and communication is the Enigma Machine.

2.1.3 The Enigma Machine

This is probably the most famous encrypting device in world history. It was used by the Germans during World War II for more secure message transmission. Most of the communication was done via radio, so encryption was vital in order the Allied forces not to understand their conversation. The encryption scheme was much stronger than older methods such as the Caeser and block cipher, the One Time Pad, etc. It remained unbroken for many years, until a fatal flaw in the system was discovered, which aided in the victory over the Nazis. More information on how the machine works and cracking the code can be found at A.1 and A.1.1.

2.2 Modern Encryption

With the Internet entering homes in the 90s, followed by its large commerce, data traffic started growing hugely. Data packets were constantly sent between devices, protocols and standards were being developed. The question was how can data be transferred securely so that it does not get tracked, revealed, or stolen. There needs to be some form of encryption to make sure that such cases are kept to a minimum. This section will consider some important modern encryption techniques and discuss why they are still used and what makes them stand above other methods.

2.2.1 HTTPS

HTTPS stands for “Hyper Text Transfer Protocol Secure”. The secure version of HTTP allows transmission of data using public key encryption. It runs on top of the TLS (Transport Layer Security) protocol and was generally meant for services like online banking, but gained popularity in the spheres of social networking, emailing, video streaming, etc. [48]. Websites using HTTP have a ‘certificate’ which proves that they are who they claim to be. To ensure the certificate’s validity, the server signs a message with its certificate’s private key and major web authorities such as the IETF (Internet Engineering Task Force) certify the authenticity of the connection.

High-level security is good but comes at a price. To start with, data passing through additional encryption layers makes traffic much slower - it is both time-consuming and computationally expensive. As much as the methods themselves protect the users, their energy consumption is huge, considering the additional handshake required on a client-server basis and all the mathematical calculations required (e.g. generation of cryptographic keys). In terms of speed, HTTPS is much slower than the HTTP and consumes more power, especially on devices using 3G connections (p.134, Introduction) [48]. In addition, the latency on the client side is very likely to go up. One of the solutions is using solid GPU architectures to reduce calculation time and latency, but creating this takes substantial time and resources (although manageable). No matter the costs, many Internet services have migrated to HTTPS services due to the much stronger security, aimed at guaranteeing end-users privacy. For instance, YouTube has been delivering more than 50% of its services on top of HTTPS as well as Facebook, which enabled HTTPS for all users in 2013 (p. 134, HTTPS Usage Trends) [48].

2.2.2 SSL and TLS

SSL (Secure Socket Layer) is a protocol designed to secure traffic on the Web but has recently been replaced by the TLS standard. It is a cryptosystem that uses both symmetric and public key encryption. This is the most commonly used cryptosystem and although many flaws have been discovered, its current version is secure enough. For example, SSL 2.0 lacked a client-server handshake authentication (both sides agreeing on what secret keys to use for communication), i.e. both sides did not need a certificate to verify that they are indeed the ones who they claim to be. This made the layer vulnerable to downgrade attacks, where a man-in-the-middle can modify the ciphers used in the handshake - if he makes them insecure, it would be easy to eavesdrop on communication and obtain sensitive data [43]. This is why the use of SSL 2.0 for establishing a client-server connection is prohibited by the TLS implementations [17].

The question here is how the certificates work. The TLS clients usually come with a list of trusted certificate authorities, integrated into the web browser used on his operating system. Basically, when a client connects to the server, the server provides a chain of certificates and if the web browser can recognise any of the certificates from the server, they are being verified by signing them in. Figure 2.1 illustrates in details how authentication is carried out.

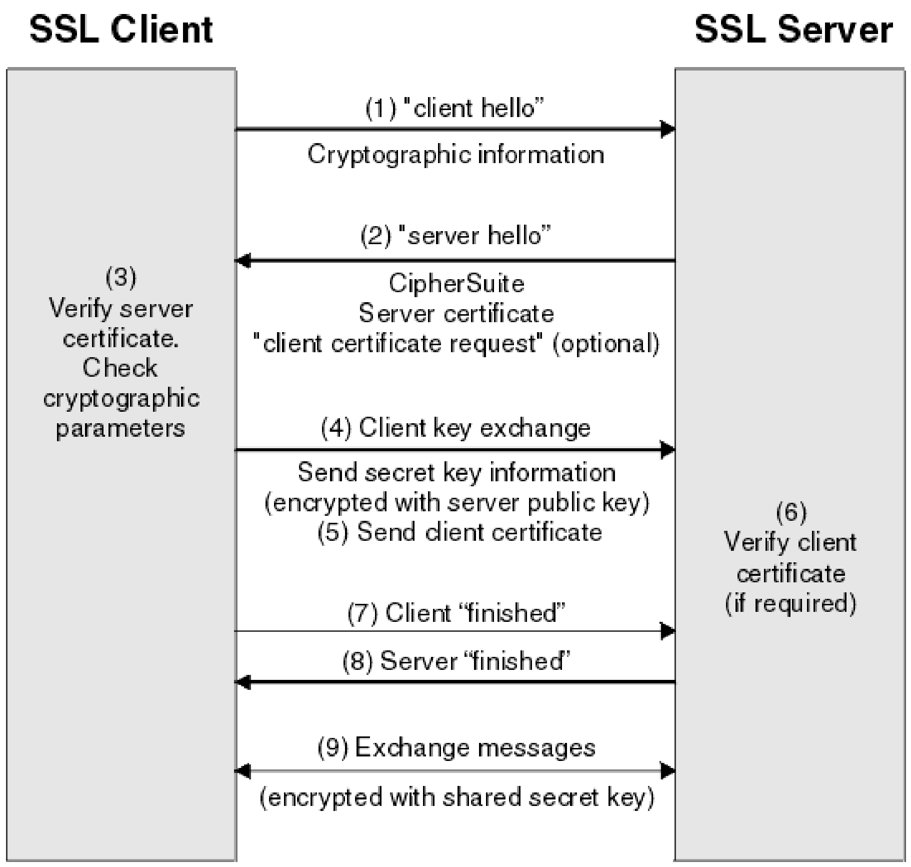

SSL Client
SSL Server
(1) "client hello"
Cryptographic information
(2) "server hello"
CipherSuite
Server certificate
"client certificate request" (optional)
(3) Verify server certificate. Check cryptographic parameters
(4) Client key exchange
Send secret key information (encrypted with server public key)
(5) Send client certificate
(6) Verify client certificate (if required)
(7) Client "finished"
(8) Server "finished"
(9) Exchange messages
(encrypted with shared secret key)

Figure 2.1: TLS Handshake [15]

In general, the client wants to send a message to the server, including the cryptographic information such as the version of SSL and TLS. The server then responds with a message that includes the server certificate. Once verified by the client, both sides exchange keys and share some secret key information. Finally, when an agreement is reached, both the client and the server can communicate. More details can be found in IBM's knowledge center site [15].

2.2.3 The Onion Router

This section expands on Tor, following the brief introduction in sections 1.1.3 and 1.1.4. Tor is a perfect example of a solid modern encryption, considering the fact that the browser has been properly functioning for 15 years now. Due to its strong level of security, it allows any form of activities on the Internet, including crimes, underground movements, and even CP. The fact that such activities are still going on and anonymity is dominating, makes Tor the perfect candidate for modern security.

To summarise, Tor is a web browser, written in C, intended to allow users to browse anonymously. This is achieved through several layers of encryption in the application layer. The IP address is nested deeply, just like the layers of an onion, which is where the browser's name came from. When a user sends a request for a website, it is sent randomly among several nodes (volunteering computer hardware) before reaching the destination server - this is the same technique as sending a request to a website in the clear web, with the difference that the nodes' IP addresses are not public and encrypted. Only the destination server is able to decrypt the content and once it does, sends a response back to the client in a similar fashion [14]. The Tor nodes can be operated either by an organization or by an individual. So the more nodes there are, the more stable and secure the connection is. A client request would pass through three relays before reaching its destination. The first and the middle relay receive data, unpack it just to see where to send it to and then re-wraps it again. Having such a relay is safe to run at home, even on your own machine, since it can never be suspected as the traffic source. The exit relay is the last one through which data passes before it reaches the server. The IP address of the final relay is interpreted as the source from which the request is given and would take the blame in case of illegal activities investigation. Therefore it would not be a good idea to run your PC as an exit relay as it is highly probable to attract the attention of law enforcement agencies. Instead, it should rather be a machine in a hosting facility that is aware that the server is running an exit node (e.g. a big corporation machine).

As always, solid encryption comes at a price. Tor is essentially an enhanced variation of HTTPS, which as discussed, is quite slow compared to a regular connection. And the fact that a Tor connection would pass through several nodes makes it even clumsier. The relays' security is vital for data traffic - sometimes individual nodes could be storing personal information, which is risky and removes any benefit of using Tor.

.Onion Domains

The Domain Name System (DNS) is a critical component of the Internet, that allows clients to match domain names with a site's IP address (p. 173, Introduction) [53]. Most of the websites accessed through Tor browser use ".onion" domains instead of ".com". This indicates that servers have the ability to host content without having an IP address, which would reveal their location, so tracerouting such websites is not possible (p. 173, Introduction) [53]. Neither the client nor the website server knows each other's IP address

- the only IP address known is the one of the exit relays. Moreover, Tor is decentralized by nature - websites do not have an official centralised directory, where a user can get a list of available services. The communication in Tor is concealed enough and every hop in the circuit reduces the chance of communicating peers being tracked through network surveillance, which would rely on the source and destination's IP address. The ".onion" websites' names are usually encrypted as well and end up for the user as a combination of random letters and numbers (e.g. *http://kpvz7ki2v5agwt35.onion* - a Wiki page containing some form of unofficial directory, i.e. list of deep web sites). The Tor browser does not allow history and bookmarks, so the site's domain has to be typed in every time access is required. As mentioned, it is thought that the primary usage of such an environment is to allow carrying out illegal activities. This is absolutely not true - its original purpose was to protect the U.S. intelligence communications online, but due to its ease of access, its usage has significantly expanded. The dark web does indeed have various types of websites for social networking, forums with a limitless scope of discussion topics, political propaganda, and much more - all of which is somewhat intriguing, eye-catching and interesting.

2.3 Cryptocurrency

Cryptographic currency, or Cryptocurrency, is an alternative to real-valued currencies (e.g. USD, GBP) that gained significant popularity in the past few years. It is a decentralized digital currency, not controlled by entities like banks, but instead are based on an open-source peer-to-peer IP. They allow instant and anonymous transactions between people using a system secured by a proof-of-work system, aided by advanced cryptography algorithms (p. 1700, Introduction) [35]. The most famous digital currency is the *Bitcoin* (BTC), introduced in 2009 by Satoshi Nakamoto. It is an innovative payment network and a new kind of money. As mentioned, there are no central authorities or banks - all transactions are carried out collectively in the network. Bitcoins allow exciting uses that could not be covered by any previous payment systems. To start with, there are no transaction fees - the currency's public design means that no one owns it, so everyone can take part with no costs. It is a global currency, so can be used in every country and bought with any currency. Every user has a digital wallet and transactions are recorded in a transparent public database. Nowadays, an increasingly larger number of websites and companies integrate the use of Bitcoins in their systems [12].

2.3.1 Why Bitcoin?

Bitcoins have several advantages compared to official currencies. The most notable are:

- **Decentralized services and reduced fees -** Bitcoins are not stored in a central repository, so fees are significantly reduced. The system is peer-to-peer based, which makes transactions much faster than the normal ones.

- **No personal data storage -** Although all transactions are stored in a single huge database, privacy and security are enforced with encryption techniques. Every user has one or more wallet addresses as well as a private key used for transaction authorisation. Instead of personal information, Bitcoin uses a distributed ledger book system called "blockchain" [42]. When transactions are made, they are sent to a particular unique wallet address, not to a specific person. So the only identifying information, in this case, are the random characters contained in an address. The benefit is that a user can always change his/her address for every transaction and this is actually preferable, since using the same one repeatedly would act as a link to the person in the long run.
- **Transaction verification validates security -** Transactions are verified through "mining", which is one of the methods of producing Bitcoins. In short, Bitcoin miners check the strength of a transaction's encryption to see if it can be verified. After each verification, miners are rewarded Bitcoins. Due to the rewards, Bitcoin mining has turned into a highly demanded profession. This is to some extent good for Bitcoin users, because the more miners, the more secure and resistant are the blocks to an attack. Invoking an attack would mean that the attacker has to hash a transaction faster than the entire network, which is almost impossible, considering that it produces around 1500 floating point arithmetic operations per second. So unless there is a malicious software that can break the encryption algorithm and outspeed the network, using bitcoins is generally safe in that respect [42].

2.3.2 Disadvantages of Bitcoin

Although security is always kept at a decent level, Bitcoins have their disadvantages, the most fundamental being:

- **Volatility -** Bitcoin is a very unstable currency due to its young economy, i.e. its value might change rapidly and unexpectedly. For example, in November 2015, the BTC's value was around \$321.00, while a year later, it rose to \$737.16, which is close to a 150% increase [2]. Being so volatile means that keeping Bitcoin savings, in the long run, is very risky, so it would be better to buy the exact amount of Bitcoins required for a purchase instead of investing thousands, which could have serious consequences.
- **Irreversible payments -** Payments can only be refunded by the person receiving the fund. This means that one should only make transactions to trustworthy sources with a good reputation - sending money to unverified entities/organisations are not recommended.
- **Identity link to an address -** One should be careful with accidentally linking themselves to a wallet address. Manually or automatically, it is strongly recommended to always change the address for every transaction carried out. Unfortunately, this is something that requires a decent knowledge of technical skills, which are not possessed by all conventional users (p. 43, Weaknesses of Bitcoin) [42].

- **Government taxes and regulations -** Bitcoin is a first-generation cryptocurrency and is reliant on third parties for establishing trade and purchase. These parties are often immature enough to really offer what the Bitcoin protocol does. It is not an official currency but has a value. That said, it is a lawful requirement to pay taxes on everything with a value. Although transaction fees are significantly reduced, taxes can still be an issue.
- **Legal issues -** This is another important aspect to mention. Due to Bitcoins' nature, it facilitates the illegal trade in the black market of drugs, weapons, hacking services, etc. Essentially, using Bitcoins is not illegal - it is a decentralized peer-to-peer system, just like paper cash. What they are used for has nothing to do with the way they work. The problem is that its direct 'relation' to the black market attracts media that can create false ideas to the public of what cryptocurrencies actually are [42].

Our economy is rapidly changing and so are Bitcoins. Their reliability and security in the future will certainly increase and the issue of volatility will be brought down to a minimum.

2.3.3 Nature and Ethics of Cryptocurrencies

This section was planned to be expanded after the literature review submission, but due to time restrictions, I could not do any further research.

2.4 Web Crawlers

Web crawling is an important topic for my thesis, as part of my development is based on it. Web crawlers, also known as spiders, are programmes for exploring web applications automatically (p. 40, Section 1) [47]. It is a client-server interaction that allows a user to fetch web content. Search engines are examples of web crawlers. More information on crawlers can be found at [49] by Cambridge UP.

I did some background reading but unfortunately did not have enough time to reflect it in the thesis. Yet, the most important references I have gone through and are worth looking up are listed below.

- **Deepbot Crawler** - a useful guide and tips to writing a crawler engine for accessing a webpage's hidden content [45].
- **Incremental Web Crawler** - an outline of a web crawler design by IBM with a focus on crawling strategies [37].
- **Not so Creepy Crawler** - writing a web crawler using standard XML queries [54].
- **Web Crawler Ethics** - a very important paper. Ethics in web crawling are a major concern as crawlers keep getting highly automated and seldom manually regulated.

> The article discusses the extent to which web crawlers respect the regulations set forth by websites in their *robots.txt* file [39].

Some ideas were taken from these papers while I was working on my crawler.

Chapter 3

Requirements

The implementation of the project will involve writing sufficient number of rulesets through a web crawler or manually to websites not automatically redirecting the client to their HTTPS version. Thorough testing using the automated checker will be performed. All of the approved rules will be counted as an individual contribution. Requirements' priorities are indicated with the words MUST, SHOULD, and MAY.

1. The individual contribution must be explicitly shown and clear.
2. Websites not supporting HTTPS server should still be connectible securely by the client - this depends on the browser extension settings. There is an option to enable/disable unencrypted requests.
3. All websites that have rulesets must be in the "HTTPS Everywhere" atlas.
4. The rulesets should be tested to work on both Chrome and Firefox.
5. If there is enough time, rulesets may be tested on Tor browser for additional functionality.
6. The code should be efficient and easy to read according to the style guide.
7. Any major changes and tests should be documented and reflected.
8. Credits should be given to the Tor community and EFF as well as any other entities involved in committing changes.

3.1 Requirements Breakdown

This section justifies the choice of requirements, explaining why each is necessary.

- **Individual contribution -** HTTPS Everywhere is an open-source project, so it is important to stress on personal contribution. In my case, it is the number of rulesets written by me.

- **HTTP connection -** When the extension is active, connection to sites with no HTTPS support has to be accessible. The add-on has a tickbox that blocks all unencrypted requests. It is a matter of personal choice to decide whether to all block HTTP requests or just when you do not need any.

- **Domain names in the Atlas -** If a domain name is in the atlas, it means that it is affected by the HTTPS Everywhere rules and redirects the user accordingly.

- **Test on Chrome and Firefox -** The browser extension is available for Chrome, Firefox, Android, Opera (unstable), and Tor. Firefox and Chrome are the most frequently used browsers out of the five and rulesets should at least work on them in order to handle as many user requests as possible.

- **Test on Tor -** HTTPS Everywhere is available on Tor browser and it is a plus to have rulesets for ".onion" domains to ensure better security in the deep web.

- **Code styling and efficiency -** Rulesets have specific styling conventions that need to be followed for best practices. Thus they will be easier to read, modify, and less likely to crash or not work properly.

- **Documenting changes and tests -** There are multiple testing scripts for rulesets, but in case I decide to do my own tests, it needs to be documented, explaining why I came up with such decision and what are the benefits from it.

- **Credits to EFF and Tor -** Thanks to the official Tor website, I found out about the "HTTPS Everywhere" project. EFF deserves credits for developing the project and allowing users to contribute, thanks to which I did a big part of my thesis.

Chapter 4

Implementation and Testing

This chapter focuses on the implementation plans based on some technology background. The idea, in the beginning, was an improvement of an existing encryption algorithm used within the Tor browser. However, as the research progressed, some very interesting and relevant papers changed this idea. Currently, the major problems with Tor are the speed and traffic leakage in the global DNS. The plan is to investigate and discuss the currently used algorithms, their bottlenecks, express a view on the big picture and propose a solution. It would make more sense to create a real-time programme, that measures the level of security while a user is browsing in Tor. Creating a new algorithm would take a huge amount of time and it is not certain whether the implementation would be any better than the existing ones.

After doing some research, it turned out that Tor has a built-in add-on called "NoScript" that performs real-time security tracking. It has the option of disabling any plug-ins, establishing new identities on every connection as well as displaying different types of warnings. However, I did a substantial amount of research for making a real-time programme before my final decision for development. The technical background and examples can be found in section A.2 in the Appendix.

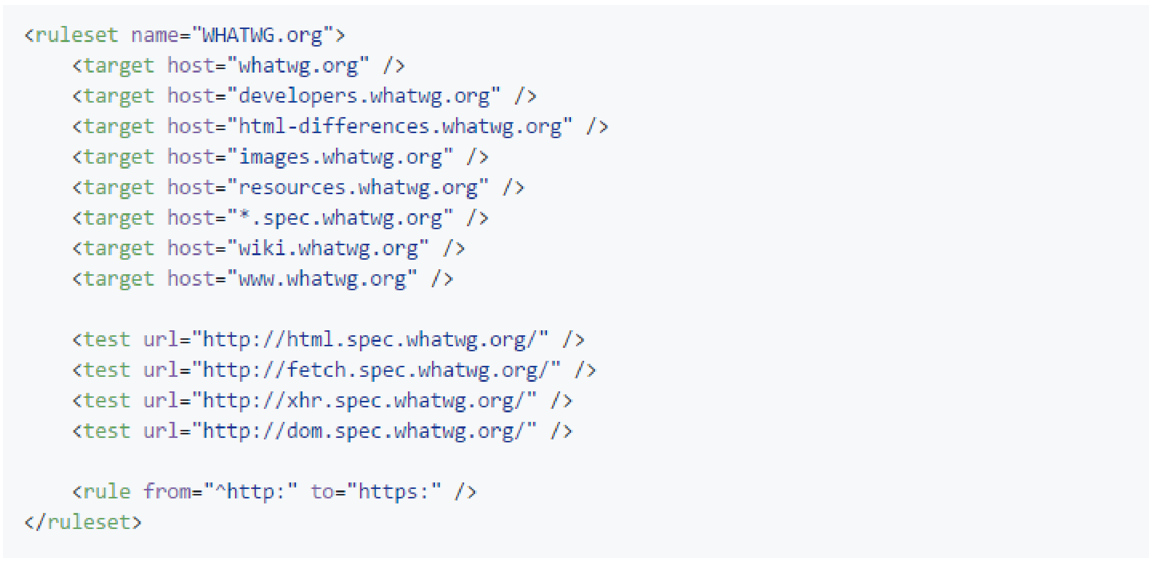
At the end, I decided to work on an open-source project called "HTTPS Everywhere".

4.1 HTTPS Everywhere

"HTTPS Everywhere" is a collaboration between the Tor community and the Electronic Frontier Foundation (EFF). It is a browser extension (or add-on) that enables secure client-server communication with websites. This is achieved through rulesets, written in XML. Many websites nowadays do not provide safe connection, making the user vulnerable to data traffic analysis and risk of sensitive information leakage. The extension cannot ensure that the website server has an SSL certificate, but if present, can encrypt all of its client connections [8].

The project is open-source and uses Git repository to allow commitments from multiple entities. The rulesets are XML files that describe the behaviour of a website. Regular expressions are used as a primary means of logic - this allows covering most of the sites' domains and subdomains (with the help of wildcard symbols). In addition to covering the site's subdomains, it is strongly advisable to secure images as well as any existing external links that would send the user to another website - this is done via test URLs. There is a certain style guide to be followed and a set of requirements need to be fulfilled before making pull requests.

Testing is performed with an automated checker that runs some basic tests on all rulesets. More information about the testing and the project, in general, can be found in the "HTTPS Everywhere" website [9]. A ruleset example is illustrated below.



```xml
<ruleset name="WHATWG.org">
    <target host="whatwg.org" />
    <target host="developers.whatwg.org" />
    <target host="html-differences.whatwg.org" />
    <target host="images.whatwg.org" />
    <target host="resources.whatwg.org" />
    <target host="*.spec.whatwg.org" />
    <target host="wiki.whatwg.org" />
    <target host="www.whatwg.org" />

    <test url="http://html.spec.whatwg.org/" />
    <test url="http://fetch.spec.whatwg.org/" />
    <test url="http://xhr.spec.whatwg.org/" />
    <test url="http://dom.spec.whatwg.org/" />

    <rule from="^http:" to="https:" />
</ruleset>
```

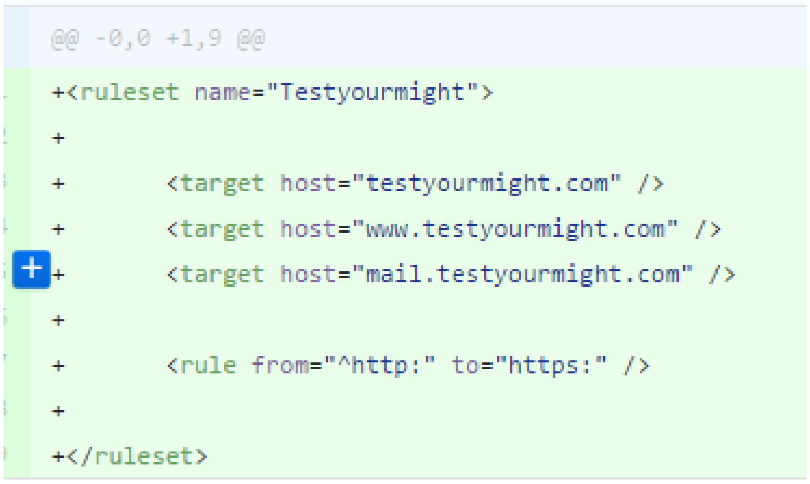
Figure 4.1: Ruleset Example

4.1.1 Development of HTTPS Everywhere

Rulesets are written in XML and need thorough testing before making pull requests to the master branch. Initially, the repository is forked to my account on Github and cloned locally. Library dependencies such as selenium, lxml, python-Levenshtein and regex need to be installed. This is facilitated through a shell script called *install-dev-dependencies.sh*. The next step is to run all tests for common rulesets mistakes in a standalone Firefox and Chromium profile. Finally, extensions are built for Firefox and Chromium, so that testing validation is possible for both web browsers.

Rules can either be manually created or via an executable that has a ruleset template. There are convention standards to be followed for best practice as well as a guide on how to efficiently perform testing. Overly complex regular expressions and left wildcard targets are not desirable since they do not always capture all of the website's subdomains. Once all tests have passed, changes can be pushed to the remote branch. The repository contains some scripts for automated testing for individual and combined rulesets.

I used a virtual machine running Ubuntu 16.04 - Linux was easier to work with due to the nice and easy setup of dependencies and libraries. I started by writing a sample ruleset for the website `testyourmight.com`. The tests failed the first couple of times, but fixed them soon after, so my pull request to the master was merged. From now on a user is automatically redirected to the HTTPS version of the website any time it is requested. My idea is to create a crawler that would have a list of websites and create rules for those that are not automatically redirected to their HTTPS version. The first ruleset written by myself looks like this:



```
@@ -0,0 +1,9 @@
+<ruleset name="Testyourmight">
+
+       <target host="testyourmight.com" />
+       <target host="www.testyourmight.com" />
+       <target host="mail.testyourmight.com" />
+
+       <rule from="^http:" to="https:" />
+
+</ruleset>
```

Figure 4.2: Ruleset for *testyourmight.com*

4.2 Web Crawling Development

I created a web crawler that retrieves a list of domains for gaming websites and checks which ones support HTTPS. Once obtained, rulesets for all websites without automatic HTTPS redirection, are created. The language of choice was Python for a couple of reasons:

1. Python supports libraries that scrape web pages really well, such as *Lxml*, *Selenium*, and *BeautifulSoup*.
2. Excellent support for HTML parsing and regular expressions.
3. For the project purpose, it was easy and fast to extract domain names, as this was all the information needed.
4. I had prior experience with Python both from university and placement year.

I started by testing a simple web crawler for extracting hyperlinks from a website's page. It is a "traditional" crawler that obtains a list of websites and manipulates them accordingly (p. 42, Table 1) [47]. The idea is broken down in the next paragraph.

Libraries:

- *Urllib/Urllib2* - The most important module. It opens a communication link with a URL. Once established, it returns information about the link, usually the website's HTML header. *HTMLParser* is a library that extracts all the text from a webpage using your own parsing specializations. The parser contains submodules to preserve the structure of the original HTML and then search the data needed.
- *BeautifulSoup* - Very practical, powerful, and flexible, best used to extract data from static webpages with the help of parsers.
- *DryScrape* - An open-source Python library used for scraping Javascript-heavy webpages. Useful for my crawler's purposes. Documentation can be found at [18]
- *Selenium* - Another Python library for dynamic webpages that opens a browser and allows to wait for a specific amount of time until a condition is met before returning results. Documentation: [24]

Ideas and experiments:

- The crawler extracts all the *href* links from a given webpage.
- The main point is to generate a list of domains that support HTTPS and create rulesets for the sites not automatically redirected to their HTTPS versions. Getting such list usually requires budget, but I managed to find a way to do it without using any. I got a list of domains from multiple gaming websites.
- HTTPS Everywhere has an atlas of all domains that have rulesets implemented. Ideally, comparing a list of HTTPS supported websites with the ones in the atlas is an optimal method to check which sites need rules.
- An option for checking HTTPS support for websites is via an SSL checker at `https://www.sslshopper.com/ssl-checker.html/`, which diagnoses any problems with a site's SSL certificate. An input is given from the list of gaming websites and wait to return results.

Common Issues:

- Robustness - HTTP error code 403, 404, etc. This happens either when crawlers are banned by the host website or are recognized as malware and denies access. Can be fixed by writing a more efficient code and authenticating to the *robots.txt* file of the given website with a valid signature.

- Lack of HTTPS Support - If a website does not support HTTPS, nothing can be done using the browser add-on, unless it is spoken to the site host, but this was out of the project scope.
- Loss of Internet Connection - Crawling took a considerable amount of time and if the connection was lost, it terminated and had to be run again. Having a solid ethernet connection minimised the risk but was not always possible due to hardware limitations.
- Timeout - For some websites, checking HTTPS support took too long. I have set the limit to 40 seconds for time efficiency purposes. If returning results took more than 40 seconds, I assumed that the site did not support HTTPS and therefore did not shortlist it. Running the crawler on a server and multithreading could increase efficiency, but time was insufficient.
- Others - Errors such as directing to a different domain when trying to open a website have occurred. Due to time constraints, I was not able to fix them.

4.2.1 Functionality

The web crawler functions the following way:

1. Obtain a set of domains from a specific webpage.
2. Check if each website supports HTTPS via `https://www.sslshopper.com/ssl-checker.html`.
3. Compare the websites that support HTTPS with the "HTTPS Everywhere Atlas", found in `https://www.eff.org/https-everywhere/atlas/`.
4. Create a ruleset for each website not in the atlas.
5. Modify or extend the rules for certain websites containing several subdomains (including exceptions and mixed content blocking).
6. Test the rulesets - if they pass all the requirements, make pull request to the master branch and wait for approval.

4.2.2 Diagrams

Flow Chart

The flow chart represents the flow of logic in my crawler.

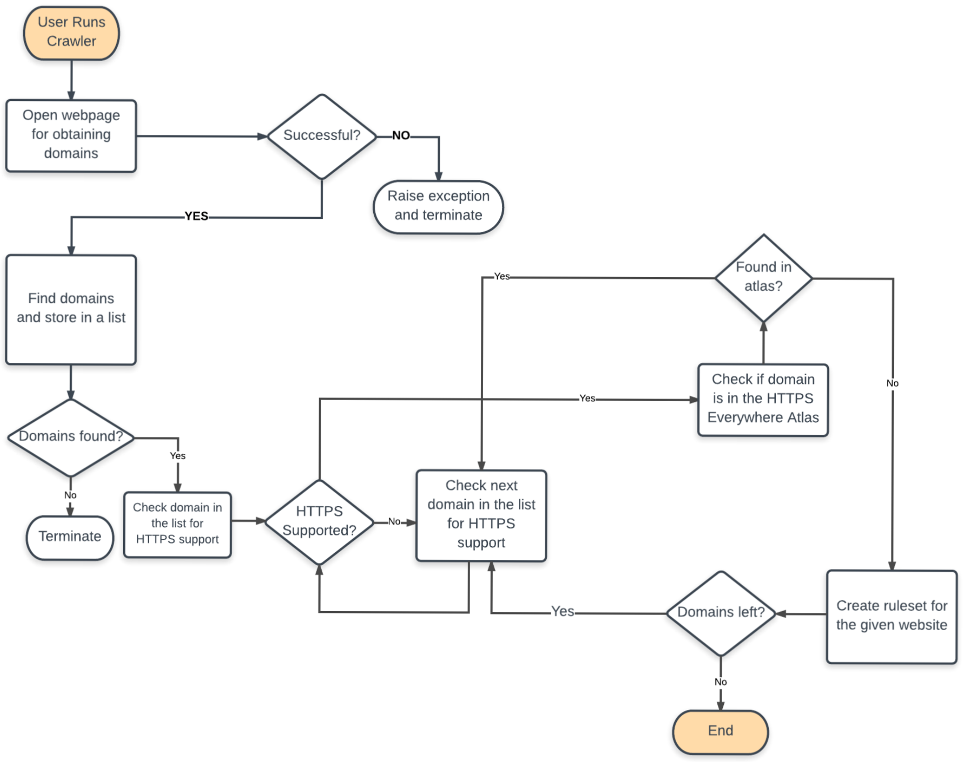

User Runs Crawler
Open webpage for obtaining domains
Successful?
NO
Raise exception and terminate
YES
Find domains and store in a list
Domains found?
Yes
No
Terminate
Check domain in the list for HTTPS support
HTTPS Supported?
No
Check next domain in the list for HTTPS support
Yes
Check if domain is in the HTTPS Everywhere Atlas
Found in atlas?
Yes
No
Create ruleset for the given website
Domains left?
Yes
No
End

Figure 4.3: Flow chart of the web crawler

Processes like checking the domains are split into two for better visualisation and to indicate that it is a loop process. The diagram does not directly represent the logic in my code.

Sequence Diagram

The sequence diagram illustrates the high-level communication between the actor (Client) and the entities (Server and Internet/EFF databases).

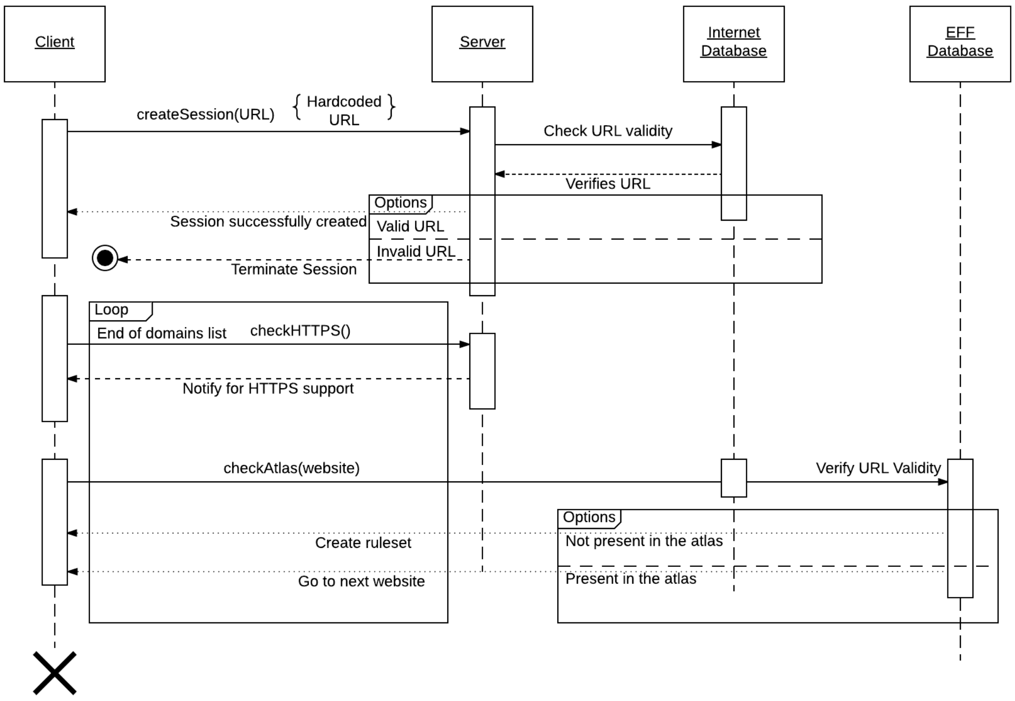


Figure 4.4: Sequence diagram of the web crawler

The figure shows the actions performed in a timeline, specifying the methods used. This is a more realistic representation of my code architecture.

4.2.3 Crawler Code Breakdown

My crawler consists of three main functions. The first one crawls a website to obtain domains. The second one checks each domain for HTTPS support and the third creates the rulesets.

It starts by opening a dryscrape session and scraping through an arbitrary webpage. Domain names are then extracted and stored in a list. Each domain is checked for HTTPS support using Selenium in `http://www.sslshopper.com/ssl-checker.html`. *PyVirtualDisplay* is used to hide the browser and save some time. The ones supporting HTTPS are shortlisted and compared to the HTTPS Everywhere atlas to check if they already have a rule using *BeautifulSoup*. For the ones that do not, rulesets are created through bash command *bash ./make-trivial-rule sitename.com* using the *subprocess* library that allows execution of console commands in a terminal.

Once created, rules are manually modified to handle all subdomains for every website as long as they support HTTPS. The project has some scripts for automated rules testing in Firefox and Chromium. Each ruleset can also be tested separately via *./fetch-test.sh Rulesetname.xml.* Once all tests have passed the criteria for formatting and functionality, changes were committed to my remote repository. Finally, pull requests are opened to the origin master and wait for approval.

4.2.4 Prerequisites

Some prerequisites are required in order to run the web crawler.

1. **Python 2.7** (might not work on 3.4).
2. **HTTPS Everywhere -** The extension can be downloaded from [8] for both Chrome and Firefox. Make sure to tick "Allow access to file URLs" to allow testing.
3. **BeautifulSoup v4** - bs4.
4. **Lxml**, **Xvfb**, **QtWebkit 5**
5. **Selenium 3.4** or greater [24].
6. **Pyvirtualdisplay -** A python library that creates a virtual display without necessary been visible. I have used it to save time when opening webpages with *Selenium* as it opens the browser by default and waits for the page to fully load before returning any results.
7. **Dryscrape -** A dynamic webpage scraping library, downloaded from [18]. No official support for Windows, but should work with cygwin.
8. **Geckodriver -** A proxy for using W3C WebDriver-compatible clients to interact with Gecko-based browsers. This program provides the HTTP API described by the WebDriver protocol to communicate with Gecko browsers, such as Firefox. It translates calls into the Marionette automation protocol by acting as a proxy between the local- and remote ends. Can be downloaded from [7].

If using Linux, make sure that the path to the geckodriver is exported in advance. This is done by typing *export PATH=$PATH:/path/to/geckodriver* in the command line. It might need to be done once a day - it is inconvenient, but time was running away, so the issue could not be handled.

4.3 Personal Contribution

The section focuses entirely on my personal contribution for "HTTPS Everywhere". The web crawler was for personal usage in order to create rulesets faster. There are currently

26 rulesets created by me - some of them are already merged, others are about to be. The exact time of merging depends on the project collaborators when they have time to check them. Pull requests rarely drop below 400, so it might take some time. A list of personally created rulesets for websites can be found in section B.6 in the Appendix.

Once merged, each ruleset needs 24 hours to appear in the Atlas. They ship and take effect in the next release of HTTPS Everywhere, which takes place when sufficient number of new rulesets are added. They can be viewed in my remote repository at [16].

4.4 Issues, Limitations, and Fixes

Throughout the implementation, I faced several issues. They are split into three categories - general, project-specific, and technical.

4.4.1 General

- **Natural time limitation -** We had roughly seven months to complete the project. During that period, I was busy studying for other modules and applying for graduate jobs. More work could be done if there was a whole semester dedicated to the dissertation or had fewer modules for the academic year.
- **Creating new repository -** While I was working on a forked repository and learning how to use Github in parallel. In the beginning, I did not have much experience and played around with commands, commits, and pull requests. At some point, it became a mess, even though I tried to clean things up. There were some pull requests that were disapproved and did not pass the Travis CI build and eventually I decided to delete my current "HTTPS Everywhere" repository and fork it again. All of the rulesets created were backed up beforehand, but doing the process all over again took away some valuable time.
- **Open-source project issues -** The advantage of such projects is that they allow contribution from many people. However, there are often issues with the documentation and setup. The script for installing library dependencies was not very well written as some of the downloaded packages were outdated. Furthermore, it was unclear on how to proceed with the testing and not following a correct order could lead to an absolute mess. Some of the prerequisites required to run the tests such as gecko- and chromedriver were not mentioned at all. In addition, the parameters passed to the chromium script did not seem quite right and caused occasionally to crash. This was the cause of not being able to push to my remote repository before removing the hook.
- **Using virtual machine -** Since most of the libraries and dependencies had conflicts with Windows, I decided to use Linux for my implementation. I created a virtual machine (VM) using VMWare workstation. Obviously, using a VM is much slower

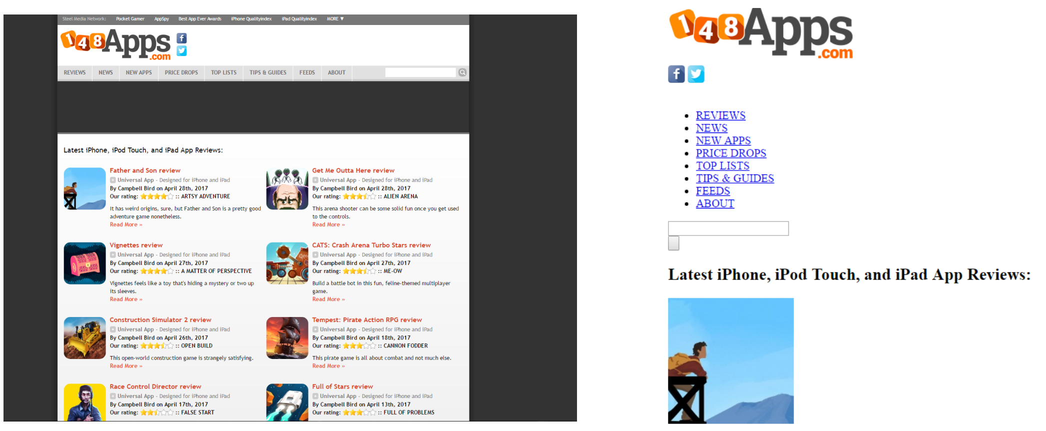
than a normal operating system and this slowed the whole process down. Amazon Web Services (AWS) would probably have been faster. Alternatively, using an external hard drive with an OS could do the job or just play around more to configure the project to work properly on Windows.

4.4.2 Project-specific

- **Handling multiple subdomains -** The command *bash make-trivial-rule sitename.com* creates a trivial XML rule file with just basic target hosts, i.e. `sitename.com` and `www.sitename.com`. Most websites have more subdomains than the two basic ones. For example, the fighting games focused site `shoryuken.com` contains subdomains like *evo, forums, rank, and wiki* that need to be handled. And since subdomains are website-specific, each ruleset had to be manually modified.

  The process was time-consuming, but fortunately, a tool called *Sublist3r* was available to use. This is a python tool made by Ahmed Aboul-Ela, designed to enumerate subdomains of websites through OSINT. Enumeration is achieved using search engines like Google, Yahoo, Baidu, etc. [19]. The script uses brute force with an improved word list to find as many subdomains as possible. It turned out to be very useful, as it saved me a lot of time and allowed me to handle most of the websites' subdomains.

- **Mixed Content Blocking (MCB) -** Some rulesets might trigger active mixed content (i.e. scripts loaded over HTTP instead of HTTPS). Such content is blocked in both Chrome and Firefox before HTTPS Everywhere has a chance to rewrite the URLs to an HTTPS version. This generally breaks the site or makes it look different to the original with few or no scripts loaded. MCB is only allowed on Tor browser. This is generally solved by setting by adding an attribute to ruleset element *platform="mixedcontent"*. Figure 4.5 shows an example of mixed content blocking.

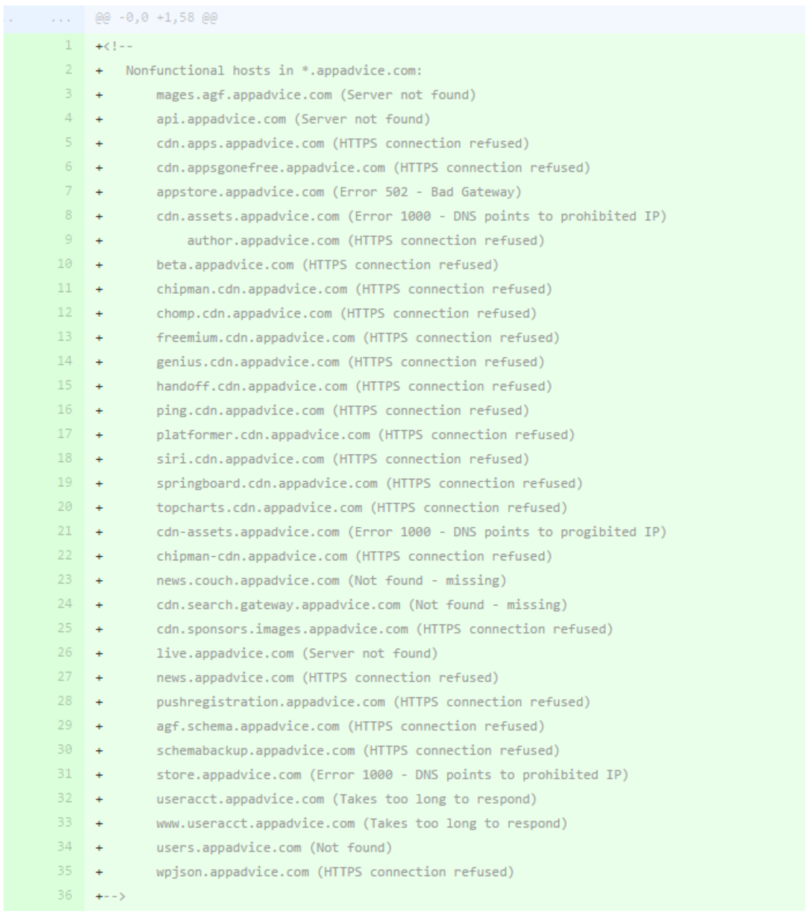


(a) No HTTPS

(b) With HTTPS

Figure 4.5: MCB issue in `148apps.com`

- **Outdated domains and rulesets -** HTTPS Everywhere contains thousands of rulesets written by different entities. In order for a user to make pull requests, all rules have to be tested in standalone Chrome and Firefox profiles. However, some rules contained expired domains and others no longer existed, which caused problems during testing.
- **Nonfunctional hosts -** Some websites have subdomains that are nonfunctional for various reasons. Some may refuse HTTPS connection, others have SSL certificate mismatch. Error 403 (Forbidden), 404 (Not Found), 502 (Bad Gateway), and 1000 (DNS points to prohibited IP) were frequent. It was a requirement (or at least good practice) for each nonfunctional host to be displayed with an indicated error type. Figure 4.6 shows an example of such hosts.



```
@@ -0,0 +1,58 @@
+<!--
+   Nonfunctional hosts in *.appadvice.com:
+       mages.agf.appadvice.com (Server not found)
+       api.appadvice.com (Server not found)
+       cdn.apps.appadvice.com (HTTPS connection refused)
+       cdn.appsgonefree.appadvice.com (HTTPS connection refused)
+       appstore.appadvice.com (Error 502 - Bad Gateway)
+       cdn.assets.appadvice.com (Error 1000 - DNS points to prohibited IP)
+           author.appadvice.com (HTTPS connection refused)
+       beta.appadvice.com (HTTPS connection refused)
+       chipman.cdn.appadvice.com (HTTPS connection refused)
+       chomp.cdn.appadvice.com (HTTPS connection refused)
+       freemium.cdn.appadvice.com (HTTPS connection refused)
+       genius.cdn.appadvice.com (HTTPS connection refused)
+       handoff.cdn.appadvice.com (HTTPS connection refused)
+       ping.cdn.appadvice.com (HTTPS connection refused)
+       platformer.cdn.appadvice.com (HTTPS connection refused)
+       siri.cdn.appadvice.com (HTTPS connection refused)
+       springboard.cdn.appadvice.com (HTTPS connection refused)
+       topcharts.cdn.appadvice.com (HTTPS connection refused)
+       cdn-assets.appadvice.com (Error 1000 - DNS points to progibited IP)
+       chipman-cdn.appadvice.com (HTTPS connection refused)
+       news.couch.appadvice.com (Not found - missing)
+       cdn.search.gateway.appadvice.com (Not found - missing)
+       cdn.sponsors.images.appadvice.com (HTTPS connection refused)
+       live.appadvice.com (Server not found)
+       news.appadvice.com (HTTPS connection refused)
+       pushregistration.appadvice.com (HTTPS connection refused)
+       agf.schema.appadvice.com (HTTPS connection refused)
+       schemabackup.appadvice.com (HTTPS connection refused)
+       store.appadvice.com (Error 1000 - DNS points to prohibited IP)
+       useracct.appadvice.com (Takes too long to respond)
+       www.useracct.appadvice.com (Takes too long to respond)
+       users.appadvice.com (Not found)
+       wpjson.appadvice.com (HTTPS connection refused)
+-->
```

Figure 4.6: Nonfunctional hosts of `*.appadvice.com`

A ruleset handling the issue is captured in B, image B.2 in the Appendix.

- **Extension error -** When opened, Chromium loads without any extensions by default. To fix it, the access file URLs in the extension options had to be enabled in order to do the tests, which took me some time to figure it out.

- **Pull request merging -** When making a pull request, it needs to be verified by *Travis CI Build*, which performs testing on Firefox and Chromium. It is automated and once all checks have passed, it is ready to be merged. However, merging requires a moderator to look at the request and approve it. No matter how many rules are created and pass the validity checks, they will not matter unless the pull request is merged and closed. Fortunately, editors seemed to be accurate and performed daily checks.

4.4.3 Technical

- **Generating a list of domains -** My initial idea was to grab some domains from `registered-domains-list.com`, but it turned out that the newly registered ones were either for sale or did not support HTTPS at all. Thus, they did not do any work for the project. At the end, I switched to making a list of gaming-related websites - there are thousands of that kind, many of which have HTTPS support.

- **Check for HTTPS support -** Checking for HTTPS support in `https://www.sslshopper.com/ssl-checker.html` was another major issue. The page is dynamic, i.e. contains scripts executed at runtime. My crawler was not able to find the tables containing the necessary information for HTTPS support. It turned out that *BeautifulSoup* and *urllib* do not have Javascript support, so can only scrape the static HTML code of a webpage. It took me some time to figure out a solution, but *dryscrape* seemed to do the job. This is a lightweight web scraping library for Python created by Niklas Baumstark. It uses a headless Webkit instance to evaluate Javascript on the visited pages. This enables painless scraping of plain web pages as well as Javascript-heavy Web 2.0 applications such as Facebook and other dynamic webpages [18]. It works for Mac OS X, Ubuntu, and Arch Linux.

  When visiting a URL, *dryscrape* waits for the page to fully load before returning the HTML body. After a couple of experiments, I found some memory leak issues. Every time a session is created, it spawns a new webkit server that does not get killed unless specifically stated at the end of the script. Thus my crawler got slower after every iteration and eventually threw a TimeOut error. Fortunately, this was quickly found and fixed.

- **Selenium Python versus Dryscrape -** Dryscrape seemed to have some issues of waiting for a dynamic web page to load. Its documentation states that it waits for the page to fully load before it returns its HTML content, but the timing was limited to ∼30 seconds. Some websites took longer for the table to load and *dryscrape* did not wait long enough. Therefore I could not get sufficient information for websites with no SSL certificate (certificate used by HTTPS websites verifying that the host is indeed the entity it claims to be).

  An alternative solution was to use *selenium*, another Python library for scraping Javascript-heavy webpages. It opens a Chrome or Firefox browser, waits specific time for the page to load then crawls and returns its content. The library has a very useful method called *WebDriverWait()* that waits a given amount of time until an expected condition is met [23]. For example, the method tells the browser to delay the crawl by 30 seconds to locate a specific element by ID, name, xpath, or class. What I essentially do is to wait 40 seconds to find a table of class *"failed"*, which indicates that the given website does not have an SSL certificate or has expired.

- **WebDriverException -** In order to submit pull requests, you have to push from your local repository to the remote one on Github. As mentioned in the previous sections, there is a git hook attached that runs all tests before pushing. Since my chrome driver crashed for some reason, I had to delete the hook in order to successfully push.

- **Semi-structured Data -** The HTML of a page is not made to be read by a machine. Unless it is XML structured, a page can have numerous elements without closing tags. This breaks the page structure, resulting in a "chaos", preventing some elements to be found. I faced this issue when scraping one of the websites - fortunately, this did not interfere with the data I wanted to retrieve but is a risk that needs careful consideration.

- **Hardcoding -** The initial scraping for obtaining a list of domains is hard coded due to the different layout and architecture of every website crawled. For example, the list of domains in `http://www.ranker.com/list/best-gaming-blogs-and-websites/blog-loblaw` and in `http://www.appfreak.net/list-of-100-mobile-game-review-sites/` are situated in different parts of the page's HTML body. Some would be in a *<div>*, others in a *<table>*, and so on. In order to search for domains, I had to change the items the crawler is looking for every different site scraped. It was not very time-consuming, but the searching could have been made more efficient with regular expressions. An example would be finding items containing a string in the form of "/^(http(s?):\/\/)?(www\.)+[a-zA-Z0-9\.\-_]+(\.[a-zA-Z]{2,3})+(\/[a-zA-Z0-9_\-\//s\.\/\?\%\#\&\=]*)?$/".

The time required for the crawler to do all the necessary checking depended on the number of websites to go through, but did the job.

Chapter 5

Results

While writing a web crawler is not complicated, making it efficient was harder than I supposed. Scraping libraries are numerous, with different speed and efficiency on certain aspects. Depending on what you are aiming to achieve by crawling, lightweight libraries might do the job, resulting in a better and faster overall experience. In my case, some pages were script-heavy and required more time to get the necessary data. *Dryscrape* is relatively fast but is limited when it comes to timing. *Selenium*, on the other hand, is a bit slower, as it involves opening the browser and displaying the web page, but can always get all elements of an HTML page. In addition, waiting time for successive requests to a host added up to the total time, but were unavoidable - multiple requests at a time without a server could easily cause overflow. Using invisible displays and terminating them at the end sped up the process slightly.

Some pages were static and could be scraped faster with libraries like *Beautifulsoup*, which provides the possibility of using different parsers and is very fast. For efficiency purposes, I used different parsing libraries for different webpages. For Javascript-heavy pages, *Selenium* and *Dryscrape* had to step in. Occasional connection drop-out was another limitation. Furthermore, the *chromium.sh* script was not well written and had an issue with an ID of an unknown extension, which kept preventing me from pushing to my remote repository and make pull requests. Fortunately, I could do the pushing by taking the testing hook off.

I got up to the point, where I could automatically create rulesets with the web crawler and test them. Due to time restrictions, I could not fix the extension loading issue for chromium, although I tried to find a solution. Nevertheless, the rulesets were tested on chromium as soon as they were pushed to the remote repository, so none of them had failed checks at the end.

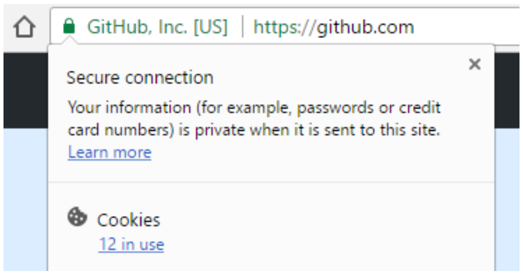
5.1 New Rulesets

As mentioned, the validity checks for pull requests were successful and the only thing left was for the editors to approve and merge the requests. Once done, rulesets were officially added as separate XML files and linked to the “HTTPS Everywhere” atlas. The project had hundreds of open pull requests by then, but merging them was not a problem as long as they have passed all the checks and had the subdomains handled, which was the case with mine. Some rulesets needed a couple of fixes such as adding a mixedcontent platform, adding/removing test URLs, and modifying nonfunctional hosts.

The first time the moderators replied to me, I was asked to use a separate pull request for every new ruleset, as I had a single one for all of them. The point was to create a new branch for each new ruleset so that it could be run through the Travis CI build. I had 26 rulesets written by myself and by the time of submission, half of them were already merged. My repository is available from [16].

5.2 Testing

The project contained some automatic testing scripts that could be run in a standalone Firefox and Chromium profile. Test options included testing rulesets individually, or all at once. Simply navigating to a specific webpage and looking at the address bar to see the connection security was sufficient. There were some edge cases, however. HTTPS pages that appear with a green lock at the front indicate that the client-server connection is secure. Having a warning sign shows that the connection is not 100% secure. Reasons might include links to external files/images, cookies, and specific page of the website, where Javascript is loaded over HTTP. This means that not all parts of the webpage have been handled by its rule. In that case, it is recommended is to be careful about what information is shared as it may compromise the user’s security. Even if that was the case, all websites that had rulesets written were redirecting the client to their HTTPS version. The two figures below demonstrate such an example.

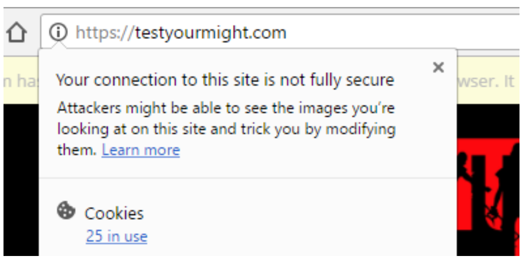

GitHub, Inc. [US] | https://github.com

Secure connection

Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more

Cookies
12 in use

(a) Secure connection


https://testyourmight.com

Your connection to this site is not fully secure

Attackers might be able to see the images you’re looking at on this site and trick you by modifying them. Learn more

Cookies
25 in use

(b) Semi-secure connection

Figure 5.1: Types of connection security

Non-working hosts on particular subdomains could not be targeted for reasons explained in section 4.4.2, so testing them was not possible. Still, they were included in the rulesets' comment sections and will hopefully be handled in the long run. Some contributors were able to speak with the websites' administrators and notify them about the problematic domains. Luckily, most domains were effectively handled when an issue was raised.

5.3 Crawler Efficiency

My web crawler took roughly 44 minutes to scrape through 100 websites and perform all the necessary operations. This is a substantial amount of time, but was normal, considering the Javascript-heavy webpages every domain had to pass through. `https://www.sslshopper.com/ssl-checker.html` took around 30-40 seconds to load all of its scripts. Below is the real, user, and system time taken by the crawler to run on my virtual machine.

- real: 44m 30.330s
- user: 4m 20.852s
- sys: 0m 59.040s

There are workarounds that could be used for an overall improvement, discussed in the next two subsections.

5.3.1 Alternative method for checking HTTPS

There is a project on Github called 'httpswatch', that tracks HTTPS support of prominent websites. It has a python script that gets its data from ".json' files containing a list of sites, split into categories (global, programming, regional- and language-specific). It runs on Python 3.4 and the time taken to run is:

- real: 1m 34.206s
- user 0m 35.260s
- sys: 0m 4.812s

This is obviously much faster than my crawler. Moreover, it does more thorough checks, including sneaky meta redirects. The list of websites is fixed and most of them already have rulesets and in order to add new websites, new ".json' files have to be created and scraped, but this is not a hard thing to do. The problem was that I discovered the project five days before the dissertation deadline. By then I was already done with my rulesets and tests and had to evaluate results, so ran out of time. More information about the project can be found at [11].

5.3.2 Different libraries for scraping dynamic webpages

Beautifulsoup, Dryscrape, Lxml, and Selenium are one of the primary libraries for web scraping. They can be mixed together to obtain better and more efficient results. Another one worth mentioning is *Scrapy*. Scrapy is a collaborative web scraping framework that can crawl through entire websites in a systematic way. It has the options of managing requests, preserving user sessions, and handling output pipelines. Its architecture is complicated, but offers a vast amount of scraping options. The framework is portable and works on Linux, Mac, and Windows. Below is an illustration overview.

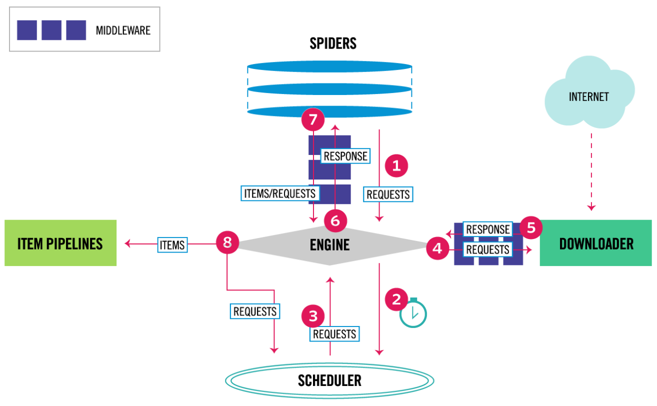


Figure 5.2: Scrapy Architecture Overview

It has multiple components, each responsible for different stages of the crawling process. The execution engine (middle) controls the data flow between components and triggers events when certain actions occur. It is an equivalent of a computer's CPU. A step-by-step breakdown of how the framework operates can be found in the official Scrapy documentation at [20]. There is also a very good tutorial on extracting data with Scrapy by Kais Hassan at [22].

The most powerful tool offered by Scrapy is, by all means, *Scrapy Cloud*. This is a battle-tested platform for running your crawlers. Spiders run in the cloud and scale on demand for millions of webpages. They can even be built visually, scheduled to run automatically, and the platform allows progress monitoring. Servers and constant backups are provided. The platform also offers extensions to speed up crawling process. For example, Crawlera, a smart proxy rotator, helps to bypass bot counter-measures, thus allowing faster crawling of large websites. For more information about Scrapy Cloud and its services, please refer to [21].

The platform is used by clients such as BSpend, SciencesPo, and Swoop. It is very stable and has a generally positive feedback. Unfortunately, I found out about the framework too late and could not experiment with its features and possibly create a better, more efficient crawler. Had I researched and used it earlier, the end result would have been different - more rulesets written (probably hundreds instead of just 26, as I would have scheduled the spider to run automatically on a server) and more thorough testing. If I have time in the future, I will start working with the platform and potentially create a solid web crawler.

Chapter 6

Conclusions

6.1 Limitations and Future Work

This section focuses on any piece of unfinished work from my original plans. The time for completing the project was limited to seven months, so some objectives had to be taken away. In terms of references and literature review, I managed to research everything planned from the beginning, with the exception of one topic - the nature and ethics of cryptocurrencies. I ran out of time, so I could not reflect on it. It would have been better if I focused more on modern encryption techniques (e.g. RSA and SHA-256) rather than emphasizing on history. In addition, I wanted to discuss the background reading for web crawlers, but once again, time did not allow - links to the papers read can be found in section 2.4.

Due to my changes of development ideas throughout the course of the project, I had to cut down on several plans and features. I started doing actual work for “HTTPS Everywhere” in mid-February. Furthermore, my ideas for creating rules changed due course as well. Initially, I wanted to create a list of sites and create rules manually before I came with the decision to create a web crawler. Generating a list of domains was another issue. My first attempt was from `http://www.registered-domains-list.com/` before I realised most of the domains were outdated or for sale. At the end, I started crawling through webpages to get gaming-oriented websites, which worked much better. Checking for HTTPS support was the most time-consuming operation of all. The SSL shopper had to check the presence of SSL certificates including their validity, issuers, and expiration date. If the certificates were trusted by all major web browsers, the checks did not take more than 10 seconds, but in a case of failure, the process of obtaining results was much slower. The time required was linear and increased with the number of websites to check.

Overall, creating rulesets was fun and interesting. It did take me some time to figure out and understand the best practices, but the final outcome was successful. I spoke to some of the project moderators, who gave me useful pieces of advice. All the rulesets I wrote followed the preferred styling guidelines:

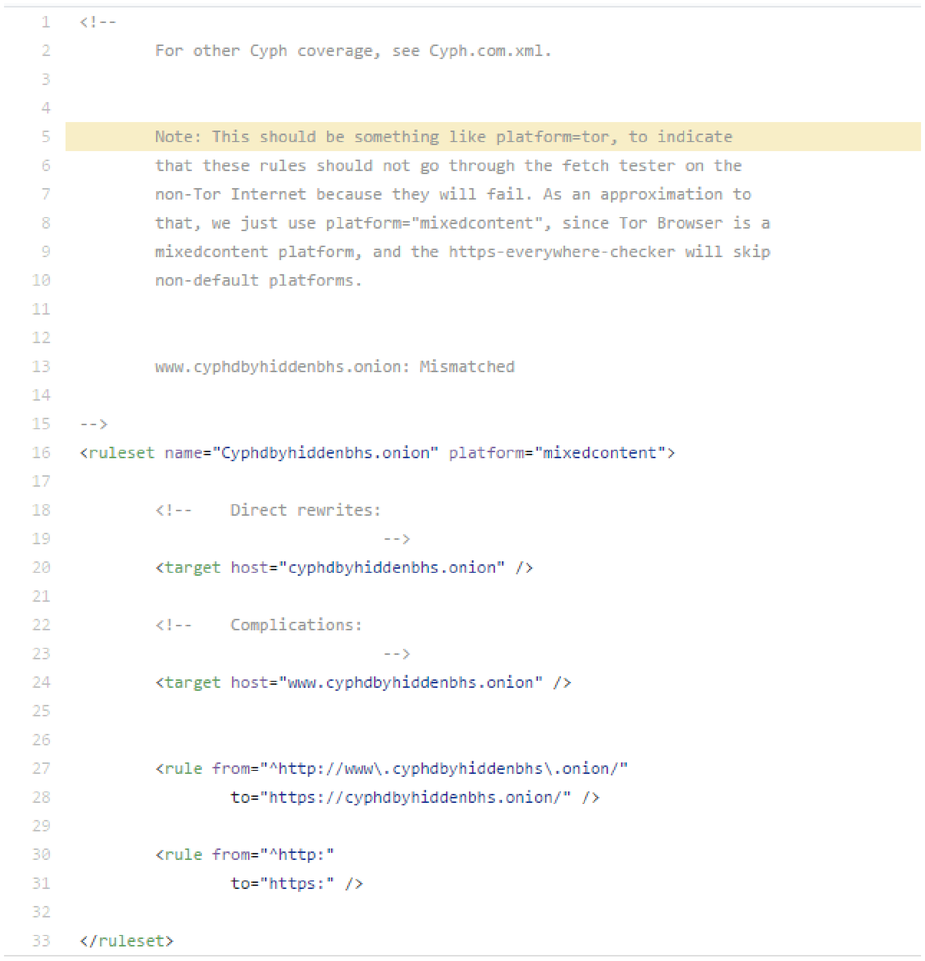
- Left wildcard targets and complex regular expressions were avoided.
- Most of the subdomains were targeted and the non-functional ones were noted down.
- Cookies were secured in case the website's server did not do it.
- Test URLs were included when necessary to ensure a correct HTTPS redirection.

6.1.1 New-purpose Crawlers

My web crawler helped substantially. Creating rulesets automatically allowed more commits and more time for modifications. However, one of the big issues of "HTTPS Everywhere" is the lack of automatic ruleset updates. It is not on a rare occasion for some domains to become outdated, whether due to a new domain is used, or the current one has expired or is for sale. A crawler for checking domain validity would be very useful. Due to the lack of such, there were many rulesets disabled by default (nearly 4000) and others that did not do any job. The idea is to have a server that would run the crawler constantly and continuously check the domains. If any errors are spotted, it should be able to create or modify existing XML files with different ownership - modification includes disabling, deleting, adding more targets, etc. Had I had more time, I would probably have initiated some work on it. In the future, I can give this as a suggestion to the project moderators.

6.1.2 HTTPS Everywhere on the Deep Web

Rulesets can be used and tested on Tor browser. This is practical because Tor works with clear web link as well. Although Tor connection is more secure in general, there can still be a man-in-the-middle to intercept client-server communication. In order for rulesets to work on the deep web, they will have to bypass the fetch tester, otherwise they will fail. There is currently no way of achieving this, but a possibility would be adding a new value for the platform property such as <*platform="tor"*>. Furthermore, there is a plugin called "Darkweb Everywhere" that can redirect the user to a website's hidden service equivalent if present. For example, if a user makes a request for `https://duckduckgo.com` in Tor, the extension would redirect the client to `http://3g2upl4pq6kufc4m.onion`. This is needed because when a user connects to a site that is not hidden services, it must connect through an exit node. This exit node is able to see which sites are being connected to. By having the hidden service loaded instead of the clearnet URL, you make your entire connection without leaving the Tor network. Figure 6.1 shows an example of a deep web ruleset.



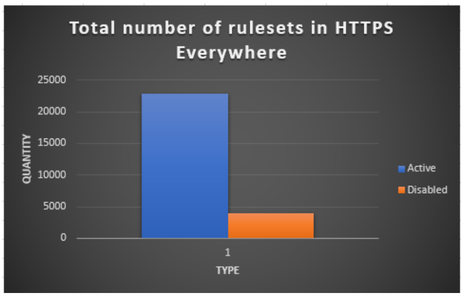
```
<!--
        For other Cyph coverage, see Cyph.com.xml.


        Note: This should be something like platform=tor, to indicate
        that these rules should not go through the fetch tester on the
        non-Tor Internet because they will fail. As an approximation to
        that, we just use platform="mixedcontent", since Tor Browser is a
        mixedcontent platform, and the https-everywhere-checker will skip
        non-default platforms.


        www.cyphdbyhiddenbhs.onion: Mismatched

-->
<ruleset name="Cyphdbyhiddenbhs.onion" platform="mixedcontent">

        <!--    Direct rewrites:
                                -->
        <target host="cyphdbyhiddenbhs.onion" />

        <!--    Complications:
                                -->
        <target host="www.cyphdbyhiddenbhs.onion" />


        <rule from="^http://www\.cyphdbyhiddenbhs\.onion/"
                to="https://cyphdbyhiddenbhs.onion/" />

        <rule from="^http:"
                to="https:" />

</ruleset>
```

Figure 6.1: A deep web ruleset for `cyph.com`

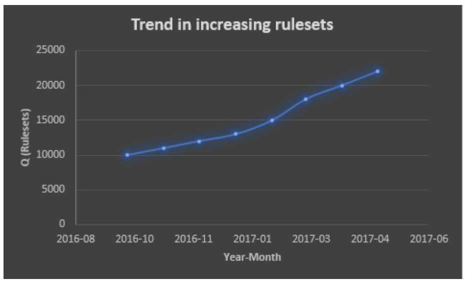
The project looks really interesting and has the potential of getting a new update. The rules can eventually be shipped together with the Tor browser bundle. I will probably try and work on this after finishing with my academic studies. As mentioned, HTTPS Everywhere at its current state cannot redirect from public hosts to hidden services as this would involve updating all the tests. More information about the project can be found at [4].

6.1.3 Statistical Gathering and Analysis

It would have been interesting to gather some statistical data for the rulesets. Rules can be split into categories, then calculate the number of active and disabled. In addition, the mean number of targets, nonfunctional hosts, exclusion patterns, and secured cookies can be derived, which would bring up some curious results. The work could be extended with standard deviations, t-tests, diagrams for displaying the findings, and analysing the discoveries. For example, based on the obtained results, some bar charts can be created that would look like the ones below.



(a) Total number of rulesets by May 2017



(b) Ruleset trends in a year-long period

Figure 6.2: General Diagrams

Data analysis can be expanded even further with tables and more chart types.

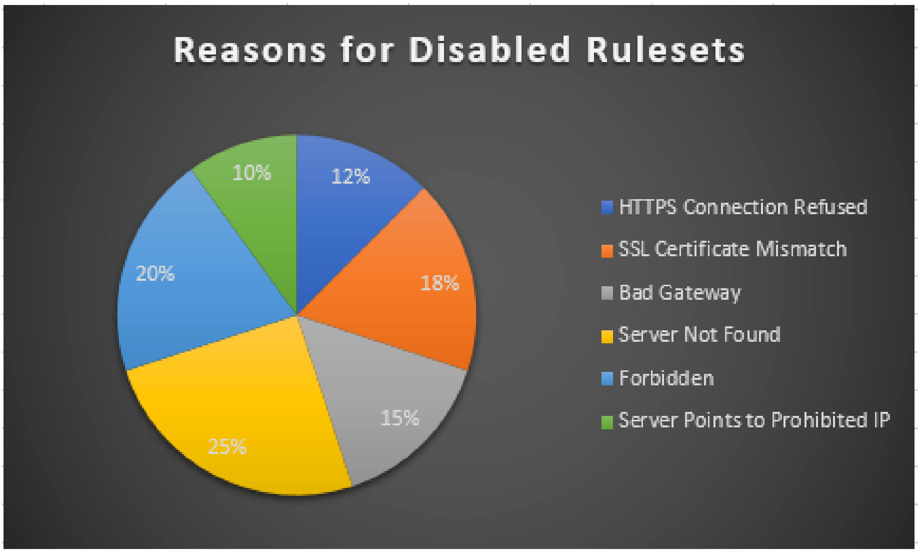


Figure 6.3: Breakdown of disabled rulesets

Empty	Rulesets	No NF* Hosts	With NF Hosts	μ (NF Hosts per Website)
Active	22845	420	22425	56
Disabled	3979	2153	3826	37
Total	**26824**	**573**	**26251**	**93**

Table 6.1: Table with Sample Results

*NF - Nonfunctional

Note that the numbers and calculations in the figures are not based on real data, they are used just for demonstrating purposes. However, I will probably keep working on the project during my free time in the upcoming months, so carrying out a statistical analysis is not impossible.

6.2 Reflection

Doing a final-year project on a topic of my choice was a good opportunity to expand my knowledge and skills in cryptography, web crawling, and cyber security. Not only I learned a great deal of encryption history and how the deep/dark web operates, but also acquired useful coding practices. Thanks to the "HTTPS Everywhere" project, Github and version control are much easier and familiar for me to use. Furthermore, my knowledge in HTTPS, SSL and TLS improved significantly.

Web crawling was the big thing due course. This was something I had no prior experience with and had to begin from scratch. Getting to understand the web scraping concepts, issues, and practices were very interesting and useful - crawlers are used on a daily basis and are the roots of search engines like Google and Yahoo. In addition, writing the crawler was a suitable moment to refresh and expand my Python skills. Due to the use of a virtual machine to write most of my code, I am now comfortable using Linux. Of course, more research and work could have been done had I had more effective time management and task prioritisation. More time was spent on theory and research than on development, which is something I would have changed if I had the opportunity - both theory and practical work are equally important, so keeping a balance between the two is critical. This would have resulted in a more efficient crawler and writing more rulesets respectively. There are multiple places for improvement, but submitting the dissertation does not mean that work is done - this is something I will try to keep working on in the long run, or at least do more research.

I am grateful to my supervisor, Dr. John Power, who was very helpful and friendly throughout the whole course of the project. Thanks to him, I improved my academic language skills and learned best practices for writing a bachelor thesis. The "HTTPS Everywhere" contributors seemed to be really accurate and everyone was helping each other. There were some very interesting open issues with the potential of being developed and expand the extension's functionalities.

To summarize, the dissertation was a big challenge - there were moments of extreme difficulty and the workload was huge. However, thanks to the thesis, the abilities and experience obtained in- and outside the scope of computer science were invaluable.

Bibliography

[1] Aes encryption. https://aesencryption.net/. Accessed: 01/11/2016.

[2] Bitcoin price index chart. http://www.coindesk.com/price/. Accessed: 22/11/2016.

[3] The caesar cipher. https://learncryptography.com/classical-encryption/caesar-cipher. Accessed: 30/10/2016.

[4] Darkweb everywhere. https://github.com/chris-barry/darkweb-everywhere. Accessed: 30/04/2017.

[5] Enigma cipher machines. http://www.cryptomuseum.com/crypto/enigma/. Accessed: 29/10/2016.

[6] Ewma control charts. http://www.itl.nist.gov/div898/handbook/pmc/section3/pmc324.htm. Accessed: 07/12/2016.

[7] Geckodriver proxy. https://github.com/mozilla/geckodriver. Accessed: 13/04/2017.

[8] Https everywhere. https://www.eff.org/https-everywhere. Accessed: 02/03/2017.

[9] Https everywhere rulesets. https://www.eff.org/https-everywhere/rulesets. Accessed: 02/03/2017.

[10] Https everywhere rulesets. https://www.eff.org/https-everywhere/rulesets. Accessed: 27/02/2017.

[11] Https watch. https://github.com/benjaminp/httpswatch. Accessed: 01/05/2017.

[12] Introduction to bitcoins. https://bitcoin.org/en/. Accessed: 15/11/2016.

[13] The one time pad. https://learncryptography.com/classical-encryption/one-time-pad. Accessed: 30/10/2016.

[14] The onion router. https://learncryptography.com/modern-encryption/tor-the-onion-router. Accessed: 06/11/2016.

[15] An overview of the ssl or tls handshake. `http://www.ibm.com/support/knowledgecenter/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm`. Accessed: 05/11/2016.

[16] Philip's https everywhere repository. `https://github.com/philip94/https-everywhere`. Accessed: 03/05/2017.

[17] Prohibiting secure sockets layer (ssl) version 2.0. `https://tools.ietf.org/html/rfc6176`. Accessed: 05/11/2016.

[18] Project dryscrape. `https://github.com/niklasb/dryscrape`. Accessed: 15/04/2017.

[19] Project sublist3r. `https://github.com/aboul3la/Sublist3r`. Accessed: 20/04/2017.

[20] Scrapy architecture overview. `https://doc.scrapy.org/en/latest/topics/architecture.html`. Accessed: 01/05/2017.

[21] Scrapy cloud. `https://scrapinghub.com/scrapy-cloud/?_ga=2.114887236.25413642.1493813992-272549978.1493807751`. Accessed: 01/05/2017.

[22] Scrapy tutorial for data extraction. `https://medium.com/@kaismh/extracting-data-from-websites-using-scrapy-e1e1e357651a`. Accessed: 01/05/2017.

[23] Selenium waits. `https://selenium-python.readthedocs.io/waits.html`. Accessed: 15/04/2017.

[24] Selenium with python. `https://selenium-python.readthedocs.io/`. Accessed: 15/04/2017.

[25] Single exponential smoothing. `http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm`. Accessed: 07/12/2016.

[26] Tor project: Overview. `https://www.torproject.org/about/overview.html.en`. Accessed: 22/10/2016.

[27] Wireshark. `https://www.wireshark.org/`. Accessed: 07/02/2017.

[28] Public key encryption and digital signature: How do they work? *CGI Busines Solutions Through Information Technology* (2004), 1–12.

[29] Akhoondi, M., Yu, C., and Madhyastha, H. V. Lastor: A low-latency as-aware tor client. *IEEE/ACM Transactions on Networking 22* (2014), 1742–1755.

[30] Backes, M., Koch, S., Meiser, S., Mohammadi, E., and Rossow, C. Poster: In the net of the spider: Measuring the anonymity-impact of network-level adversaries against tor. *CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), 1626–1628.

[31] Bin He, Mitesh Patel, Z. Z., and Chang, K. C.-C. Accessing the deep web. *Communications of the ACM 50*, 5 (may 2007), 96–97.

[32] Brooke, Z. A marketer's guide to the dark web. *Marketing Insights 28*, 1 (2016), 24–28.

[33] Chen, H. *Exploring and Data Mining the Dark Side of the Web*, vol. 30. Springer, 2012, ch. 22: Botnets and Cyber Criminals. 429-430, Section 1.1 'The Underground Economy'.

[34] Dade, L. How enigma machines work. `http://enigma.louisedade.co.uk/howitworks.html`. Accessed: 26/10/2016.

[35] Dziembowski, S. Introduction to cryptocurrencies. *CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), 1700–1701.

[36] Eddy, M. Inside the dark web. *PC Magazine* (2015).

[37] Edwards, J., McCurley, K., and Tomlin, J. An adaptive model for optimizing performance of an incremental web crawler. *WWW '01 Proceedings of the 10th international conference on World Wide Web* (2001), 106–113.

[38] Gehl, R. W. Power/freedom on the dark web: A digital ethnography of the dark web social network. *New Media & Society 18*, 7 (2016).

[39] Giles, C. L., Sun, Y., and Councill, I. G. Measuring the web crawler ethics. *WWW '10: Proceedings of the 19th international conference on World wide web* (2010), 1101–1102.

[40] Gopal, D., and Heninger, N. Torchestra: Reducing interactive traffic delays over tor. *WPES '12 Proceedings of the 2012 ACM workshop on Privacy in the electronic society* (2012), 31–42.

[41] Goulding, T. A first semester freshman project: The enigma encryption system in c. *ACM Inroads '13 4* (2013), 43–46.

[42] Hobson, D. What is bitcoin? *Crossroads, The ACM Magazine for Students - The Complexities of Privacy and Anonymity 20*, 1 (2013), 40–44.

[43] Houtven, L. V. *Crypto 101.* Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0) license, 2013-2016, ch. 15: SSL and TLS. 182-184, Section 15.1 and 15.2'.

[44] Johnson, A., Wacek, C., Jansen, R., Sherr, M., and Severson, P. Users get routed: Traffic correlation on tor by realistic adversaries. *CCS '13 Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), 337–348.

[45] lvarez, M., Raposo, J., Pan, A., Cacheda, F., Bellas, F., and Carneiro, V. Deepbot: A focused crawler for accessing hidden web content. *DEECS '07 Proceedings of the 3rd international workshop on Data enginering issues in E-commerce and services: In conjunction with ACM Conference on Electronic Commerce (EC '07)* (2007), 18–25.

[46] Miessler, D. The internet, the deep web, and the dark web. `https://danielmiessler.com/study/internet-deep-dark-web/#gs.SKDYXBg`. Accessed: 22/10/2016.

[47] Mirtaheri, S. M., Dincturk, M. E., Hooshmand, S., Bochmann, G. V., Jourdan, G.-V., and Onut, I. V. A brief history of web crawlers. *CASCON '13 Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research* (2013), 40–54.

[48] Naylor, D., Finamorey, A., Leontiadisz, I., Grunenbergerz, Y., Melliay, M., Munafy, M., Papagiannakiz, K., and Steenkiste, P. The cost of the 's' in https. *CoNEXT '14: Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (nov 2014), 133–140.

[49] Press, C. U. Web crwaling indexes. - (2009), 443–459.

[50] Shirazi, F., Diaz, C., and Wright, J. Towards measuring resilience in anonymous communication networks. *WPES '15 Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society* (2015), 95–99.

[51] Smith, N. Classic project: The enigma machine. *Engineering and Technology 9* (2014), 42–43.

[52] Tang, C., and Goldberg, I. An improved algorithm for tor circuit scheduling. *CCS '10 Proceedings of the 17th ACM conference on Computer and communications security* (2010), 329–339.

[53] Thomas, M., and Mohaisen, A. Measuring the leakage of onion at the root. *WPES '14 Proceedings of the 13th Workshop on Privacy in the Electronic Society* (2014), 173–180.

[54] von dem Bussche, F., Weiand, K., Linse, B., Furche, T., and Bry, F. Not so creepy crawler: Easy crawler generation with standard xml queries. *WWW '10 Proceedings of the 19th international conference on World wide web* (2010), 1305–1308.

[55] Wagstaff, K. The internet and the world wide web are not the same thing. `http://www.nbcnews.com/tech/internet/internet-world-wide-web-are-not-same-thing-n51011`, mar 2014. Accessed: 23/10/2016.

Appendix A

Useful Side Information

This chapter contains additional information for the fields investigated in the thesis that do not go along with the main line of thought.

A.1 How Enigma Works

Part of the information in this paragraph is taken from the article *'A First Semester Freshman Project: The ENIGMA Encryption System in C'*, which briefly explains how the machine works [41]. The Enigma Machine consists of a keyboard, a set of lights for each letter in the alphabet (a lightboard), three to five rotors (a small wheel that would spin around and calculate a letter to substitute for each character in the message), and a plug board at the bottom. The algorithm works by typing a letter and for each letter, a corresponding one would light up on the light board. For each key press, the rotors would move, perform the appropriate letter substitutes, and eventually send the message to the recipient. The letters are scrambled with the plug board, which switched pairs of letters around, adding a significant amount of complexity, and then encrypting the message with rotors, which move each character in the message as mentioned previously. One can type the same letter continuously, but Enigma would output a bunch of different individual letters (p. 44, How the Enigma Works) [41]. To decode the message, one needs to know what rotor and plug board settings are used to encrypt the message. These settings are accomplished with monthly sheets - different codes would be used daily and they would get a new sheet at the end of each month. The allies' aim was to get the code sheet and understand the Germans' communication, without them knowing. Figure A.1 shows the journey of a single letter in the Enigma.

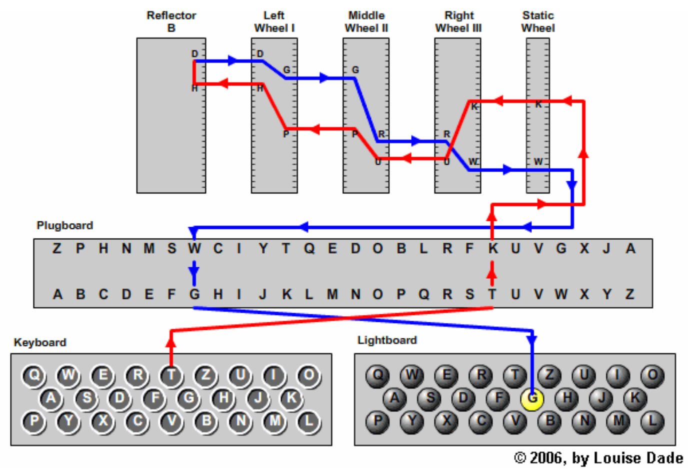


Figure A.1: An Enigma Emulator [34]

There have actually been various models of the cipher machine, developed before and during WWII. Development started in 1915 with the invention of the Rotor Machine, the glow lamp Enigma in 1924, and much more variations until the Enigma M3 and M4, created in 1942 by the German Navy and used until the end of the war [5]. Its design has inspired the development of other machines such as the American KL-7 and the Russian Fialka [5]. More information about the Enigma can be found in Louise Dade's website, including emulator source code, key generators, mechanics, etc. [34]. The "Classic Project" by Nick Smith contains information about Wehrmacht Enigma I and its descendants [51].

A.1.1 Cracking the Enigma Code

The machine had a very complicated encryption scheme. However, it was not perfect and was eventually broken. The plug board was an old telephone style mapping between letter pairs, making the two letters swap before being sent into the rotors for encryption. This added a crazy moment of mathematical redundancy to the Enigma, but at the same time allowed for an important weakness to be discovered. In particular, the machine's design did not allow for a character to be encoded into itself (p. 44, How the Enigma Works) [41]. The flaw eventually led to the deduction of some plug positions. Assuming that no other problems were found (e.g. two plugs pointing to the same letter), one could brute-force the plug positions without having to try all of the possible options. So the Allies had to build a device to automate the process of decrypting messages daily, thus giving birth to 'The Bombe'.

> *Despite increased complexity being added to the system with everchanging daily key settings and auxiliary documents, Enigma was eventually broken, not so much because of any technical flaws in the design of the system, but more due to the capture of codebooks on 9 May 1941 from German U-boat U-110 as well*

as procedural malpractices by Enigma operators (p. 43) [51].

The Bombe was designed by Alan Turing and Gordon Welchman and took the form of emulating several hundred Enigma rotors, as well as functioning as a logical electrical circuit to automate the deductions needed to rule out the flawed possible attempts. Its invention took place at Bletchley Park Mansion, Milton Keynes, the centre of Allied code-breaking during World War II, currently turned into a history museum. Around 150 Bombes had been built by 1944. To give a simple example, imagine you have the Nazi's broadcasting weather report every morning encrypted by the Enigma code. This was done in the same format daily, but with a different code sheet. Later, the Germans switched to five rotor machines and double-encrypting messages, but the British caught up with them very quickly. At some point, the Allies even improved the Enigma cipher concept by fixing the issue of letters not being able to be mapped to themselves. In the end, the Bombe was better than the Enigma itself, so the Nazis could not crack it and essentially led to their loss in the war [51].

A.2 Technical Background

This section contains technical background and results analysis from some papers. The focus is on researching and discussing software systems that measure the security and leakage of *.onion* domains. Data leakage clearly presents an issue to Tor users and needs to be explored. Anonymity browsers are vulnerable to traffic correlation attacks, where an adversary is monitoring the entry and exit node (p. 1626, Introduction) [30]. There are two fundamental methods of exposing security on Tor:

- **Network Adversary Model -** Such models aim at gathering data in nodes between a client and a server - the more data obtained, the less secure the browser is. Data gathering is mostly subject to user settings, which are often ignored by the majority of people using Tor. This is done through traffic analysis using a certain behaviour - an adversary would usually act as a relay in order to monitor data more easily. The important point here is the adversary's resource endowment, i.e. how many resources to allocate so that the traffic correlation attack can be carried out effectively [44]. The most valuable resource is bandwidth. This is because a node route from a client to a server is chosen based on how much bandwidth each relay is capable of storing and transferring to the next one. The higher the bandwidth, the bigger the probability for this specific node to be chosen. Several user models exist in order to maximise efficiency, depending on the type of traffic in Tor. For this project, some models and attack scenarios will be discussed that would help to analyse Tor's resilience against the attacks.

- **Distributed Denial of Service Attack -** Performing a deliberate attack of this kind is a method of revealing weaknesses in client/server communication and based on the results, aim to improve it. For example, some of the attacks aim at increasing Tor's latency, which by default is higher than normal networks, due to the additional security layers. An increased latency means reduced quality of service - such attacks would not only obtain traffic data but would also dissuade users from using the network. Latency can be measured in “ttfb” (time to first byte) and “ttlb” (time to last byte). A function $L_u(r)$ is defined, that returns the latency of route r chosen by user u (p. 97, Tor's Resilience in Terms of Latency) [50].

A.2.1 Torchestra: Reducing Interactive Traffic Delays Over Tor

Apart from data leakage, there are other methods for reducing interacting traffic delays over Tor. Recall that Tor traffic is much slower than the one in the clear web due to the multi-level encryption and the nodes controlled by user entities. This means that the bandwidth for each node is different and dependent on connection speed and hardware properties. A method by Gopal and Heninger [40] seems to have a potential solution to the delay issues.

The Torchestra approach aims at creating two separate connections between each pair of nodes. Ideally, it separates bulk and interactive traffic. The interactive one would be prioritised by having a separate socket and output buffer from the bulk - congestion is minimised, which will improve Tor's light traffic performance. A method called *Exponentially Weighted Moving Average* (EWMA) by Tang and Goldberg (p. 331, Section 2.3) [52] seems suitable for facilitating Tor traffic. From reading different papers with potential solutions, EWMA looks the most promising - the formula is not difficult to understand and results show that if implemented properly, node traffic will be significantly reduced, leading to a better connection. It calculates a moving average of the number of cells sent on a circuit and adds greater weight to recent values. Circuits begin moving to the light traffic and depending on the data threshold, might move to heavy and the other way round. The question here is how to decide whether a circuit should be put for light or heavy traffic. This is done by the exit node. To start with, EWMA for a given period of time t needs to be calculated, given by the formula below.

$$EWMA(t) = \alpha \cdot Y(t) \ + \ (1-\alpha) \cdot EWMA(t-1)$$

Y(t) is the data observed for time t and α is a constant between 0 and 1 that determines the depth of memory of EWMA. It represents the degree of weighting decrease. The rate at which older data enters into the calculation of EWMA is determined by α. For example, if α has a large value (e.g between 0.5 and 1), there will be more weight to recent data and less to the older data - smoothing will be quick. A value closer to 0 on the other hand, will result in a heavier influence by older observations as well as slower smoothing [6, 25].

Deciding whether a circuit belongs to the light or heavy connection is determined by threshold values T_l and T_h. If a circuit's EWMA is larger than T_l times the average EWMA for the light connection, it will be moved to heavy - the reverse procedure is done for moving circuit from light to heavy. When a client request for a website is given, data cells are sent through circuits from source to destination and then reverse (in this case from the exit node, through the middle, followed by the entrance one). The procedure is explained in Gopal and Heninger's paper [40] step by step:

1. *Check whether a heavy connection has been created between exit and middle router. If not, create a new connection.*
2. *The exit node sends a `SWITCH` cell on the light connection to inform the middle node that no more cells for this circuit will be coming in on this connection.The payload in the `SWITCH` cell contains a flag that the exit node sets to inform the middle node that it needs to extend the heavy connection towards the entrance. The exit node continues to receive cells from the middle node on the light connection.*
3. *The exit node sends a `SWITCHED_CONN` cell on the heavy connection followed by the circuit's cells.*
4. *Once the middle router has received both the control cells, it sends a `SWITCHED` cell on the light connection and only then does it start processing the circuit's cells from the heavy connection. The cells that might have arrived on the heavy connection before the `SWITCH` cell arrived on the light connection are saved in a queue and these cells are processed once both the control cells are processed. This completes the circuit switch on the middle node.*
5. *After the exit node receives the `SWITCHED` cell on the light connection, no more cells for this circuit will be coming from the middle node on this connection. It completely switches the circuit to the heavy connection. The cells that might have arrived on the heavy connection before the `SWITCHED` cell arrived on the light connection are saved in a queue and these cells are processed once the `SWITCHED` cell has been processed.*

Experiments in the paper were performed using Tor emulation toolkit that simulates an entire network. Details about the results can be found in the "Experimental Results" section at [40].

A.2.2 Examples with Results

Some examples are given in order to better understand the concept of traffic reduction. One is taken from Gopal and Heningber's paper [40], where web traffic is simulated, and the other one is from Arkhoondi et al [29], in which a low-latency Tor client is developed.

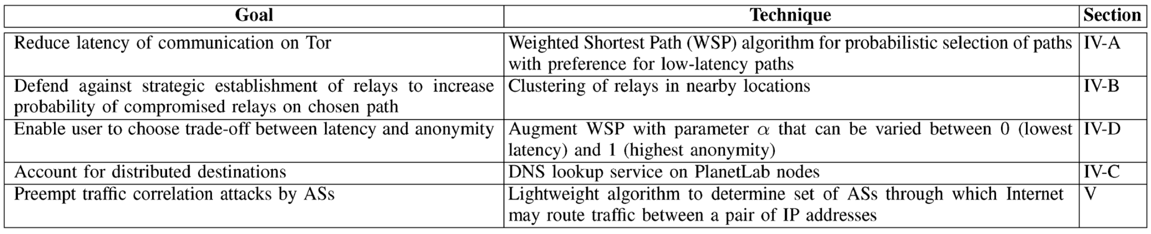
Simulating Web and SSH Traffic

This is a useful method for representing real traffic patterns. Sessions were made with the help of *Wireshark*, a widely-used network protocol analyzer. It allows monitoring network traffic on a really low level and has been a standard for the government and other enterprises [27]. Wireshark traffic was replayed as a light one on the background of some heavy clients downloading files of roughly 100MB. Experiments were performed on the original Tor, prioritized, and Torchestra with a Java process to simulate traffic timings. The process *creates a separate thread for every circuit ID to be simulated, each of which is created with a different IP address and listens on different sockets. The ExperimenTor source clients connect to each of these threads' sockets through Tor. Every thread maintains a table of cell transmission times and sleeps for the appropriate interval after sending a cell.* More information including results can be found in the paper from which the example is taken (p. 36, section 5.3) [40].

LASTor: A Low-Latency AS-Aware Tor Client

Tor is famous for its relatively low-level latency communication, compared to other browsers facilitating anonymity (e.g anonymous Google Chrome). However, latency may increase in the desire to secure anonymity. For example, when a client requests a website, a tunnel is established at both ends with three relays in between. The relays may turn out to be at a very large distance from each other, which would result in high overall latency. Improvements on the client-side can be made to achieve low latency and preserving security by making an efficient path-finding algorithm. This is indeed the Low-Latency Tor client (p. 1742, section I) [29].

To begin with, the client must be resilient to attacks from an autonomous system (AS). Such systems are able to analyse traffic from the client to the entry node and from the exit node to the server. The LASTor client should be aware of Internet routing between relays and end-hosts. The point is to predict a set of ASs through which the Internet may route traffic between a pair of IP addresses. However, there might be some users that would not like the tradeoff of reduced anonymity for reduced latency (p.1743, section I) [29]. Ideally, the LASTor would enable a user to choose a value between 0 (lowest latency) and 1 (highest anonymity) for a single parameter that would indicate the algorithm to choose an appropriate path. Keeping the right balance is what makes the implementation difficult, i.e. choosing geographically shorter paths without significantly compromising security. Latency is measured by visiting several world-famous websites through both LASTor and conventional Tor client. In addition, AS-level routes are used to measure LASTor's ability to avoid them. The results in the paper show that the low-latency Tor client missed only 11% of the possible AS-level routes, while the conventional client ended with a significantly higher rate of 57% (p. 1743, section I) [29]. The overview of techniques used to build LASTor is displayed as a table in Figure 7.



Goal	Technique	Section
Reduce latency of communication on Tor	Weighted Shortest Path (WSP) algorithm for probabilistic selection of paths with preference for low-latency paths	IV-A
Defend against strategic establishment of relays to increase probability of compromised relays on chosen path	Clustering of relays in nearby locations	IV-B
Enable user to choose trade-off between latency and anonymity	Augment WSP with parameter α that can be varied between 0 (lowest latency) and 1 (highest anonymity)	IV-D
Account for distributed destinations	DNS lookup service on PlanetLab nodes	IV-C
Preempt traffic correlation attacks by ASs	Lightweight algorithm to determine set of ASs through which Internet may route traffic between a pair of IP addresses	V

Figure A.2: Goals and Techniques for LASTor [29]

The table is just a basic explanation of the techniques. More details can be found in Akhoondi and Yu's paper [29].

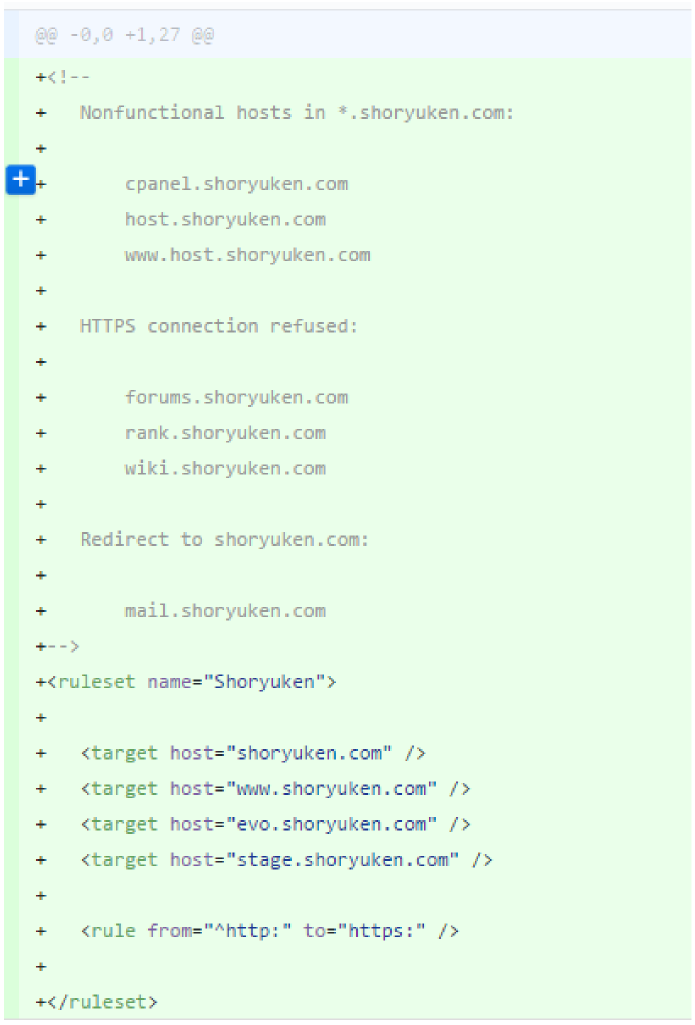
Appendix B

Rulesets

This chapter contains images with examples of rulesets. Some are general, others are more specific.

B.1 General

The first image shows a “conventional” ruleset with multiple targets and some nonfunctional hosts.

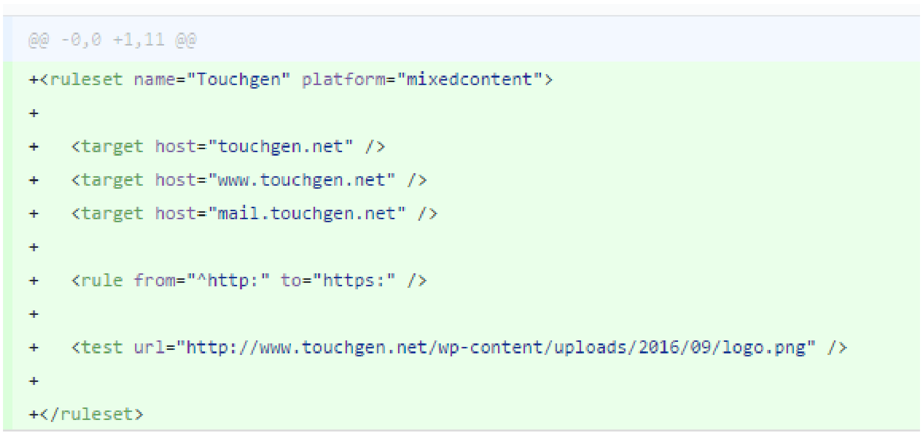


```
@@ -0,0 +1,27 @@
+<!--
+   Nonfunctional hosts in *.shoryuken.com:
+
+       cpanel.shoryuken.com
+       host.shoryuken.com
+       www.host.shoryuken.com
+
+   HTTPS connection refused:
+
+       forums.shoryuken.com
+       rank.shoryuken.com
+       wiki.shoryuken.com
+
+   Redirect to shoryuken.com:
+
+       mail.shoryuken.com
+-->
+<ruleset name="Shoryuken">
+
+   <target host="shoryuken.com" />
+   <target host="www.shoryuken.com" />
+   <target host="evo.shoryuken.com" />
+   <target host="stage.shoryuken.com" />
+
+   <rule from="^http:" to="https:" />
+
+</ruleset>
```

Figure B.1: Ruleset for `shoryuken.com` with non-functional hosts included

B.2 Mixed Content Platform

The next figure is a ruleset with no nonfunctional hosts, but with a mixed content platform. As explained in 4.4.2, MCB occurs when some images and scripts of a webpage execute over HTTP instead of HTTPS. This results in a big difference in the HTML content of the two website versions. There is a test URL to ensure that the image with the main logo of the website loads over HTTPS.

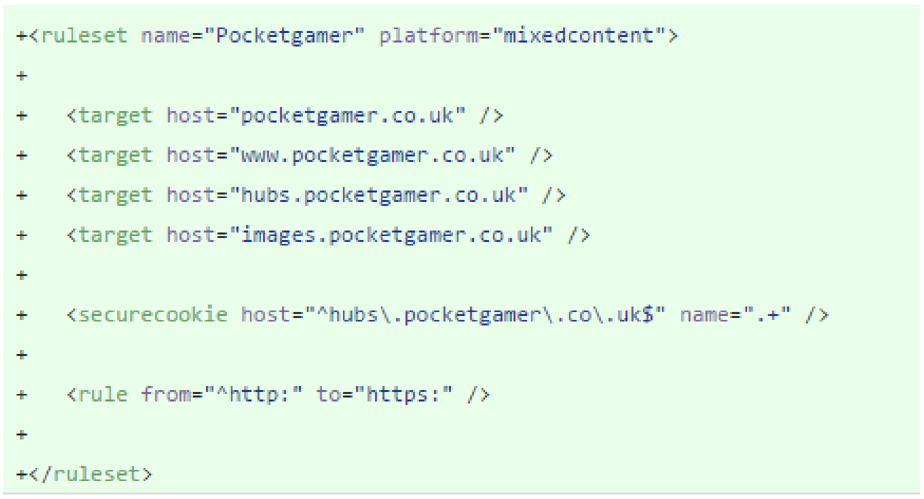


```
@@ -0,0 +1,11 @@
+<ruleset name="Touchgen" platform="mixedcontent">
+
+   <target host="touchgen.net" />
+   <target host="www.touchgen.net" />
+   <target host="mail.touchgen.net" />
+
+   <rule from="^http:" to="https:" />
+
+   <test url="http://www.touchgen.net/wp-content/uploads/2016/09/logo.png" />
+
+</ruleset>
```

Figure B.2: Ruleset for `touchgen.net` with a mixed content platform and test URL

B.3 Secure Cookie

The third image is an example of a ruleset that explicitly needs to secure the website's cookies. Some HTTPS websites fail to set the secure flag on authentication and tracking cookies. The <*securecookie*> tag makes sure to set the flag correctly.



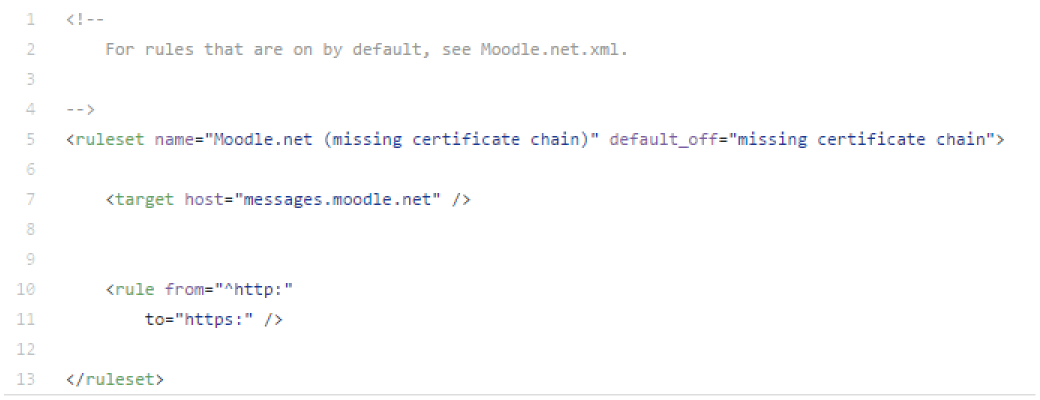
```
+<ruleset name="Pocketgamer" platform="mixedcontent">
+
+   <target host="pocketgamer.co.uk" />
+   <target host="www.pocketgamer.co.uk" />
+   <target host="hubs.pocketgamer.co.uk" />
+   <target host="images.pocketgamer.co.uk" />
+
+   <securecookie host="^hubs\.pocketgamer\.co\.uk$" name=".+" />
+
+   <rule from="^http:" to="https:" />
+
+</ruleset>
```

Figure B.3: Ruleset for `pocketgamer.co.uk` with a *securecookie* tag

The tag accepts two parameters. *Host* specifies the domain that should have its cookies on and the *name* is the ID of the cookie to be secured. To be secured, the cookie must be sent by a target host for that ruleset. A cookie whose domain attribute starts with a "." will be matched as if it was sent from a host name made by stripping the leading dot [10].

B.4 Rulesets Disabled by Default

Some rules might cause problems when enabled by default on a browser. This might happen if the site breaks, has an expired certificate, or any general issue. It usually remains disabled until the problem is fixed.



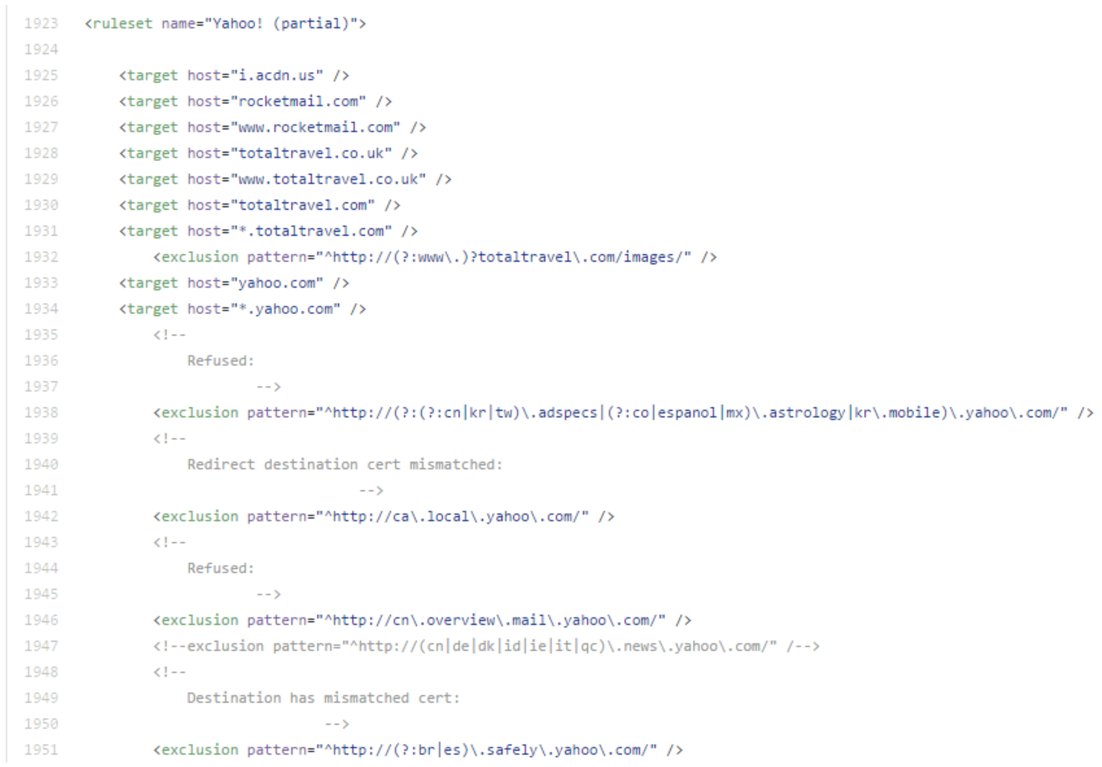
```
<!--
    For rules that are on by default, see Moodle.net.xml.

-->
<ruleset name="Moodle.net (missing certificate chain)" default_off="missing certificate chain">

    <target host="messages.moodle.net" />


    <rule from="^http:"
        to="https:" />

</ruleset>
```

Figure B.4: Ruleset for `moodle.net` with a *default_off* attribute

The <*default_off*> attribute switches the rule off and has a value of explaining why. In the above case, `http://moodle.net`'s SSL certificate chain is missing (N.B. This ruleset was NOT written by me, it is illustrated for example purposes).

B.5 Exclusion Pattern

The exclusion pattern is a regular expression that specifies a domain on which the rule should not be applied. Reasons can be refused connection or login path so that logins do not break.



```
<ruleset name="Yahoo! (partial)">

    <target host="i.acdn.us" />
    <target host="rocketmail.com" />
    <target host="www.rocketmail.com" />
    <target host="totaltravel.co.uk" />
    <target host="www.totaltravel.co.uk" />
    <target host="totaltravel.com" />
    <target host="*.totaltravel.com" />
        <exclusion pattern="^http://(?:www\.)?totaltravel\.com/images/" />
    <target host="yahoo.com" />
    <target host="*.yahoo.com" />
        <!--
            Refused:
                    -->
        <exclusion pattern="^http://(?:(?:cn|kr|tw)\.adspecs|(?:co|espanol|mx)\.astrology|kr\.mobile)\.yahoo\.com/" />
        <!--
            Redirect destination cert mismatched:
                                -->
        <exclusion pattern="^http://ca\.local\.yahoo\.com/" />
        <!--
            Refused:
                    -->
        <exclusion pattern="^http://cn\.overview\.mail\.yahoo\.com/" />
        <!--exclusion pattern="^http://(cn|de|dk|id|ie|it|qc)\.news\.yahoo\.com/" /-->
        <!--
            Destination has mismatched cert:
                                -->
        <exclusion pattern="^http://(?:br|es)\.safely\.yahoo\.com/" />
```

Figure B.5: Partial ruleset for `yahoo.com` with an *exclusion pattern* tag

`Yahoo.com` seems to have a couple of exclusion patterns. The first and last one are due to refused connection, while the second one has a mismatched SSL certificate and redirects the client to a different domain. The ruleset has multiple authors, with credits given to all of them.

B.6 List of Personal Rulesets

- `http://148apps.com`
- `http://appadvice.com`
- `http://appolicious.com`
- `http://bluesnews.com`
- `http://controllerpro.com`
- `http://cultofmac.com`
- `http://eventhubs.com`

- `http://freshapps.com`
- `http://gamecritics.com`
- `http://gamegeek-denter.de`
- `http://gamerswithjobs.com`
- `http://gamesandpals.com`
- `http://gameselectors.com`
- `http://gamewithyourbrain.com`
- `http://howinerd.com`
- `http://iphoneblog.com`
- `http://macobserver.com`
- `http://pcgamer.com`
- `http://pocketgamer.biz`
- `http://pocketgamer.co.uk`
- `http://shoryuken.com`
- `http://siliconera.com`
- `http://testyourmight.com`
- `http://theiphonemom.com`
- `http://touchgen.net`
- `http://videogamer.com`

Appendix C

Code

The code for my crawler consists of a single Python file and can be viewed below. Prerequisites required to run it are explained in 4.2.4.

C.1 File: pn-webcrawler.py

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import
    expected_conditions as EC
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.common.by import By
from selenium.common.exceptions import
    NoSuchElementException
from pyvirtualdisplay import Display
from webkit_server import InvalidResponseError
from bs4 import BeautifulSoup
import dryscrape
import os
import requests
import urllib2
import subprocess

siteList = []
shortListedSites = []

#Open webpage for domain name scraping
def createSession(crawlingWebsite):
    print "\nCrawler␣Initiated.␣Searching␣in␣'" +
        crawlingWebsite + "'␣for␣domains.\n\n"
    dryscrape.start_xvfb()
    #Begin new session with loaded scripts
    try:
        session = dryscrape.Session()
        session.visit(crawlingWebsite)
        response = session.body()
        #Reset session for handling memory issues
        session.reset()
    except InvalidResponseError as e:
        print "Cannot␣open␣" + crawlingWebsite + "\n"
        print 'InvalidResponseError:', e
        quit()
    soup = BeautifulSoup(response, "html.parser")
    #Searches for hyperrefs in a page. This is the
        hardcoded bit.
    #Searching for items is webpage-specific. For a
        different website, please refer to its HTML
        content
    #to find out which tags are needed to obtain a list
        of domains if any.
    tableFound = soup.findAll("a", {"target": "_blank"})

    if len(tableFound) == 0:
        print "Nothing␣found.␣Terminating␣crawler."
        quit()
    else:
        for row in tableFound:
            #Add found domains to the list of sites
            siteList.append(row.get('href'))

#Check every domain for HTTPS Support
def checkHTTPS():
    #Selenium opens the web browser by default
    #I have made the display invisible to save time and
        memory efficiency.
    display = Display(visible = 0, size = (800, 600))
    display.start()
    httpsSiteList = []

    for website in siteList:
        print website
        #Change path depending on the location of your
            geckodriver
        driver =
            webdriver.Firefox("/home/osboxes/Downloads/")
        driver.get("http://www.sslshopper.com/ssl-checker.html?hostname="
            + website)

        delay = 40
        try:
            #Wait for 40 seconds before returning any
                results.
            #This makes sure that whole page will be
                loaded before scraping.
            WebDriverWait(driver,
                delay).until(EC.presence_of_element_located((By.CLASS_NAME,
                'failed')))
            print website + "␣does␣NOT␣support␣HTTPS!"
            print "Moving␣on␣to␣the␣next␣website.\n\n"
            driver.quit() #Close browser
            display.stop() #Close display
        except TimeoutException, NoSuchElementException:
                print website + "␣supports␣HTTPS!"
                driver.quit()
                display.stop()
                httpsSiteList.append(website)
                checkAtlas(website)

#Check if site is present in the atlas and create ruleset
def checkAtlas(wSite):
    hyperLinks = []
    try:
        siteChunks = wSite.lower().split(".")
        if "www" in siteChunks[0]:
            del siteChunks[0]
        elif "http://" in siteChunks[0] or "https://" in
            siteChunks[0]:
            siteChunks[0] = siteChunks[0].split('/')[2]
        r =
            requests.get("http://www.eff.org/https-everywhere/atlas/letters/"
            + siteChunks[0][0].lower() + ".html")
        resp = urllib2.urlopen(r.url)
        soup = BeautifulSoup(resp.read(), "html.parser")

        domainList = soup.findAll('ul', id='domain-list')
        #Loop through all domains starting with the
            first letter of the
        #website currently being checked
        for link in domainList:
            for site in link.findAll('li'):
                hyperLinks.append(site.text)
        for atlasSite in hyperLinks:
            if siteChunks[0].lower() ==
                atlasSite.split(".")[0]:
                print siteChunks[0].lower() + "␣is␣in␣
                    the␣atlas␣and␣already␣has␣a␣
                    ruleset.\n\n"
                break
            else:
                if atlasSite == hyperLinks[-1]:
                    print siteChunks[0].lower() + "␣is␣
                        NOT␣in␣the␣atlas.␣Creating␣
                        ruleset..."
                    ruleSite = ".".join(str(x) for x in
                        siteChunks)
                    #Create ruleset for the website
                    ruleOutput = subprocess.call(["bash␣
                        make-trivial-rule␣" +
                        ruleSite.split('/')[0]], shell =
                        True, cwd =
                        "/home/osboxes/Documents/https-everywhere/rules")
                    if (ruleOutput == 0):
                        print "Successfully␣created␣a␣
                            ruleset␣for␣'" + ruleSite +
                            "'!\n\n"
                        shortListedSites.append(ruleSite)
                    else:
                        print "Oops!␣'" + ruleSite + "'␣
                            seems␣to␣already␣have␣a␣
                            ruleset␣in␣your␣repository.␣
                            Please␣examine␣it!\n\n"
                    break
                continue
    #Catch some exceptions
    except IOError, e:
        if hasattr(e, 'code'): # HTTPError
            print "Error␣loading␣website␣" + r.url
            print "HTTP␣error␣code:␣" + str(e.code) + "␣
                -␣" + str(e.reason) + "."
            pass
        elif hasattr(e, 'reason'): # URLError
            print "Cannot␣connect,␣reason:␣" +
                str(e.reason) + "\n"
            pass
        else:
            pass
    except ValueError, ex:
        print "Value␣error\n"
    except:
        print "Unexpected␣error.\n"
        pass

#Hardcoded - need to specify in the function call which
    site you would like to crawl
createSession("http://www.appfreak.net/list-of-100-mobile-game-review-sites/")
print "Sites␣found:␣" + str(len(siteList)) + "\n"
for foundSite in siteList:
    print foundSite
print "\n\n"
checkHTTPS()
```

```
print "The␣crawler␣successfully␣created␣rulesets␣for␣" +
    str(len(shortListedSites)) +"␣websites:\n"
for shortSite in shortListedSites:
        print shortSite
```