20 years of Bleichenbacher attacks

Gage Boyle

Technical Report

RHUL–ISG–2019–1

27 March 2019

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

Student Number: 100866673
Gage, Boyle

20 Years of Bleichenbacher Attacks

Supervisor: Kenny Paterson

Submitted as part of the requirements for the award of the
MSc in Information Security
at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, and in accordance with these regulations I submit this project report as my own work.

Signature:

Date:

Acknowledgements

I would first like to thank my project supervisor, Kenny Paterson. This project would not have been possible without his continuous encouragement to push the boundaries of my knowledge, and I am grateful for the commitment and expertise that he has provided throughout. Secondly, I would like to thank Nimrod Aviram for his invaluable advice, particularly with respect to algorithm implementation and understanding the finer details of this project. Further thanks should go to Raja Naeem Akram, Oliver Kunz and David Morrison for taking the time to teach me Python and how to run my source code on an Ubuntu server.

I am grateful for the time that David Stranack, Thomas Bingham and James Boyle have spent proof reading this project, and for the continuous support from my partner, Lisa Moxham. This has allowed me to dedicate such a great deal of time to both research and writing. My final thanks goes to the RHBNC Trust for their financial support – without which, neither the course nor this project would have been have feasible.

Thank you to all of those who I have mentioned above, and to anyone else within the ISG who has assisted my learning over the last 12 months.

Contents

Executive Summary

A secure implementation of SSL/TLS is a critical part of modern internet security. Consequently, one would hope that a 20-year-old side-channel attack against the protocol would be easy to defend against. In this project, we establish that this is not the case, illustrate how to exploit such side-channel information, and provide a detailed investigation into how Bleichenbacher's attack has evolved since 1998. Our example, which we build upon throughout, demonstrates the improvements to the attack in both a technical and non-technical way. Furthermore, our research into optimising the algorithm provides tangible evidence regarding its most efficient implementation.

The main findings of this project revolve around the security concerns brought about by known padding schemes, and that the exploitation of side-channel leakage during an SSL/TLS handshake severely damages the security of hybrid encryption that utilises RSA. We found that many high-profile websites and implementations are or have been vulnerable, including Facebook, PayPal, Java Secure Socket Extension and OpenSSL. As such, the pervasiveness of this attack warrants careful consideration by all those who are responsible for implementing SSL/TLS – including TLS 1.3 and its deprecation of RSA encryption. The attack exists in many forms, and this is something that we highlight as we analyse literature from the last 20 years. Its application is also not restricted to the SSL/TLS protocol, and our overview of its ability to exploit the XML standard and the QUIC protocol is testament to this. Furthermore, we discovered a mathematical error in the original algorithm, and our research and experimentation towards the end of the project provides both insight and improvements to the available literature.

With this in mind, the purpose of this project is to present a comprehensive understanding of the attack and ultimately provide evidence regarding its cause. Although we seek the most optimised version of the attack algorithm, this project was written in an attempt to gain and provide awareness of poor SSL/TLS implementations. Subsequently, our intention is to uncover what must be done to successfully defend against an adversary with the knowledge to invoke such an attack.

Chapter 1

Introduction

Since Daniel Bleichenbacher introduced his "Million Message Attack" in 1998 [1], a number of improvements to the original algorithm have been published. The attack is an example of an adaptive chosen ciphertext attack and it takes advantages of side-channel leakage in an SSL/TLS implementation. By exploiting this information and the known structure of a common RSA padding scheme, it allows one to conduct private key operations without knowledge of the private key [1]. As such, an attacker can not only decrypt messages that are encrypted with an RSA public key, but also successfully forge a digital signature for any chosen message. It was called the "Million Message Attack" because to be successful it required around 1 million SSL/TLS connections, but today this number is significantly lower.

Although there is some good literature detailing optimisations, statistics and practical research, it is somewhat disjointed. At times some subtle details are omitted so an explanation would improve the understanding of the reader. Furthermore, throughout the literature, the original algorithm in [1] is regularly referred to but there are no published examples to aid the understanding of how the algorithm actually works. As a result, there is a need to; bridge the gaps between current publications; provide a coherent and detailed overview of the subject area; explore working examples of the original algorithm and its optimisations; and conduct some research into the most recent of these optimisations.

1.1 Objectives and Motivation

Our first objective is to provide a comprehensive overview of Bleichenbacher-style attacks, including the original algorithm and how it has evolved over the last 20 years. We will present a full example of an attack using Python, investigate how the algorithm has been optimised, and discuss how variations of this attack still exist in modern SSL/TLS protocol implementations. The second objective we intend to achieve is to analyse the efficiency of the attack under different parameters and quantitatively investigate the improvements offered by the proposed optimisations. Finally, the third objective is to provide research into the most effective way to implement the aforementioned optimisations.

With such an array of literature on the topic of Bleichenbacher-style attacks, our primary motivation is to present a single point-of-reference that will suffice from

both a technical and practical perspective. Therefore we will be sure to not skip over the computationally difficult details. A secondary motivation is to improve our understanding of the literature that is available, and to demonstrate the practical effects of such attacks on real-world security. It is highly unlikely that we will see RSA or the SSL/TLS protocols disappearing any time soon. Therefore, as security professionals, we would like to be able to understand precisely why we are seeing a 20-year-old attack still happening today.

1.2 Methodology

In order to achieve the above objectives, we will first look to introduce the individual puzzle pieces that are fundamental to understanding Blechenbacher's attack. Once these are established, we will conduct detailed analysis alongside a comprehensive literature study of the original paper and all subsequent publications. This alone will provide a good level of understanding, but we will also provide an example of a simulated attack, of which we will continue to develop as the later optimisations are interpreted and analysed. This will then lead on to a discussion around more recent realisations of practical Bleichenbacher-style attacks; the aim of which is to provide education regarding how these attacks are still a threat to modern security. As such, we will present the diagnosis for the most recent assortment of Bleichenbacher-style vulnerabilities; in particular why 33% of internet servers were vulnerable in 2016 [2], and why Facebook and PayPal were vulnerable in 2017 [3].

Following a successful Python implementation of the original algorithm, we will also provide our source code for the additional optimisations. This will then allow for some basic statistical research around the effectiveness of these optimisations. Furthermore, we will vary the SSL/TLS model across all variations of the algorithm in such a way that it will enable us to present statistics on a number of scenarios that can be observed in real SSL/TLS implementations. Finally, in order to add some clarity to the vague areas of the most recent optimisation of the algorithm, we will present statistics from our own research after having conducted thousands of experiments.

1.3 Structure of the Project

This project will begin with some preliminaries that are of fundamental importance in understanding Bleichenbacher-style attacks. Therefore, Chapter 2 will provide details on RSA, Public Key Cryptography Standards and the SSL/TLS protocols. Once we have developed an understanding of the background to Bleichenbacher attacks, Chapter 3 provides an introduction, description, analysis and example of the original attack from 1998. Chapter 4 will then push the focus towards three major improvements to the efficiency of the algorithm. These improvements will be further demonstrated by optimising the same example presented in Chapter 3. In Chapter 5 we will illustrate the practical Bleichenbacher-style attacks that have been discovered over the last 4 years, including a cross-protocol attack that exploits weaknesses in SSLv2 to compromise a previously secure TLS session key. There are additional applications of Bleichenbacher-style attacks outside of SSL/TLS, so in

Chapter 6 we will briefly explore these. In Chapter 6 we will also provide analysis around the changes to TLS 1.3, and ultimately how these changes help but do not eliminate Bleichenbacher-style attacks. In Chapter 7 we will exhibit and interpret the results of our attack simulations across multiple optimisations and scenarios, then finally in Chapter 8 we will provide some validation regarding the best way to optimise the algorithm following the outcome of our research. The project will then end with a conclusion, bibliography and appendix.

Chapter 2

Preliminaries

2.1 RSA Encryption

2.1.1 The Basics

In 1978, Rivest, Shamir and Adleman published a ground-breaking paper [4] that detailed a means of revealing a public encryption key without revealing the corresponding private decryption key. Their intuition equally applies to digital signatures, whereby revealing the verification key does not reveal the private signature key. Although there are alternatives to RSA encryption and signing, such techniques have become a cornerstone of internet security. To set up an RSA cryptosystem, we must do the following:

1. Generate two large random prime numbers p and q, and let $n = pq$. For a good level of security, p and q should ideally be a minimum of 1024 bits in length.

2. Choose a public encryption key e such that e and $(p-1)(q-1)$ are coprime, and $1 < e < (p-1)(q-1)$. It is worth noting that e does not need to be random, and a popular value of is e is $2^{16}+1 = 65537$, which allows for faster encryption than other values of a similar magnitude.

3. We set our public key as (n, e).

4. In order to generate the private key, we must calculate $d \equiv e^{-1} \bmod (p-1)(q-1)$. By inputting e, p and q into the Extended Euclidean Algorithm, we can calculate the unique value d, and we set our private key as (p, q, d).

Once we have generated the keys, encryption (and decryption) is a simple process. The message m that we wish to encrypt must first be converted to a number (or series of numbers) less than n. Then if c represents the ciphertext, $c \equiv m^e \bmod n$. Decryption simply reverses this calculation, and we have $m \equiv c^d \bmod n$.

It is important to note that the above description is often referred to as textbook cryptography – that is, simplified cryptography for educational purposes. Such an implementation is not suitable for deployment in the real world, and in practice, one should follow standards to implement an RSA cryptosystem (or any cryptosystem for that matter). We will look at these standards in Section 2.2.

2.1.2 Manipulating RSA Encryption

Bleichenbacher's attack takes advantage of the malleability of RSA, and the fact that an attacker is able to surgically manipulate the message m without ever knowing m or the private decryption key d.

Suppose an attacker intercepts $c \equiv m^e \bmod n$. Without the private decryption key, this information alone is not enough to decrypt m. However, it is possible for the attacker to multiply m by some chosen value s such that when the receiver decrypts the message, instead of returning $m \bmod n$, it returns $m \cdot s \bmod n$. This can be demonstrated as follows:

$$\begin{aligned} c \cdot s^e \bmod n &\equiv m^e \cdot s^e \bmod n \\ &\equiv (m \cdot s)^e \bmod n \end{aligned} \tag{2.1}$$

This means that by multiplying the ciphertext by s^e and reducing modulo n, an attacker is able to successfully manipulate the message m without ever knowing m or the private decryption key d. We will return to this idea again in Chapter 3, as it is crucial to understanding Bleichenbacher's attack.

2.2 Public Key Cryptography Standards

Public Key Cryptography Standards (PKCS) are a set of standards that should be followed to successfully implement public key cryptosystems. Textbook cryptography is important, but if it is implemented directly then it does not provide practical security. For example, if the same message is encrypted under the same RSA public key, then the ciphertext for both encryptions will be the same. Ultimately, this could lead to large scale dictionary attacks; so as a result, probabilistic encryption provides a means of preventing such situations from arising. PKCS #1 is the RSA cryptography standard, and in this project the focus will be on PKCS #1 v1.5 [5], which is a recommended padding standard for SSL/TLS. This standard dictates how one should encrypt or sign data when using the RSA public key cryptosystem, and more specifically, how such data should be padded. We will now provide an overview of the standard, of which the structure is of fundamental importance in understanding the logic behind Bleichenbacher's attack – which is detailed in Chapter 3. For completeness we will also provide an overview of an alternative padding scheme, which can be found in the more recent PKCS #1 v2.2 standard [6].

2.2.1 PKCS #1 v1.5

In order to provide an effective overview of the PKCS #1 v1.5 padding scheme [5], we will introduce it utilising an example with appropriate explanations throughout.

Let n be an RSA modulus and let $k \geq 12$ be the number of bytes that n is constructed from. Note that for the purposes of this project, unless otherwise stated, an RSA modulus n will be 1024 bit; so in this situation, $k = 128$. The smallest decimal value n can take is a `0x01` byte followed by 127 lots of `0x00` bytes, and the largest decimal value n can take is one less than a `0x01` byte followed by 128 lots of

0x00 bytes. Thus $256^{127} \leq n < 256^{128}$ or $2^{8 \cdot 127} \leq n < 2^{8 \cdot 128}$. More generally we say:

$$2^{8(k-1)} \leq n < 2^{8k}$$

Then, the hexadecimal block format of PKCS #1 v1.5 padding is of the form:

0x00 || BT || PS || 0x00 || D

where BT is the block type, PS is the padding string, and D is the concerned data. The block type is one byte in length, the padding string must be at least eight bytes in length for security purposes, and the length of the data block ($|D|$) will vary depending on the application (but it must not exceed $k - 11$). We can assume that the data block is at least one byte in length and therefore k must be at least twelve bytes in length. The leading 0x00 byte ensures that the decimal value of the entire block is always less than the modulus n when converted to an integer. The padding string will consist of $k - 3 - |D|$ bytes.

There are 3 block types; 0x00 and 0x01 are used for private key operations such as digital signatures, and 0x02 is used for public key operations such as encryption.

- For block type 0x00, the data block D must begin with a non-zero byte and the padding string will consist of all 0x00 bytes.
- For block type 0x01, the data block D can begin with a zero byte since this time the padding string contains all 0xff bytes.
- For block type 0x02, the data block D can also begin with a zero byte because the padding string must now be generated pseudo-randomly with no 0x00 bytes. There will then be a 0x00 byte that separates the padding string from the data block.

Bleichenbacher's attack exposes weaknesses in the padding and encryption used to initiate an SSL/TLS session, so block type 0x02 is of most interest to us here. In order to encrypt a message m, we first prepend 0x0002, a random padding string of length $k - 3 - |m|$, a 0x00 delimiter byte, then the message m. This process is known as encoding, and the size of the encoding (in bytes) is equal to the size of the modulus. For example, if the message is a267b29891b1, an example of a PKCS conforming encoding is

0x0002 || 6d2209ac16...bf534954aa73 || 0x00 || a267b29891b1

The next step is to convert this string from hexadecimal to decimal, compute the classic RSA encryption computation, and then convert the result back into a hexadecimal ciphertext. To decrypt, we simply convert the ciphertext from hexadecimal to decimal, compute the classic RSA decryption computation, convert from decimal back to hexadecimal, and then the blocks should be parsed to check that the structure discussed above is present. If the decrypted ciphertext is structured correctly then we say that the ciphertext is PKCS conforming. By enforcing a padding structure, it provides the receiver with a means of confirming the validity of the plaintext without the need for additional symmetric or asymmetric integrity checks.

The standard also discusses a similar method for producing a signature of a message (using block type `0x01`). The main difference is that we utilise a hash function to produce a digest of the message prior to encoding. The encoding is then converted to decimal, signed with the private signature key, then converted back to hexadecimal. This means that a receiver can verify the signature by calculating the decimal value of the same encoding, then using the public verification key on the decimal value of the digital signature to compare. If they match then the signature is verified. Although this process of producing a digital signature is not particularly important to Bleichenbacher's attack, its relevance will become apparent when we look at signature forgeries in Section 5.3.

2.2.2 PKCS #1 v2.2

PKCS #1 v2.2 is the newest version of the standard [6] and within this version there exists an alternative padding scheme for RSA encryption known as RSA Optimal Asymmetric Encryption Padding (RSA-OAEP). Again, we will explain how to pad and encrypt using this slightly more complicated scheme by means of an example.

Suppose that n represents an RSA modulus, e is the public encryption key, d is the private decryption key, and k is the byte length of the modulus n. Further suppose that $h(x)$ is a hash function such that the digest is j bytes in length. The sender can also generate a label `L` to be associated with the message m. Such a label is optional, so if it is not provided then it can be set to the empty string by default. For this example, we will set $k = 128$, $j = 32$, and the message m to be 48 bytes in length.

First, the sender should check that `L` is not greater than the maximum input of the hash function and that the length of the message m ($|m|$) to be encrypted is not greater than $k - 2j - 2$ bytes (or 62 bytes in this case). The reasoning behind the restriction on $|m|$ should become clear shortly. The sender must then generate a padding string (`PS`) consisting of $k - |m| - 2j - 2$ `0x00` bytes. Such a padding string may have length zero, but in our case it contain 14 `0x00` bytes. Once we have these parameters, we are able to construct the data block as follows:

$$\texttt{DB} = h(\texttt{L}) \;||\; \texttt{PS} \;||\; \texttt{0x01} \;||\; m$$

In our example, the data block consists of $32+14+1+48 = 95$ bytes. Next we mask the data block, and to do this we first generate a random string of length j bytes, and use this as the `seed` for a mask generation function (MGF). Such a function is deterministic and takes 2 arguments; one of which is a `seed` and the other is our desired output length. In a way, it is similar to a hash function, but with the added flexibility of allowing us to also input the length of the digest we desire. We want the digest (denoted `dbMask`) to be the same length as the data block. We can then `XOR` this to our data block to mask it. In our case, `dbMask` should be 95 bytes, but more generally it is $k - j - 1$ bytes. Thus it follows that:

$$\begin{aligned} \texttt{dbMask} &= \texttt{MGF(seed, 95)} \\ \texttt{maskedDB} &= \texttt{DB} \oplus \texttt{dbMask} \end{aligned}$$

However, it now means that the seed must also be sent, and therefore it must also be masked. To do this, again we can use the MGF, but instead we use the already calculated maskedDB as the seed, and the desired length of the digest will be $j = 32$. The result of this can then be XOR'ed to the seed to produce a masked seed, as follows:

$$\begin{aligned} \texttt{seedMask} &= \texttt{MGF(maskedDB, 32)} \\ \texttt{maskedSeed} &= \texttt{seed} \oplus \texttt{seedMask} \end{aligned}$$

Finally, we concatenate a 0x00 byte, the maskedSeed and the maskedDB, then the result is denoted the encoded message (EM), which will be of length k.

$$\texttt{EM = 0x00 || maskedSeed || maskedDB}$$

All that remains is to convert EM from hexadecimal to decimal, compute the classic RSA encryption computation, and then convert the result back into a hexadecimal ciphertext. The ciphertext is then sent to the receiver, along with the label L if applicable.

To decrypt, first the receiver must convert the ciphertext from hexadecimal to decimal, compute the classic RSA decryption computation, and then convert the result back into a hexadecimal plaintext. Such a process will return EM. Next the receiver must derive the seed, which can be done by computing the seedMask, then XOR'ing that to the maskedSeed.

$$\begin{aligned} \texttt{seedMask} &= \texttt{MGF(maskedDB, 32)} \\ \texttt{seed} &= \texttt{maskedSeed} \oplus \texttt{seedMask} \end{aligned}$$

Now the receiver has the seed, they are able to derive DB by first computing dbMask, then by XOR'ing that to the maskedDB.

$$\begin{aligned} \texttt{dbMask} &= \texttt{MGF(seed, 95)} \\ \texttt{DB} &= \texttt{maskedDB} \oplus \texttt{dbMask} \\ &= h(\texttt{L}) \texttt{ || PS || 0x01 || } m \end{aligned}$$

The receiver should of course check the padding structure, and the standard recommends the following 3 checks:

1. There must be a 0x01 byte to separate the PS from m.
2. The receiver must compute $h(\texttt{L})$ independently (denoted $h'(\texttt{L})$), then check that $h(\texttt{L}) = h'(\texttt{L})$.
3. The first byte of the originally decrypted encoded message EM must be 0x00.

If any of the checks fail then an error should be returned. However, it is important that an attacker is not able to distinguish which of the above checks failed.

When we look at Bleichenbacher-style attacks in Chapter 3 onwards, it may seem that a sensible countermeasure would be to switch from PKCS #1 v1.5 padding to PKCS #1 v2.2 padding. However, it should be noted that in 2001, James Manger

showed that the PKCS #1 v2.2 padding scheme is potentially even more susceptible to a different chosen ciphertext attack [7]. His attack required significantly less effort than the attacks that we will look at against the PKCS #1 v1.5 padding scheme. As a result, although the use of PKCS #1 v1.5 may be required to maintain compatibility with legacy system, it is perhaps due to Manger's attack that the PKCS #1 v1.5 padding scheme is still well recommended within the standard and continues to be heavily relied upon.

2.2.3 A PKCS #1 v1.5 Padding Oracle

A padding oracle is an entity that an attacker can query (such as a server), and it will simply inform the attacker as to whether the received ciphertext is PKCS #1 v1.5 conforming [1] (which for the rest of this project will be simplified to PKCS conforming). In reality, how strict the checks are on PKCS conformance can vary, as can the nature in which the attacker may differentiate between a valid and invalid ciphertext. For example, consider the scenario where a server only checks to see if the padding begins with `0x0002`. If it does not begin with `0x0002` then the server responds with an error; otherwise the attacker receives no error. If c is a random ciphertext, n is the RSA modulus and d is the respective private RSA decryption key, we can model such an oracle as follows:

$$O(c) = \begin{cases} 1 & \text{if } m \equiv c^d \bmod n \quad \text{starts with } \texttt{0x0002} \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

If the oracle responds to the random ciphertext c with a 1 (no error), then the attacker has learnt that the ciphertext passed the PKCS conformance checks, and as a result, they can deduce that the first two bytes are `0x0002`. This is an example of the strongest possible oracle, and it does not properly check for PKCS conformance. However, the strength of an oracle can vary dramatically. In Section 4.2 we will focus on a piece of literature published by Bardou et al. in 2012. However, we will introduce their simple notation for oracle strengths in this section to demonstrate the oracle variations that one could observe. The below is a summary of Section 2.4 of their paper [8].

Naturally an oracle will return an error if the decrypted plaintext does not begin with `0x0002`, so we will assume that this is always checked. However, as we saw in Section 2.2.1, there are additional checks that could (and should) be conducted to confirm that the padding is valid. Bardou et al. characterised these checks using either a **T** to represent the check being skipped, or an **F** to represent the check not being skipped. The three checks that they observed were as follows:

1. Check for the `0x00` delimiter byte somewhere after the first ten bytes.

2. Check that the non-zero padding does not contain a `0x00` byte.

3. Check that the length of the message is equal to some pre-determined length. That is, check that the `0x00` delimiter is in a specific position. So if the message being padded is strictly 48 bytes in length, the `0x00` delimiter should contain no more or no less than 48 bytes after it.

Based on the above, Bardou et al. define 5 oracles:

TTT: This oracle skips all 3 checks and so it will return 1 for any plaintext that begins with `0x0002`.

TFT: This oracle only checks that the non-zero padding does not contain a `0x00` byte. So it will return 1 for any correctly padded plaintext where the message is of any length, as well as plaintexts that do not contain a `0x00` delimiter.

FTT: This oracle only checks that the padded plaintext contains a `0x00` delimiter byte somewhere after the first 10 bytes. So it will return 1 for any correctly padded plaintext where the message is of any length, as well as plaintexts that contain a `0x00` byte within the first 8 padding bytes.

FFT: This oracle will check that the padded plaintext contains a `0x00` delimiter byte somewhere after the first 10 bytes, and that the non-zero padding does not contain a `0x00` byte. Hence it will return 1 for any correctly padded plaintext where the message is of any length. This is actually the same as the oracle that Bleichenbacher assumed in [1].

FFF: This oracle will check that the padded plaintext contains a `0x00` delimiter byte in a specific position, and that it does not contain a `0x00` byte in the non-zero padding prior to the delimiter byte. Hence it will only return 1 for a correctly padded plaintext where the message is of a pre-determined length. This is an extremely strict oracle that renders Bleichenbacher's attack practically infeasible.

Intuitively, one can see that a stricter oracle reduces the chances of $O(c) = 1$ for a random ciphertext c, thus weakening the scenario for the attacker. However, based on the above, it is clear that even with a stricter oracle, it is still possible that an attacker could differentiate between a valid and invalid PKCS conforming ciphertext using error messages. It should be noted that it may not necessarily be based on whether an attacker receives an error message, but if the server responds in any way that differentiates a valid ciphertext to an invalid ciphertext (such as response time, dropping the connection, etc.), then that server may be considered a padding oracle and some variation of equation (2.2) holds.

2.3 SSL/TLS

Secure Sockets Layer / Transport Layer Security (SSL/TLS) is a cryptographic protocol that establishes and maintains a secure communication channel between two parties (typically a client and a server) [9]. Historically, RSA has been used as a public key encryption mechanism during the handshake protocol, and it is this subsection of SSL/TLS that we are particularly interested in. Figure 2.1 illustrates the SSL/TLS handshake protocol, and the messages that are marked with a * are optional. For the purposes of this project, we are only interested in a server that unilaterally authenticates itself according to implementation advice in [10]. As such, we will focus on the messages that are mandatory. For clarity, we will first describe the purpose of each of these messages, then look in more detail at the one specific

message which is the target of Bleichenbacher's attack.

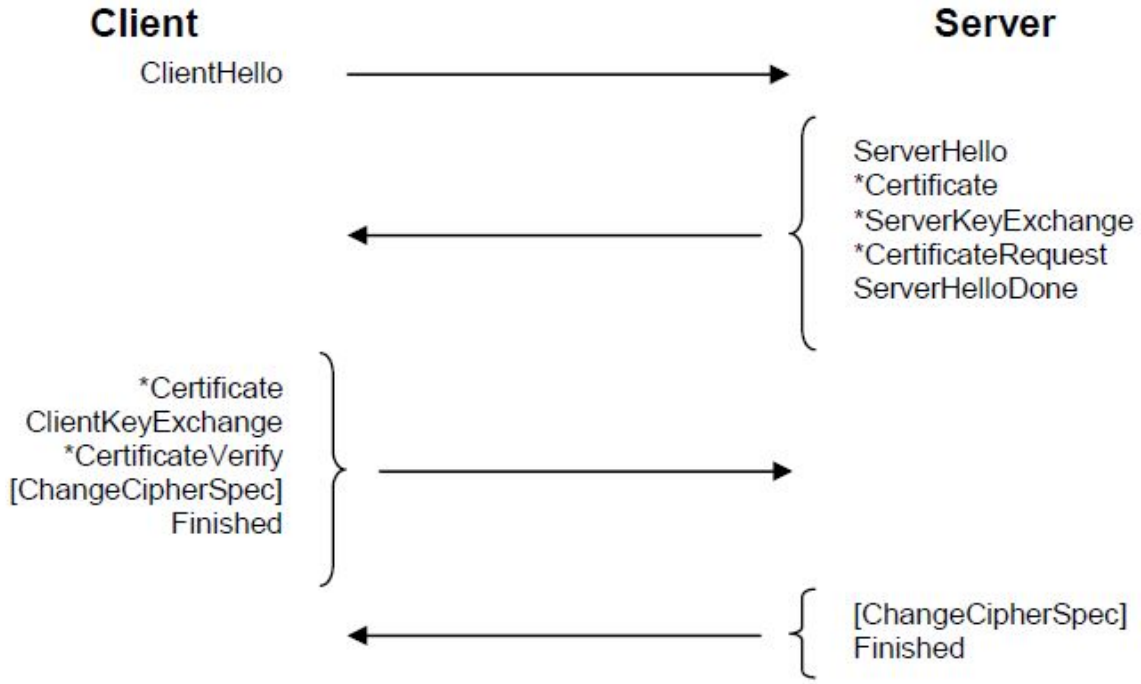

Figure 2.1: The SSL/TLS protocol (taken from [11]).

1. `ClientHello`: This is sent when the client first connects to the server. This message should contain the SSL/TLS version number, a client nonce and a list of supported ciphersuites.

2. `ServerHello / ServerHelloDone`: This is a response to the `ClientHello`, assuming there is an agreed set of algorithms. The message will contain the server version number, a server nonce, a session ID, a selected ciphersuite and the public key certificate of the server. The final part of the message is `ServerHelloDone`.

3. `ClientKeyExchange / ClientFinished`: This message contains a randomly generated pre-master secret (`PMS`), encrypted under the server's public key. The `PMS` is generated by the client and should be 48 bytes in length. The `PMS` should then be appropriately padded (see Section 2.2) and the padding process allows the server to check that the message it receives and decrypts is valid. Finally, the `ClientFinished` message is sent, which will contain a `MAC` on all messages that have been sent to far – the key to which is derived from the `PMS`.

4. `ServerFinished`: Assuming the server can decrypt the `ClientKeyExchange` message and confirm its validity, the server is also able to deduce the `PMS`. As a result, the final message sent by the server is a `MAC` of all the messages which have been sent – again deriving the key from the `PMS`.

The section of the handshake protocol that Bleichenbacher's attack looked to exploit was the `ClientKeyExchange` message and its subsequent decryption (as detailed in

Section 7.4.7 of [10]). As far as the server is concerned, if the `ClientKeyExchange` message has a valid structure then it can extract the `PMS` and continue with the protocol. However, a problem arises when the `ClientKeyExchange` message does not have a valid structure. The obvious reaction would be to return an error to the client, but such a reaction is precisely what enabled Bleichenbacher's attack. In fact, the server must continue with the protocol, being sure not to leak any information as to the invalidity of the `ClientKeyExchange`. Some examples of information leakage include log file differences, timing differences, dropping the connection or error messages being sent.

Although implementing SSL/TLS in such a way that thwarts Bleichenbacher's attack is not a trivial task, guidance on a successful implementation is outlined in Section 7.4.7.1 of [10]. Hence, it should be made clear that the version of the attack we are discussing here and will be discussing in Chapter 3 should no longer be feasible. However, in Chapter 5 we will look at some more recent developments that build on Bleichenbacher's foundations, and there we will see that sufficient countermeasures against Bleichenbacher-style attacks still warrant careful consideration 20 years on.

Chapter 3

Bleichenbacher's Padding Oracle Attack

In 1998, Daniel Bleichenbacher published an adaptive chosen ciphertext attack that exploited weaknesses in RSA-based protocols [1]. In this section, we will describe and analyse the details of this historical attack, as well as presenting an implementation in Section 3.3 to aid understanding.

3.1 The Idea Behind the Attack

Suppose that n and e make up an RSA public encryption key of a server and d is the private decryption key of the server. Furthermore, suppose that such a server incorporates PKCS #1 v1.5 padding and that it also suffices as a padding oracle whereby, for a chosen ciphertext c, the server indicates whether the corresponding plaintext is correctly padded – as discussed in Section 2.2. If these properties hold, then any message which is encrypted using the RSA public key can be deduced without knowing the respective private key d. Additionally, if the server also uses the private key d for signing messages, then this attack can also successfully forge a digital signature – again without knowing the private key d. One can see that such a situation rapidly deteriorated the security of SSL/TLS because the `PMS` represents a single point of failure if it is compromised. As a result, if an attacker were able to record an SSL/TLS session then mount such an attack to recover the `PMS`, they would be able to decrypt all communications between the client and the server. With the above in mind, the basis for the attack is as follows:

Suppose an attacker intercepts the `ClientKeyExchange` message m from the SSL/TLS handshake protocol. By its construction, we know that the ciphertext is PKCS conforming and therefore, if the attacker were to query the oracle with the ciphertext, the oracle would return 1. As a result, the attacker knows that the first two bytes of the decrypted message m are `0x0002`. However, if the attacker were to intercept an arbitrary ciphertext c which is not PKCS conforming, then the attacker can choose an integer s, compute $c' \equiv c \cdot s^e \bmod n$, and then query the oracle with c'. If the oracle returns 1 then the attacker knows that c' is PKCS conforming, and from equation (2.1) in Section 2.1.2, this means the attacker now knows that the first two bytes of $m \cdot s$ are `0x0002` for some chosen integer s. Note that the first scenario is an example of the second scenario with $s = 1$.

Based on the above, if $k = 128$ is the byte length of our modulus n, then we know that the smallest value that $m \cdot s$ can take is `0x0002`, followed by 126 `0x00` bytes. If this is the case then we have:

$$\begin{aligned} m \cdot s &\geq 2^{8k-15} \\ &\geq 2 \cdot 2^{8k-16} \\ &\geq 2 \cdot 2^{8(k-2)} \end{aligned} \tag{3.1}$$

Similarly, the largest value that $m \cdot s$ can take is one less than `0x0003`, followed by 126 `0x00` bytes. Hence it follows that:

$$\begin{aligned} m \cdot s &\leq 2^{8k-15} + 2^{8k-16} - 1 \\ &\leq 2 \cdot 2^{8k-16} + 2^{8k-16} - 1 \\ &\leq 2 \cdot 2^{8(k-2)} + 2^{8(k-2)} - 1 \\ &\leq 3 \cdot 2^{8(k-2)} - 1 \end{aligned} \tag{3.2}$$

By setting $B = 2^{8(k-2)}$, we now have assurance that if $c \cdot s^e \bmod n = (m \cdot s)^e \bmod n$ is PKCS conforming, then it follows that $2B \leq m \cdot s < 3B$. After finding the first s value, denoted s_0, we set $c_0 \equiv c \cdot (s_0)^e \bmod n$. The attack then proceeds by collecting many more s values such that $c_0(s)^e$ is PKCS conforming, and this iterative process allows the attacker to reduce the interval that contains $m_0 \equiv m \cdot s_0 \bmod n$ until there is only one possible value, say a. That is, the upper bound of the interval is the same as the lower bound of the interval. As a result we have $m_0 = a \equiv m \cdot s_0 \bmod n$, and so we can calculate $m \equiv a \cdot (s_0)^{-1} \bmod n$, where $m \equiv c^d \bmod n$. Again, it is worth noting that if c was already PKCS conforming then $s_0 = 1$, and $m_0 = a = m$. In an SSL/TLS handshake protocol, m corresponds to a padded `PMS`, so the attacker can extract the `PMS` by removing the padding, and now they are able to decrypt any subsequent communications that took place. We will now provide the details of the algorithm which Bleichenbacher presented in 1998, then in Section 3.3 we will demonstrate an implementation of the attack.

3.2 The Details of the Attack

First we will present the algorithm which was published in [1], and then we will explain the logic behind each step.

3.2.1 The Attack Algorithm

For clarity, Bleichenbacher denotes the variable M_i to be the set of closed intervals, which are computed after a successful s_i has been found, such that m_0 is contained in one of the intervals of M_i.

Step 1: Blinding
Given an integer c, choose different random integers s_0, then check, by accessing the oracle, whether $c(s_0)^e \bmod n$ is PKCS conforming. Again note that from Section

2.1.2, $(m \cdot s_0)^e = c(s_0)^e \bmod n$. For the first successful value s_0, set

$$\begin{aligned} c_0 &\leftarrow c(s_0)^e \bmod n \\ M_0 &\leftarrow \{[2B, 3B-1]\} \\ i &\leftarrow 1 \end{aligned}$$

So, at this stage we know that $2B \leq m \cdot s_0 \leq 3B-1$. As mentioned in Section 3.1, if c is already PKCS conforming then we can set $s_0 = 1$ to simplify the above.

Step 2: Searching for PKCS conforming messages

Step 2a: Starting the search
If $i = 1$, then search for the smallest positive integer $s_1 \geq n/(3B)$, such that the ciphertext $c_0(s_1)^e \bmod n$ is PKCS conforming.

Step 2b: Searching with more than one interval left
Otherwise, if $i > 1$ and the number of intervals in M_{i-1} is at least 2, then search for the smallest integer $s_i > s_{i-1}$, such that $c_0(s_i)^e \bmod n$ is PKCS conforming.

Step 2c: Searching with one interval left
Otherwise, if M_{i-1} contains exactly one interval (say $M_{i-1} = \{[a, b]\}$), then choose integer values r_i, s_i such that

$$r_i \geq 2\frac{bs_{i-1} - B}{n}$$

and

$$\frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

until the ciphertext $c_0(s_i)^e \bmod n$ is PKCS conforming[1].

Step 3: Narrowing the set of solutions
After s_i has been found, the set M_i is computed as

$$M_i \leftarrow \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil\right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor\right)\right] \right\} \tag{3.3}$$

$$\text{for all } [a, b] \in M_{i-1} \text{ and } \frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$$

Step 4: Computing the solution
If M_i contains one interval of length 1 (so $M_i = \{[a, a]\}$), then set $m \leftarrow a(s_0)^{-1} \bmod n$ and return m as the solution to $m \equiv c^d \bmod n$. Otherwise, set $i \leftarrow i + 1$ and go to step 2.

[1]The bound we have presented for r_i is actually a correction to the bound given in the original paper and all subsequent literature. This will be explained in step 2c of Section 3.2.2.

3.2.2 An Explanation of the Algorithm

In this section, we will analyse how each step of the algorithm works.

Step 1: Blinding
This step is quite self-explanatory and probably the easiest step to understand. In order to commence the attack, we require a ciphertext to be PKCS conforming. If we are trying to decrypt an intercepted ciphertext c, such as a `ClientKeyExchange` message in the SSL/TLS handshake, then by its own construction it should be PKCS conforming. As a result we can set $s_0 = 1$. However, if we are trying to forge a digital signature, then c will not be PKCS conforming. In such a situation, c represents the initial encoding of a digital signature prior to the computation which would typically be computed using the private signature key. Therefore, as discussed in Section 2.2.1, c will begin with `0x0000` or `0x0001`. Clearly we can no longer have $s_0 = 1$, so instead we should try random or increasing integer values of s_0 until $c(s_0)^e \bmod n$ is PKCS conforming. Once we find such a value of s_0, we set $c_0 \equiv c(s_0)^e \bmod n$ and then we are done.

So $m \equiv c^d \bmod n$ is the message that we wish to decrypt, and from Section 2.1.2, by finding an s_0 such that $c_0 \equiv c(s_0)^e \bmod n$ is PKCS conforming tells us that $m_0 \equiv m \cdot s_0 \bmod n$ begins with `0x0002`. We know the value of s_0, so if we can calculate m_0 then we can deduce m. Therefore, steps 2, 3 and 4 work on reducing the possible values of m_0 until there is only one option.

Step 2a: Starting the search
This step will always happen immediately after step 1, and since it only happens when $i = 1$, we know that it will only take place once. The aim of this step is to find the smallest positive integer s_1 such that $c_0(s_1)^e \bmod n$ is PKCS conforming. In his algorithm, Bleichenbacher informs us that $s_1 \geq n/(3B)$ because, for smaller values of s_1, $c_0(s_1)^e \bmod n$ cannot be PKCS conforming. However, what is omitted from the paper is why $c_0(s_1)^e \bmod n$ cannot be PKCS conforming for smaller values of s_1. Therefore we will provide the reasoning here:

We wish to find s_1 such that $m_1 = m_0 \cdot s_1 \bmod n$ is a PKCS conforming plaintext. Hence for some positive integer r, it follows that

$$m_1 = m_0 \cdot s_1 - rn$$

We do not know the value of r but we can create upper and lower bounds for r.

$$r = \frac{m_0 s_1 - m_1}{n} \tag{3.4}$$

Since both m_0 and m_1 are PKCS conforming, we know that $2B \leq m_0, m_1 < 3B$, and so we can calculate the highest and lowest possible values of r:

$$\frac{2B \cdot s_1 - (3B-1)}{n} \leq r \leq \frac{(3B-1)s_1 - 2B}{n}$$
$$\text{which implies } r \leq \frac{3B \cdot s_1}{n}$$

So if $s_1 < n/(3B)$ then $r < 1$ and therefore $r = 0$. This means that $m_1 = m_0 \cdot s_1$ without the need for modular arithmetic. Therefore $m_1 < n$, and since $s_1 > 1$, it should now be clear that m_1 can never be less than $3B$, which means it can never be PKCS conforming. In fact, the smallest m_1 can be is $4B$, and this is when $m_0 = 2B$ and $s_1 = 2$. Therefore, it makes sense to start our search for s_1 with $\lceil n/3B \rceil$.

Starting with $s_1 = \lceil n/3B \rceil$, we query the oracle to check for PKCS conformance, and increase s_1 by 1 each time until we find a ciphertext $c_0(s_1)^e$ which is PKCS conforming. Unfortunately we do not yet have a means of narrowing down the search for such a value beyond this, and as such, the search for s_1 is a naive search whereby the first two bytes of $c_0(s_i)^e \bmod n$ will be uniformly distributed. As a result, the chances of finding a successful s_1 (where the first two bytes of $c_0(s_1)^e$ are `0x0002`) is $(1/256)^2 = 1/65536$. Once we have found s_1, we move on to step 3. We will look at step 3 first before returning to step 2b and step 2c as it will help to aid understanding.

Step 3: Narrowing the set of solutions
The first time we enter step 3 we have found s_1 and we know that $m_0 \in M_0 = [a, b]$, where $a = 2B$ and $b = 3B - 1$. We also know that $m_1 = m_0 \cdot s_1 - rn$, and that both m_0 and m_1 are PKCS conforming plaintexts. We wish to generate a new set of intervals M_1, but it should be noted that we do not know the values of m_0 or m_1; therefore we do not know the value of r.

Since $m_1 = m_0 \cdot s_1 - rn$, it follows that $m_0 = \dfrac{m_1 + rn}{s_1}$.

Furthermore, we know that $2B \leq m_1 \leq 3B - 1$, therefore, it must hold that

$$\frac{2B + rn}{s_1} \leq \frac{m_1 + rn}{s_1} \leq \frac{3B - 1 + rn}{s_1}$$

$$\frac{2B + rn}{s_1} \leq m_0 \leq \frac{3B - 1 + rn}{s_1}$$

So we now have a new bound for m_0, but we still do not know the value of r. However, using equation (3.4) from step 2a, we can bound r using the lowest value of $m_0 = a$ and the highest value of $m_1 = 3B - 1$, then the highest value of $m_0 = b$ and the lowest value of $m_1 = 2B$. Hence the range for r is

$$\frac{a \cdot s_1 - 3B + 1}{n} \leq r \leq \frac{b \cdot s_1 - 2B}{n} \tag{3.5}$$

By using each possible value of r, we are now able to calculate new intervals for m_0. So if there are three possible values for r then we will generate three new intervals for m_0. Clearly we already have a lower and upper bound for m_0, so we are only interested in lower bounds which are greater than the previous lower bound, and upper bounds which are less than the previous upper bound – otherwise the new bound can be ignored (hence the maximum and minimum functions). Once we have our new intervals, we take the union of the intervals and the result is M_1.

Of course the above can be generalised for when we enter step 3 and $i > 1$. In this case we should replace s_1 with s_i, m_1 with m_i, M_0 with M_{i-1} and M_1 with

M_i. It is also worth mentioning that it is entirely plausible that M_{i-1} will consist of multiple intervals for m_0. If this is the case then all possible values of r should be calculated for each of the intervals from M_{i-1}, and using these intervals and their respective r values, we can create a set of new intervals. Then we can take the union of all the new intervals, and as a result M_i may consist of a single interval for m_0, or again contain multiple possible intervals for m_0. In general, the number of possible r values is very low, and often there is only one possible value. However, all possible values of r must be utilised to avoid missing an interval which does actually contain m_0. When we have finished step 3 we go to step 4, and unless we only have one interval containing only one value, we set $i = i + 1$ and then go back to step 2. Therefore we will look at step 2b and step 2c before step 4.

Step 2b: Searching with more than one interval left
This is a simple (but time-consuming) step where we must search for the smallest positive integer $s_i > s_{i-1}$ such that the ciphertext $c_0(s_i)^e \bmod n$ is PKCS conforming. Since there are multiple possible intervals which contain m_0, in general we are unable to bound s_i in such a way that speeds up this step. Thus, as was the case in step 2a, the search for s_i is a naive search; so the chances of finding a successful s_i is 1/65536. Once we have found a successful value of s_i, we proceed to step 3.

Step 2c: Searching with one interval left
At first glance, this step seems quite complicated. However, if M_{i-1} contains only one interval, then Bleichenbacher implemented a clever technique which is able to determine s_i in an extremely efficient way, and divide the size of the interval roughly in half for each iteration of step 2c.

Notice that in equation (3.3), to form a new interval, we must divide the respective numerators by s_i. For a set value of r, if s_i is twice the size then the resulting interval will be half the size. Therefore, this step works by choosing r_i first in such a way that we guarantee s_i to be at least twice the size of s_{i-1}. This is done as follows:

We can rearrange equation (3.5) to create a bound for s_i, giving us

$$\frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a} \tag{3.6}$$

Now, since we want $s_i \geq 2s_{i-1}$, we need to bound r_i such that:

$$2s_{i-1} \leq \frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

Hence

$$\begin{aligned} 2s_{i-1} &\leq \frac{2B + r_i n}{b} \\ r_i &\geq \frac{2bs_{i-1} - 2B}{n} \\ r_i &\geq 2\frac{bs_{i-1} - B}{n} \end{aligned}$$

It should be noted that our bound for r_i is a correction on the bound given in the original paper [1], as well as all subsequent literature that we have observed. That

being said, this bound does not actually improve the attack, but does make it mathematically sound. The reason our bound for r_i does not make a difference is as follows:

By substituting our bound for r_i into equation (3.6), it simplifies to $2s_{i-1} \leq s_i$, as expected. On the other hand, by substituting the original bound for r_i into equation (3.6), it actually simplifies to $2s_{i-1} - (2B)/b \leq s_i$. Since s_{i-1} and s_i are positive integers, we would only be testing additional values of s_i if $(2B)/b \geq 1$. However, since b is the upper bound of an interval within M_{i-1}, it cannot be less than $2B$, and due to the structure of the padding, b can never equal $2B$ either. Therefore $(2B)/b < 1$, and so the first value of s_i will be the same in both situations.

So to complete step 2c, we choose the smallest possible integer r_i which satisfies the above equation, then starting with the smallest possible integer value s_i which satisfies equation (3.6), we check to see if the ciphertext $c_0(s_i)^e \bmod n$ is PKCS conforming. We continue to increase s_i by 1 until we find a PKCS conforming ciphertext. If we have tried all values of s_i satisfying equation (3.6) without success, then we increase r_i by 1 and start again with a new batch of possible values for s_i. We continue in this way until we find a valid PKCS conforming ciphertext. By defining r_i first, the search for the next value of s_i is significantly more efficient than the naive search implemented in step 2b. Furthermore, it also means that when we narrow the set of solutions in step 3, the larger value of s_i improves the efficiency here too.

Step 4: Computing the solution
This is the final step of the algorithm and we only remain in step 4 if M_i contains only one interval of length 1. So $M_i = [a, a]$, hence the upper bound and lower bound for m_0 are the same. This means that $m_0 = a$.

As discussed in step 1, we know that $m_0 \equiv m \cdot s_0 \bmod n$, which means that $a \equiv m \cdot s_0 \bmod n$. So we now know the value of a, and we know the value of s_0 from step 1. Therefore $m \equiv a(s_0)^{-1} \bmod n$, we return m as the solution to $m \equiv c^d \bmod n$, and we have successfully decrypted the ciphertext c without the private decryption key d. If c was already PKCS conforming then $s_0 = 1 = s_0^{-1} \bmod n$. Hence we can set $m = a$ as the solution to $m \equiv c^d \bmod n$ to simplify this last step.

3.3 An Implementation of the Attack

I will now provide (some of) a simulated example of this attack in an attempt to discover the `PMS` (encoded within m) from an SSL/TLS handshake, encrypted with a 1024 bit RSA public key. Our example assumes an attacker has access to an FFT oracle, and the parameters are purposely chosen to demonstrate all possible steps of the algorithm. For simplicity, we will omit the process of converting from hexadecimal to decimal and vice-versa (instead all workings will be in decimal). You can find the primes used to generate this RSA modulus and the respective private key in Appendix A. Appendix A also contains our Python source code for this example. Unfortunately, the real numbers used in this example provide poor visual clarity and perhaps inhibit understanding. Based on this, we will utilise "..." to represent most of the digits, and the full numbers can be found in Appendix B.

Suppose that n is the RSA modulus of the server, e is the public encryption key, and c is an intercepted ciphertext. As an attacker, gaining knowledge of these values is trivial, so suppose the attacker obtains the following such values:

$$n = \texttt{1584975239...} \quad e = \texttt{65537} \quad c = \texttt{5267820764...}$$

We begin in step 1 and start by setting $s_0 = 1$, then querying the oracle with $c(1)^e \bmod n$. Since this is an intercepted ciphertext, it is reasonable to assume that it is PKCS conforming. Therefore, we can set $c_0 \equiv c(1)^e \bmod n = c$, $m_0 \equiv m \cdot 1 \bmod n = m$, and we can also define the first interval for m_0 using $B = 2^{8 \cdot 126}$:

$$M_0 = \{\texttt{[5486124068..., 8229186103...]}\}$$

Although we are able to bound m_0, the size of this interval is 2^{1008} – which is clearly too large to conduct an exhaustive attack such as a chosen plaintext attack. Instead we set $i = 1$ and move onto step 2a, where we can begin reducing the size of the interval.

This is a time consuming step, but the aim is to find the value of s_1. We know that $s_1 \geq n/(3B) = 19260.4131109$. Thus we start with the value $s_1 = 19261$, and continue to query the oracle with $c_0(s_1)^e \bmod n$, increasing s_1 by one with each query. We stop when we find the first value which implies $c_0(s_1)^e \bmod n$ is PKCS conforming. In this example we find that $s_1 = 82005$.

We now move onto step 3, and here we will use s_1 to reduce the interval containing m_0. Using equation (3.3), first we need to calculate the possible values of r. By inputting a as the lower bound of M_0, b as the upper bound of M_0, and s_i as our newly found s_1, we discover that the possible values of r are 3 and 4. With this in mind, we will now form two possible intervals which may contain m_0; one for $r = 3$ and one for $r = 4$. We calculate these new intervals and then take the union of the intervals. Therefore, if the two new intervals overlap then M_1 will consist of one interval again, however, if they do not overlap then M_1 will consist of two intervals. In this example we find that the intervals do not overlap, but we have improved our bound on m_0:

$$\begin{aligned} M_1 = \{&\texttt{[5798403242..., 5798436691...],} \\ &\texttt{[7731182022..., 7731215472...]}\} \end{aligned}$$

By establishing this value of s_1, step 3 has reduced the bound on m_0 by more than 99.997%, with the number of possible options falling to just under $2^{992.7}$. We now go to step 4, immediately set $i = 2$ and then return to step 2. We never return to step 2a, so now we will either go to step 2b or step 2c. In this example M_1 contains more than one interval, and so we must go to step 2b.

We begin the search for s_2 starting with the value $s_1 + 1$. We then continue increasing s_2 by 1 until the ciphertext $c_0(s_2)^e \bmod n$ is PKCS conforming. It turns out that in this scenario $s_2 = 355351$ and we move onto step 3 again.

Now we will use s_2 to reduce the size of the interval containing m_0. However,

this time we can see that there is currently more than one possible interval left for m_0. Nonetheless, the process is very similar, and using equation (3.3), we can calculate all possible values of r for all the given intervals. For the first interval in M_1, the only possible value of r is 13, and just like before, we can now calculate the new interval. However, when we look at the second interval in M_1, it turns out that there are no possible integer values of r. As a result, m_0 cannot possibly be in that interval, thus the interval is discarded. There is no need to take the union of the intervals as we only have one, and so we return this as M_2:

$$M_2 = \{[5798417049..., 5798424768...]\}$$

We have now reduced the number of possible options for m_0 to just under $2^{989.6}$. Again we go to step 4, set $i = 3$, and then return to step 2. This time around, M_2 contains only one interval, and so we enter step 2c. In step 2c, the first thing to do is find the smallest possible integer value of r_3 which guarantees s_3 to be at least twice the size of s_2. Using s_2 and M_2, we can calculate $r_3 = 27$, then using r_3 we can calculate a bound (or set of bounds) which must contain s_3. So using equation (3.6), we know that $738034.690857 \leq s_3 < 738036.146457$, and therefore we can cycle through these possible integer values of s_3 until the ciphertext $c_0(s_3)^e \bmod n$ is PKCS conforming. If we do not find a valid s_3 then we set $r = r + 1$, generate a new bound for s_3 and try again. We continue until s_3 is valid, and in this scenario (when $r = 29$) we find that $s_3 = 792705$. Once we have established s_3, we proceed to step 3, continuing as previously discussed. This then provides us with M_3:

$$M_3 = \{[5798417049..., 5798419869...]\}$$

After 3 iterations of step 3 we have now reduced the number of possible options for m_0 to just over 2^{988}. Throughout the rest of the example, we actually continue to loop through step 2c and step 3, with the values of s_i increasing and the interval containing m_0 decreasing. In fact, after finding s_{989}, we enter step 3 for the last time and it produces our final bound for m_0. It makes sense that completion of the attack took around this number of iterations since we know from Section 3.2.2 that step 2c chooses s_i in such a way that it more than halves the size of the interval containing m_0 for each iteration. Our final bound is M_{989}:

$$M_{989} = \{[5798419205...7665573158, 5798419205...7665573158]\}$$

This time when we enter step 4 it is true that M_i contains only one interval of length 1. So here we have $m_0 = a$, we set $m \equiv a(s_0)^{-1} \bmod n$, and return m as the solution to $m \equiv c^d \bmod n$. From step 1 we have $s_0 = 1$, so $s_0^{-1} \equiv 1 \bmod n$, and therefore $m = a$:

m = 579841920589406700723097951151738986844836670449757912367054444
235659537243763227900123637103641595191709903054542208387271409740203
912969123044117570729427967848283325577889366122243437084764362681915
869339573392245686648058962247658928156442361659628221880783291029946
017265476439229817208759766557315 8

For confirmation we can check that our value of m is correct by encrypting it and comparing to the ciphertext which we started with. Finally, we will convert from

decimal to hexadecimal, and we can see that m has the appropriate padding scheme – from which we can extract the targeted `PMS`.

m = `0x00021d2536e6983f9e2a8d833006dbb868feac2d61e2aa41a85d05bc50236da522ac153cb68f7edcd0edd96add4db5dba0e2e79c0de130cd6493cfc54669178fdc2ae0ec4879d9149c7cc4f369c52a004c8eb83f20bc8f89a0cce55519359b06e4544a0a08c8d8e2b75103c097c6fdb5d1723ea033e91a20ba5e67b00c835926`

`PMS = 4c8eb83f20bc8f89a0cce55519359b06e4544a0a08c8d8e2b75103c097c6fdb5d1723ea033e91a20ba5e67b00c835926`

This particular example required 342445 queries to the oracle to decrypt the ciphertext.

Chapter 4

Improvements to the Attack

Looking back at the example in Section 3.3, one can observe that the majority of queries to the oracle took place in step 2a and 2b. In fact, of the 342445 oracle queries required to conduct the attack, 336091 (more than 98%) took place in either step 2a or step 2b. Considering we only entered those steps once and step 2c 986 times, the example clearly demonstrates the inefficiencies that step 2a and 2b contain. As a result of this, two different pieces of literature set about improving the performance of these 2 steps. The motivation behind the improvements were different for each set of authors, but in this section we will detail these improvements and summarise their impact on the original algorithm.

4.1 Improvements from 2003

As mentioned previously, the version of the attack that we have discussed so far should not be practically feasible. In fact, in his paper (Section 5 of [1]), Bleichenbacher explained that a good SSLv3 implementation would make the attack practically infeasible – even in 1998. This implementation not only conducts all the checks of an FFF oracle, but it must also check the version number which has been encoded within the `ClientKeyExchange` message. So the sender will follow the PKCS #1 v1.5 padding scheme, the message should be 48 bytes in length, and the first 2 bytes of the message will represent the SSL/TLS version number (`0x0300` for SSLv3). This can be seen in figure 4.1; and the version number was initially recommended to prevent a rollback attack (see Section 7.4.7.1 of [10]).

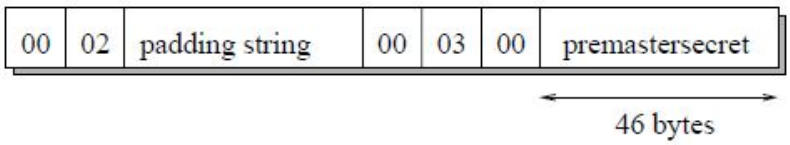

Figure 4.1: PKCS #1 v1.5 padding with the SSL/TLS version number (taken from [1]).

Bleichenbacher advised that if a ciphertext decrypted to a correctly padded PKCS conforming plaintext, but with a version number that did not match what was expected, then the server should issue an error which is indistinguishable to an error that may be issued when the plaintext is not PKCS conforming.

In such a situation, the search for each s_i becomes practically infeasible. This is because, with a naive search, in order to find an s_i such that $c_0(s_i)^e \bmod n$ passes all of these checks, $c_0(s_i)^e \bmod n$ must begin with `0x0002`, there must be a `0x00` delimiter byte 49 bytes from the end, the padding string must not contain a `0x00` byte, and the first two bytes of the message must be `0x0300`. For a 1024 bit RSA modulus, we can calculate the probability of this as follows:

$$\begin{aligned} P(s) &= \left(\frac{1}{256}\right)^2 \cdot \left(\frac{1}{256}\right) \cdot \left(\frac{255}{256}\right)^{77} \cdot \left(\frac{1}{256}\right)^2 \\ &\approx \left(\frac{1}{256}\right)^5 \cdot 0.74 \\ &\approx 6.73 \times 10^{-13} \end{aligned}$$

That works out to be roughly one in 1.5 trillion attempts to find such a value for s_i – which is clearly infeasible. However, in 2003, Klíma et al. published a paper [11] demonstrating a new side-channel attack which made attacks against SSL/TLS implementations incorporating a version number check possible. Furthermore, they also presented a number of improvements to Bleichenbacher's algorithm, particularly to step 2b. In this subsection, we will look at both of these important findings.

4.1.1 The Version Number Side-Channel Attack

Klíma et al. noticed that although SSL/TLS implementations were checking for the version numbers, it had not been properly specified precisely what should be done if the message was correctly padded but the version number was incorrect. As a result, of the 611 random servers that they tested, two thirds of them leaked information on this error in a way that was distinguishable to other PKCS padding errors. This meant that although conducting the attack was still computationally expensive, it was within the realms of feasibility. The oracle that they discovered, which they termed a Bad Version Oracle (BVO), is very similar to that of the FFF oracle discussed in Section 2.2. If k is the byte length of an RSA modulus, they describe a plaintext message m as S-PKCS conforming if it is PKCS conforming, $m_j \neq$ `0x00` for all $j \in \{3, k-49\}$, and $m_{k-48} =$ `0x00`, where m_j represent the j^{th} byte of the message m. Using this, if `major` and `minor` represent the 2 bytes that make up the SSL/TLS version number, they modelled the BVO as follows:

$$BVO(m) = \begin{cases} 1 & \text{if } m \text{ is S-PKCS conforming, } m_{k-47} \neq \texttt{major}, \text{ and } m_{k-46} \neq \texttt{minor} \\ 0 & \text{otherwise} \end{cases}$$

Their observations showed that if a message was not S-PKCS conforming then in general, the servers would generate a random `PMS` and the protocol would break down at a later point. Conversely, if the message was S-PKCS conforming with the correct version numbers, then it would extract the correctly padded `PMS` and continue with the protocol. However, if the message was S-PKCS conforming but with an incorrect version number, the servers would issue a distinguishable error. They had proven that such Bad Version Oracles existed, but to exploit this behaviour they still needed to improve the efficiency of Bleichenbacher's algorithm. As such, we will now look at how they optimised the attack to practically exploit a BVO.

4.1.2 Improving Step 2b

We have already seen how inefficient step 2b is. However, what we have not yet mentioned is that the stricter the oracle is, the more likely we are to enter step 2b (as we will demonstrate in Chapter 7). Given that the BVO modelled above hints at an extremely strict oracle, it made sense for Klíma et al. to focus on optimising step 2b.

Their idea was based on the fact that in step 2c, we can apply certain rules to bound the next value of s_i. However, if you have more than one interval left, these rules still hold for each of the intervals in isolation. After some experimenting they noticed that it was significantly more efficient to start step 2c for each possible interval in parallel – and they called this the Parallel-Threads Method. So if $|M_{i-1}| > 1$, for each interval in M_{i-1}, we start a thread of step 2c as if it were the only interval in M_{i-1}. Then each of the threads will take it in turns in a cyclic fashion to query the oracle (BVO). When any one of the intervals finds a valid s_i, we proceed to step 3 with this value of s_i, thus projecting it onto all intervals contained in M_{i-1}. Any intervals which disappear as a result of step 3 are discarded, and as normal, we proceed to step 4, then back to step 2. If the number of intervals is now back to one then we enter step 2c as before, but if it is still greater than 1, we repeat the above Parallel-Threads Method. In Section 3.3.2 of their paper [11], they do actually provide an upper bound on $|M_{i-1}|$, whereby if M_{i-1} contains more intervals than this bound, then the Parallel-Threads Method should not be used. However, the bound is very high, and after the thousands of simulations that we have conducted, it is unlikely that such a situation will arise. However, it should be mentioned that due to time restrictions, we have not implemented any simulations that emulate a BVO. It could well be that the significantly stricter criteria imposed by the BVO on the attacker may warrant this bound to be considered.

In Section 4.1 of [11], they present their results for local simulations of this attack, and for a 1024 bit RSA modulus, the mean and median number of oracle calls were 20,835,297 and 13,331,256 respectively. They also conducted the attack on a real server, which allowed for 52.68 oracle calls per second. With such a server, they concluded that half of all attacks conducted with their set up would succeed in less than 70 hours and 18 minutes. Hopefully, one can now see that with this optimisation and the existence of practical Bad Version Oracles, conducting their attack back in 2003 became practically feasible.

Furthermore, coupled with their findings that two thirds of the tested servers were vulnerable may seem worrying, and from a theoretical cryptographic perspective, it is. However, they also emphasise that if administration of a server is done properly then the number of connections and length of time required to conduct the attack should almost certainly be noticed in the log files. As a result, an attacker can be blocked from initiating any further connections to the server. Nonetheless, this ingenious optimisation equally applies to the stronger oracles discussed in Section 2.2, and in Chapter 7 we will look at the effects it has on these oracles.

Finally, it should also be noted that there were two other minor optimisation presented in this piece of literature. Ultimately, their effects are negligible, so we will not present them in detail. However, we will summarise them for completeness.

The first optimisation takes advantage of the fact that if a plaintext message m is PKCS conforming then the attacker knows that the first 2 bytes are `0x0002`. However, depending on the strength of the oracle, the attacker may also know any of the following:

- There is no `0x00` byte in the first 8 padding bytes
- The byte in position m_{k-48} is `0x00`
- The byte in position m_j is non-zero for all $j \in \{3, k-49\}$
- The bytes in position m_{k-47} and m_{k-46} represent a known SSL/TLS version number

Using this information, they were able to replace the previous lower bound of $2B$ with a new lower bound E, which takes into consideration the minimum value of the non-zero padding. Similarly, they replaced the previous upper bound of $3B-1$ with a new upper bound F, which takes into consideration the fixed position of the `0x00` delimiter byte. This slightly reduces the number of possible options for m prior to any queries to the oracle.

The second optimisation is based on their observation that it is possible to find the next valid s_i using a linear combination of two previous valid values of s_i. More specifically, if s_a and s_b are both valid multipliers, then we can try and search for the next valid s_i by setting $s_i = \beta \cdot s_a - (\beta - 1) \cdot s_b$ for some integer value β. Although this is interesting, for small values of β the reduction in the size of M_i is slow, and for large values of β the reduction is faster, but suitable values of β take a long time to find. As a result, they suggest that although this method could be used, it offers very little when compared to the Parallel-Threads Method discussed previously.

4.1.3 The Impact on our Example

As a means of demonstrating the effectiveness of the Parallel-Threads Method, let us consider the example from Section 3.3. Recall that after finding $s_1 = 82005$, we produced a new bound for m_0, but it contained more than one interval:

$$M_1 = \{[5798403242\ldots, 5798436691\ldots], \\ [7731182022\ldots, 7731215472\ldots]\}$$

Instead of entering step 2b from the original algorithm, we now enter step 2c twice in parallel; once with the first interval and once with the second interval. Just as we demonstrated in the example of step 2c, we must first establish the respective values of r_2 for each of the two intervals in M_1. We can then establish a suitable bound for potential values of s_2 for each of the intervals. Once we have this information, we can let the intervals take it in turns to query the oracle, alternating between their potential values of s_2. Also, as before, if we have tried all possible values within the bound then we set the respective $r_2 = r_2 + 1$ to create a new bound and continue. In this situation, it turns out that the first interval finds a value for s_2 first, so we stop looking with the parameters for the second interval, take the value

of s_2 and proceed to step 3. As expected, using this method we find $s_2 = 355351$, and since we only entered step 2c from this point on previously, the same is true here.

However, using this modified version of the algorithm, the attack now requires 69136 queries to the oracle to decrypt the ciphertext. This figure is impressive, but in the original example, 273,346 queries were made to the oracle in step 2b to find s_2. In this example, just 37 queries were made to the oracle to find s_2. This is an extremely impressive optimisation and we will analyse this more in Chapter 7.

4.2 Improvements from 2012

Between 2003 and 2012, there did not appear to be much of a focus on Bleichenbacher-style attacks. In some respects this may have been due to good implementation advice which prevented side-channel leakage, but the more likely reason is that focus was elsewhere. Then, in 2012, Bardou et al. published a detailed paper looking at Bleichenbacher-style attacks in a completely new setting [8]. Their focus was cryptographic hardware, where imported symmetric keys were often encrypted under an RSA public key using PKCS #1 v1.5 padding. They had discovered that the oracles discussed in Section 2.2 existed in real hardware, including Gemalto Cyberflex Smartcards, RSA SecurID devices and Estonian ID cards (which can be used for authenticating an SSL/TLS session and producing legally binding digital signatures). As a result, Bleichenbacher-style attacks against hardware were a reality, but they were not efficient enough to be practical against lower powered devices. This is because on devices such as a smart card, a single RSA decryption (which is required for each oracle query) is slow when compared to the speed offered by a server [8]. Based on this, Bardou et al. set about improving the efficiency of the algorithm. Their 2 areas of focus included trimming the initial interval M_0 and improving the efficiency of step 2a. In this subsection, we will detail these improvements and demonstrate how they affect our example.

4.2.1 Trimming M_0

Throughout this project, we have focused heavily on the fact that an attacker can multiply a plaintext message m_0 by some integer s, without having access to the private decryption key or the plaintext itself. However, in Section 2.2 of [8], they make the observation that it is also possible to divide the plaintext message m_0 by t, simply by multiplying it by $t^{-1} \bmod n$. This works exactly the same as we saw in Section 2.1.2, but instead we multiply the ciphertext by t^{-e}:

$$\begin{aligned} c \cdot t^{-e} \bmod n &\equiv m_0^e \cdot (t^{-1})^e \bmod n \\ &\equiv (m_0 \cdot t^{-1})^e \bmod n \end{aligned}$$

If the decimal value of the plaintext message m_0 is divisible by t, then $m_0 \cdot t^{-1} \bmod n = m_0/t$ (without the mod n). If m_0 is not divisible by t then the result will be some unknown value. However, this is very interesting as it allows the attacker to manipulate m_0 in such a way that they can reduce the interval which contains m_0 before the attack really begins. To demonstrate this, we will first need to present Proposition 1 from [8], along with its proof for completeness.

Proposition 1. *Suppose u and t are coprime positive integers such that $u < 3t/2$ and $t < 2n/9B$. If m_0 and $m_0 \cdot ut^{-1} \bmod n$ are PKCS conforming then t divides m_0.*

Proof.

$$m_0 \cdot u < m_0 \cdot \frac{3t}{2} < 3B \cdot \frac{3t}{2} < 3B \cdot \frac{3}{2} \cdot \frac{2n}{9B} = n$$

Hence it holds that $m_0 \cdot u \bmod n = m_0 \cdot u$ because it is less than n.
Now let $x = m_0 \cdot ut^{-1} \bmod n$. Since x is PKCS conforming, we have $x < 3B$. So

$$xt < 3Bt < 3B \cdot \frac{2n}{9B} = \frac{6n}{9} < n$$

Hence it also holds that $xt \bmod n = xt$ because it is less than n.
Therefore, we have $x \cdot t \bmod n = x \cdot t = m_0 \cdot u \bmod n = m_0 \cdot u$. As a result, t must divide m_0 because u and t are coprime by definition. □

This proposition means that if we can find such coprime integers u and t where, for a PKCS conforming message m_0, $m_0 \cdot ut^{-1} \bmod n$ is also PKCS conforming, then we know t divides m_0 and that $m_0 \cdot ut^{-1} \bmod n = m_0 \frac{u}{t}$, without the mod n. Since $m_0 \cdot ut^{-1} \bmod n = m_0 \frac{u}{t}$ is PKCS conforming, it holds that:

$$2B \leq m_0 \frac{u}{t} < 3B$$

Hence we have:

$$2B \cdot \frac{t}{u} \leq m_0 < 3B \cdot \frac{t}{u} \tag{4.1}$$

So if we are able to find a trimmer $\frac{u}{t}$ where t divides m_0 and $2B \leq m_0 \frac{u}{t} < 3B$, then we can trim the initial range containing m_0 before we begin the attack. Intuitively, one can see that we want to find a trimmer where $u < t$ to bring the lower bound up, and a trimmer where $u > t$ to bring the upper bound down.

For example, suppose that m_0 is a PKCS conforming plaintext, and further suppose that both $m_0 \cdot \frac{4}{5}$ and $m_0 \cdot \frac{8}{7}$ are also PKCS conforming. Then, using equation (4.1), it follows that:

$$2B \cdot \frac{5}{4} \leq m_0 < 3B \cdot \frac{7}{8}$$
$$2.5B \leq m_0 < 2.625B$$

Using these 2 trimmers alone, we have been able to reduce the bound which contains m_0 by 87.5% – bringing down the number of possible options for m_0 from 2^{1008} to 2^{1005}.

Clearly, the smaller the value of t, the more likely t is to divide m_0. In fact, the probability that t divides m_0 is $\frac{1}{t}$. Furthermore, the closer u and t are, the more likely the result of $m_0 \cdot \frac{u}{t}$ is to lie between $2B$ and $3B$.

Aside from the bounds given in Proposition 1, the search for trimmers should be restricted to $\frac{2}{3} < \frac{u}{t} < \frac{3}{2}$, since it is not possible for trimmers to exist outside of this range. A final note that Bardou et al. make on the subject of trimmers is

that although a smaller value of t is more likely to generate a successful trimmer, a larger value of t allows for more efficient trimming. Based on this, they optimised the trimming process by taking the set of all successful trimmer denominators t, and then computed the lowest common multiple of this set, denoted t'. Using this much larger denominator t' (which by construction divides m_0), they then proposed that you search for the highest and lowest numerators u_h, u_l which imply a valid padding. As a result, we have:

$$2B \cdot \frac{t'}{u_l} \leq m_0 < 3B \cdot \frac{t'}{u_h} \tag{4.2}$$

Unfortunately they do not detail the most effective way to search for these trimmers, but this is something that we will be investigating in Chapter 8. They do, however, provide an indication as to how many oracle queries one should expend searching for such trimmers. It seems that their suggested limits are a result of experimental observation, but they recommend 500, 600, 2000 and 1500 queries for a TTT, TFT, FTT and FFT oracle respectively. Nonetheless, trimming the initial interval M_0 naturally improves the efficiency of Bleichenbacher's algorithm, and after we have presented their improvements to step 2a, we will look at how trimming M_0 affects our running example from Section 3.3 and Section 4.1.3.

4.2.2 Improving Step 2a

In Section 2.2 of [8], the authors also presented a method to speeds up step 2a by increasing the starting value of s_1 and skipping over values of s_1 which cannot produce a PKCS conforming ciphertext $c_0(s_1)^e \bmod n$. This optimisation consists of two parts and we will present these now.

Their first observation was that it is possible to increase the lower bound for the starting value of s_1. Recall from Section 3.2 that in the original algorithm, we begin with the first integer $s_1 \geq n/(3B)$. However, we observed that when we find s_1, it must hold that $m_0 \cdot s_1 > n$. Thus it follows that if $m_0 \cdot s_1$ is a PKCS conforming plaintext message, then $m_0 \cdot s_1 \geq n + 2B$. Furthermore, since m_0 is PKCS conforming, we have $m_0 \leq 3B - 1$. As a result the following holds:

$$\begin{aligned} m_0 \cdot s_1 &\geq n + 2B \\ (3B - 1) \cdot s_1 &\geq n + 2B \\ s_1 &\geq \frac{n + 2B}{3B - 1} \end{aligned}$$

They explained that on its own this does not make much of a difference to the number of queries required to find s_1, but if we have already applied the trimming to M_0 discussed in Section 4.2.1, then the upper bound of M_0 is now b (which is less than $3B - 1$). As a result, since the largest value that m_0 can take is b, we have an improved starting bound for s_1 of the form:

$$s_1 \geq \frac{n + 2B}{b} \tag{4.3}$$

Their second improvement to step 2a stems from the fact that although we do not yet know the value of s_1, we do know that if $m_0 \cdot s_1 \bmod n$ is PKCS conforming,

then it holds that $2B \leq m_0 \cdot s_1 \bmod n < 3B$. Hence, for some positive integer j, $2B \leq m_0 \cdot s_1 - jn < 3B$. Rearranging this equation, it holds that

$$\frac{2B + jn}{m_0} \leq s_1 < \frac{3B + jn}{m_0}$$

Since we have trimmed the range M_0 containing m_0 to $[a, b]$, it also holds that:

$$\frac{2B + jn}{b} \leq s_1 < \frac{3B + jn}{a}$$

As a result, if there is a gap between this interval for j and $j + 1$, then any value of s_1 in this gap cannot be valid. More specifically, if

$$\frac{3B + jn}{a} < \frac{2B + (j + 1)n}{b}$$

then there must be a "hole" of values where a suitable value for s_1 cannot be found.

As a result, they deduced that if for some positive integer j we have:

$$\frac{3B + jn}{a} \leq s_1 < \frac{2B + (j + 1)n}{b} \tag{4.4}$$

then do not query the oracle with s_1 since it cannot be valid.

Bardou et al. noted that when this "Skipping Holes" technique is used in conjunction with the trimming technique, they often found several holes, and as a result, this optimisation leads to a significant improvement in the efficiency of the search for s_1. In fact, trimming M_0 both increases the lowest possible value that s_1 can take and increases the size of the holes that will not contain a valid s_1. We know this because from equation (4.3), it is clear that a reduced upper bound b of M_0 increases the size of $\frac{n+2B}{b}$. Furthermore, if we have increased the lower bound a of M_0 and reduced the upper bound b of M_0, then for a given value j in equation (4.4), this will reduce the size of $\frac{3B+jn}{a}$ and increase the size of $\frac{2B+(j+1)n}{b}$. Hence it follows that the size of the hole between these two values will increase, and as such, the distance between holes will be reduced.

The effects are much more noticeable when the oracle is strong since it is easier to find trimmers. However, even with a stricter oracle, the search for s_1 is still much more efficient. In Chapter 7, we will demonstrate how these improvements presented in 2012 and the improvements presented in 2003 (Section 4.1.2) statistically measure up against Bleichenbacher's original algorithm from 1998. However, in the meantime, we will return to our running example from Section 4.1.3 to see how trimming M_0 and skipping holes impacts the number of oracle queries.

4.2.3 The Impact on our Example

Our first task is to try and trim M_0 by searching for suitable trimmers $\frac{u}{t}$ such that m_0 and $m_0 \cdot ut^{-1} \bmod n$ are both PKCS conforming. Since m_0 is already PKCS conforming from the blinding step, it remains to find values of u and t where $m_0 \cdot ut^{-1} \bmod n$ is PKCS conforming.

Clearly we cannot continue searching for trimmers indefinitely since testing a trimmer costs a query to the oracle. However, as mentioned previously, Section 2.5 of [8] recommends spending around 1500 oracle queries searching for such trimmers for an FFT oracle. Please see Chapter 8 for our discussion on the most effective method to find trimmers, but for the purposes of this example, the only valid trimmer that we could find for these parameters within a limit of 1500 oracle queries was $\frac{17}{14}$.

Since we only have one trimmer ($\frac{17}{14}$), there is no need to compute the lowest common multiple of the denominator since it will be the same value – hence $t' = 14$. We now need to search for the highest and lowest numerators (u_h, u_l) that imply a valid padding. It turns out that $u_h = 17$ and $u_l = 14$, so by using equation (4.2), it follows that:

$$2B \cdot \frac{14}{14} \leq m_0 < 3B \cdot \frac{14}{17}$$

Hence we have been able to trim the interval containing m_0. As before, the actual numbers are too big to present here, but they can be found in Appendix C. It now follows that:

$$M_0 = \{\texttt{[5486124068..., 6776976790...]}\}$$

This process of trimming has more than halved the size of the interval containing m_0, and the number of possible options has been reduced from 2^{1008} to just under 2^{1007}.

We can now begin our search for s_1, and we start our search with $s_1 \geq \frac{n+2B}{b}$, where b is the new upper bound of the interval M_0. As a result, the first possible value for s_1 is 23389. Using this value and increasing by one after each failed oracle query, we continue to searching until we find a valid s_1. However, this time we can skip any "holes" that we know will not contain a valid s_1.

For example, by setting $j = 1$ in equation (4.4), we have:

$$\frac{3B + n}{a} = 28892.12$$
$$\frac{2B + 2n}{b} = 46776.01$$

Hence it follows that s_1 cannot be between 28893 and 46776. Similarly, by setting $j = 2$, we find that s_1 can also not be between 57783 and 70163.

As we would expect, it turns out that $s_1 = 82005$, but one can see that we have been able to find this value using significantly less calls to the oracle. In fact, in our previous examples, we required 62745 calls to the oracle to find s_1. However, here we have been able to find s_1 with just 28352 oracle queries (a reduction of nearly 55%). The rest of the algorithm would then continue as described previously.

It should be clear that the most efficient version of the algorithm would be to combine the trimming of M_0, the optimisation of step 2a, and the optimisation of step 2b. As a result, by implementing all optimisations from Section 4.1 and 4.2,

we are now able to decrypt the ciphertext with just 34719 calls to the oracle. This is 10 times more efficient than our original algorithm implementation in Section 3.3; which took 342445 queries to the oracle to decrypt the ciphertext.

Chapter 5

Recent Bleichenbacher-style Attacks in Practice

Although we have demonstrated the practical implications of Bleichenbacher-style attacks on SSL/TLS, the oracles that we have observed so far should not exist within more modern implementations. However, that is not to say that padding oracles do not exist at all. In this section, we will analyse how Bleichenbacher-style attacks have exploited side-channel leakage over the last 4 years to perform private key operations without knowledge of the private key.

5.1 New Side-Channel Attacks against SSL/TLS

In 2014, Meyer et al. published a paper [12] identifying four new side-channel attacks on SSL/TLS, one of which utilised error messages and three of which were timing-based. All of these attacks took advantage of a PKCS #1 v1.5 padding oracle, and this piece of literature was the first to demonstrate Bleichenbacher-style attacks against SSL/TLS using time-based information leakage. After practical testing of these four new side-channel attacks, in three of them they were able to correctly recover the `PMS`. We will describe all four of these new side-channel attacks in this subsection, but first we will introduce the fundamentals for timing-based attacks on the SSL/TLS protocol.

In Section 4.1.1, we briefly mentioned that if the `ClientKeyExchange` message does not decrypt to a conforming plaintext, then the server should generate a random `PMS` and continue with the protocol. As a result, this means that no error message is sent at all and the protocol will break down at a later stage. This thwarts the Bleichenbacher-style attacks discussed so far, and as such, this was the countermeasure which was advised in Section 7.4.7.1 of both TLS 1.0 [13] and TLS 1.1 [14]. However, clearly it will take a small amount of time to compute a random `PMS`, so there may well be some information leakage here. With this in mind, TLS 1.2 changed the recommendation, instead advising that a random `PMS` should always be generated, regardless of whether the `ClientKeyExchange` message is PKCS conforming (Section 7.4.7.1 of [10]). If this is implemented correctly then processing times for a correctly and incorrectly padded `ClientKeyExchange` message should be identical. Nonetheless, even with such advice available, it was these differences in processing times that Meyer et al. exploited in their time-based side-channel

attacks.

5.1.1 Error Messages in Java Secure Socket Extension

Java Secure Socket Extension (JSSE) is the build-in SSL/TLS implementation for Java, and this side-channel attack was a result of an implementation bug within JSSE.

Meyer et al. explained that an improper padding check and the processing of the subsequent `PMS` meant that they could force the server to respond with different error messages (Section 5 of [12]). More specifically, we know that if the `PMS` is of a fixed length, then there is a pre-determined padding string which should be non-zero. If a `0x00` byte is found in the non-zero padding string, then the padding check should fail and the server should respond with a `HANDSHAKE_FAILURE` alert [12]. Although this was true, it was not always true. For a 2048 bit and 4096 bit RSA modulus, Meyer et al. noticed that if the plaintext began with `0x0002` and a subsection of the non-zero padding contained a `0x00` byte preceded by non-`0x00` bytes, then instead the server would respond with an `INTERNAL_ERROR` alert. Such an error would leak the information required to conduct Bleichenbacher's attack. With a 2048 bit RSA modulus, this subsection makes up 57% of the padding string, and for a 4096 bit RSA modulus, this subsection makes up 81% of the padding string. Furthermore, they also discovered that if the penultimate byte of the `PMS` was `0x00` and all preceding bytes of the `PMS` were non-`0x00`, then the server would also respond with an `INTERNAL_ERROR` alert. This second discovery was true for a 1024 bit, 2048 bit and 4096 bit RSA modulus. Please see Figure 7 in [12] for more information.

Following their findings, they conducted some analysis and explained that assuming the decrypted message begins with `0x0002`, for a 2048 bit RSA modulus, the chances of receiving an `INTERNAL_ERROR` is 35.6%, and for a 4096 bit RSA modulus, the chances are 74.4% [12]. One can see that this results in a very strong oracle where the Bleichenbacher attack can be efficiently executed. In fact, with a 2048 bit RSA modulus, the median number of oracle queries was 37399, and with a 4096 bit RSA modulus, the median number of oracle queries was 27744. Although it is possible for an attacker to receive an `INTERNAL_ERROR` when working with a 1024 bit RSA modulus, on average this will only happen 0.24% of the time [12] – thus providing a much weaker oracle. This bug has now been fixed.

5.1.2 Timing Differences in OpenSSL

Their second side-channel attack was discovered after noticing that OpenSSL did not implement the TLS 1.2 countermeasures to prevent Bleichenbacher-style attacks exploiting the processing time to generate a random `PMS` (as discussed in Section 5.1). Instead, the random `PMS` was generated if and only if the ciphertext was not strictly PKCS conforming. As a result, this time leakage allowed an attacker to distinguish between a conforming and non-conforming ciphertext. The timing difference between a valid and invalid ciphertext was extremely small (around 1.5 microseconds), and they did also note that they were unable to confirm that this time difference was entirely a result of the random number generation. This is due

to the presence of other branches and loops in the OpenSSL source code [12].

Nonetheless, a side-channel existed, but it turned out to only be theoretical. This was because although they could distinguish between a valid and invalid ciphertext, the oracle that this allowed for was extremely strict. OpenSSL may have failed to implement a random `PMS` generator before validating the padding structure, but the padding check was sufficiently strict. Assuming a 2048 bit RSA modulus, the implementation checked for the first two bytes being `0x0002`, a 205 byte non-zero padding string, a `0x00` delimiter byte, a 48 byte `PMS`, and correct major and minor version number bytes following the delimiter byte. As a result, even if the ciphertext begins with `0x0002`, the chances of the ciphertext complying with the padding check is roughly 1 in 37 million [12]. Hence they were clearly not able to provide any practical results, but they did predict that such an attack would require around 5 trillion oracle queries. Despite the fact that this timing leakage could not be exploited, the theoretical attack made the OpenSSL implementation cryptographically insecure. Therefore, this provides a good argument for following the most recent standards at the time (TLS 1.2), and generating the random `PMS` before validating the padding.

5.1.3 Java Secure Socket Extension Internal Exception

This is another side-channel attack that Meyer et al. discovered in JSSE, but unlike the first, this one utilises timing differences instead of the direct error messages discussed in Section 5.1.1. They were able to confirm that the implementation strictly checked for the message beginning with `0x0002`, contained at least eight non-zero padding bytes, and contained a `0x00` delimiter byte. However, it did not check for a pre-defined length of the `PMS`; so this implementation is very similar to the FFT oracle that we discussed in Section 2.2.3. The slight difference was that although an internal exception was thrown if the padding was invalid, the attacker would not receive a distinguishable error message. As a result, it thwarts Bleichenbacher's attack if we are using error messages alone. On the other hand, exception handling in Java can introduce timing delays [12], and Meyer et al. were able to measure this delay. The result of their measurements showed a time delay of around 20 microseconds, and so they were able to construct a timing oracle.

A model of such an oracle would process a valid padding quickly and return a 1, and if the padding is invalid, then the internal exception would slow down the processing speed – thus the model returns a 0. By being able to distinguish between a valid and invalid padding in this way provides an oracle of almost equal strength to the FFT oracle discussed previously. By using the optimised Bleichenbacher algorithm discussed in Chapter 4, with a 2048 bit RSA modulus, they were able to conduct the attack on Java 1.7 in 55 hours, querying the oracle 20662 times [12]. This number is slightly higher than they expected, but Meyer et al. attributed this to the presence of false negatives. In such a scenario, false negatives can be easily introduced through networks delays. For example, an unexpected network delay may cause a ciphertext with a valid padding structure to be deemed invalid based on the processing time. In order to counter this time leak, they did suggest introducing a time constant, which essentially looks to reduce the difference in processing time to zero. Another countermeasure that they did not mention would be to reduce the

probability of a random ciphertext passing the padding validity check. Therefore, like OpenSSL, JSSE could require the `PMS` to be strictly 48 bytes in length, and contain valid SSL/TLS version numbers. This would ensure that the strength of the oracle would be so low that the attack would become infeasible – even with a 20 microsecond processing time difference between a valid and invalid padding.

5.1.4 Unexpected Timing Behaviour in Hardware Appliances

The final side-channel attack that Meyer et al. were able to establish existed in F5 BIG-IP and IBM Datapower, both of which used a Cavium NITROX SSL accelerator chip [12]. The oracle that they observed initially seemed strong, accepting any padding structure that began with `0x0002`. However, the most interesting point about this side-channel attack was that the oracle also accepted any padding structure that began with `0x??02`. So here, `0x??` represented any one of the 256 possible options for a byte. As a result, the oracle became much weaker.

Again, it should be noted that valid and invalid ciphertexts were not being distinguished through error messages. Just like in the previous two findings, Meyer et al. noticed a distinguishable difference in the processing times of valid and invalid padding structures. More specifically, their measurements showed that if the ciphertext was valid (began with `0x??02`), then the processing time was around 10 to 15 microseconds longer than if the ciphertext was invalid. As such, they could construct an oracle which would return 1 if the ciphertext was considered to be valid.

Since, in such a scenario, it was not guaranteed that valid messages would begin with `0x0002`, they had to modify Bleichenbacher's algorithm to cater for the fact that the first byte could be any one of the 256 possible options. This modification can be found in Section 9 of [12], but the main difference is that instead of knowing that a valid plaintext is between 2B and 3B, we now know that it is between xB and yB, where $x = 256a + 2$, $y = 256a + 3$, and $a = 0, 1, 2, ..., 255$. They then used these additional intervals to optimise the search for values of s_i, which in turn allowed them to reduce the interval containing the message m_0.

After 500 simulated test runs, they established a median of 4700 queries to decrypt a given ciphertext using this method. In general, this low number of queries is based on the fact that it is relatively easy to find a value of s_i that ensures $m_0 \cdot s_i \bmod n$ lies within any one of the 256 intervals discussed above. Furthermore, they also performed a real attack – which required 7371 oracle queries. As mentioned previously, the difference between a simulated and a real attack is the existence of false negatives. Network delays may give the impression that a valid ciphertext is invalid or an invalid ciphertext is valid.

All things considered, the four new side-channel attacks that we have discussed in this section from [12] show that a great deal of care should be taken when implementing SSL/TLS. It is important that the padding structure is strict, but it is also important to ensure information is not leaked regarding the validity of the strict padding check; be it via error messages, timing delays, or some other means.

5.2 DROWN

DROWN is an acronym for Decrypting RSA with Obsolete and Weakened eNcryption, and it is a Bleichenbacher-style cross-protocol attack which exploits weaknesses in SSLv2 to decrypt a TLS connection. It was published in August 2016 by Aviram et al. [2], but before we can explain how DROWN works, we must first introduce some additional preliminaries.

5.2.1 SSLv2

Recall from Section 2.3 that after the client has sent the `ClientKeyExchange` message, this is followed up by a `ClientFinished` message. In all the Bleichenbacher-style attacks discussed so far, the `MAC` contained within this `ClientFinished` message will be incorrect. This is because the attacker does not know the `PMS` at this stage, and therefore they cannot derive the `MAC` key. However, this makes no difference to an attacker because they have already obtained the information they want by observing the response of the server to the `ClientKeyExchange`. Nonetheless, the structure and message content that we discussed in Section 2.3 is true for all versions of SSL/TLS, with the exception of SSLv2. There are a few small differences in SSLv2. For clarity, the first two differences we will mention are that the `PMS` is now referred to as a master key (`MK`), and the `ClientKeyExchange` is now referred to as the `ClientMasterKey`. The third difference we will mention is extremely important for the DROWN attack and we will explain it here.

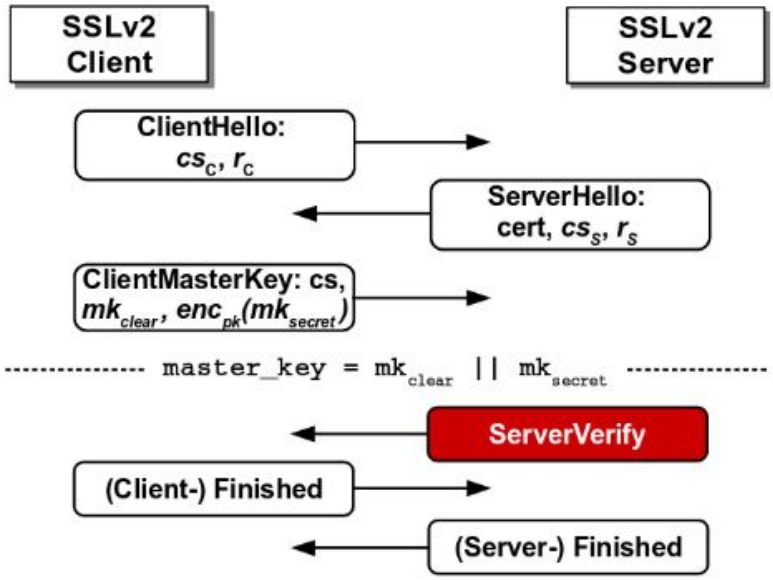

Figure 5.1: The SSLv2 protocol (taken from [2]).

As can be seen in Figure 5.1, we have an additional message which is sent from the server to the client known as the `ServerVerify` message. According to [15], the purpose of this message is to authenticate the server to the client. To construct the `ServerVerify` message, the random nonce (r_c) sent by the client is encrypted using the `server_write_key` (deduced from the `MK`). Thus, in theory, only the server with

knowledge of the RSA private decryption key should be able to produce such a message. Additionally, the implementation advice in Section 5.2.1 of [15] recommends the `ServerVerify` message to be sent after the `ClientFinished` message. However, Aviram et al. noticed that this was not the case, and so the protocol shown in Figure 5.1 represent the reality of an SSLv2 implementation. With this in mind, once an attacker has received the `ServerVerify` message, they know the ciphertext, they know the plaintext (r_c), but they do not know the key (`server_write_key`).

Although SSLv2 was quickly proven to be insecure, it may be enabled based on the fact that it seems sensible to assume that any encryption is better than no encryption. This is perhaps even more relevant for legacy systems which have not been updated, and so SSLv2 is better than plaintext communication – perhaps sufficing as a last point of call if SSLv3 and above cannot be negotiated. Furthermore, if a server supports SSLv2, as well as more modern versions, then it is fair to assume that they will share the same RSA key and certificate. This may be considered poor key management, but Section 9.2 of [2] explains that there is no agreed way to ensure that an X.509 certificate is version specific. Web servers (such as Apache) also lack configuration options to enable good key separation, and from a commercial standpoint, there is a much greater cost involved in obtaining individual RSA keys and certificates for each SSL/TLS version. Finally, although it is an old implementation, Aviram et al. observed that an SSLv2 server will still check the validity of the PKCS #1 v1.5 padding, and if it is not valid, then a randomly generated `MK` will be used instead (as discussed in Section 5.1 and advised in [13] and [14]).

Hopefully, from the above, one can see that if an attacker were to send the same invalid `ClientMasterKey` to the SSLv2 server twice, then each `ServerVerify` response would be encrypted under a different `server_write_key` from a different randomly generated `MK`. However, if an attacker were to send the same valid `ClientMasterKey` to the SSLv2 server twice, then each `ServerVerify` response would be encrypted under the same `server_write_key`. This is because the `MK` would be extracted from the valid `ClientMasterKey` message on both occasions. We will return to this important result in Section 5.2.3.

5.2.2 40 Bit Export Encryption

We mentioned in Section 5.2.1 that, upon receipt of the `ServerVerify` message, the attacker knows the ciphertext and the plaintext, but they do not know the key. If the length of the key is 128 bit, then a brute force attack is infeasible. However, if the length of the key is 40 bit, then an attacker can conduct an exhaustive key search to deduce the key. Note that when we say a 40 bit key length, what we really mean is a 128 bit key, of which 40 bits are secret. As a result of the Crypto Wars, SSLv2 allows the use of a 40 bit export cipher key. Since the client can choose the ciphersuite, negotiating this weakened key length is trivial.

In Figure 5.1, we can see that the `ClientMasterKey` consists of a ciphersuite selection and `MK` itself. Suppose the ciphersuite selected by the attacker is a 40 bit export cipher, then `MK` is a concatenation of $\texttt{MK}_{clear}$ and $\texttt{MK}_{secret}$ [2]. If the length of `MK` is 128 bits, then the length of $\texttt{MK}_{clear}$ is 88 bits and the length of $\texttt{MK}_{secret}$ is

40 bits. As expected, $\texttt{MK}_{clear}$ is sent in the clear so is not encrypted and we can assume is public knowledge. However, $\texttt{MK}_{secret}$ is padded according to the PKCS #1 v1.5 padding scheme, then encrypted with the public RSA key of the server. This means that the encoding will begin with `0x0002`, the 6^{th} byte from the end will be the `0x00` delimiter, and the message will contain just 5 bytes. The padding string in between should be non-zero. A ciphertext that decrypts to a plaintext with such an encoding will now be referred to as an SSLv2 conforming ciphertext. Again, we will return to this in Section 5.2.3

5.2.3 Implementing DROWN

As mentioned previously, perhaps it was fair to assume that SSLv2 is better than no encryption at all. However, this attack demonstrates that SSLv2 can devastatingly weaken the security of all later SSL/TLS version implementations. Although this attack does not guarantee that we can decrypt a specific TLS session, it does allow an attacker to decrypt one of many TLS sessions. As such, the attacker could monitor a victim over a short period of time to collect a large number of such sessions before conducting the attack. Furthermore, the server that is running TLS may be completely invulnerable to a Bleichenbacher-style attack, but if it shares an RSA key with another server that supports SSLv2, then the DROWN attack will succeed by using the SSLv2 implementation as an oracle. As a result, the attacker will learn the session key for a specific TLS connection. Perhaps even more surprising is that to conduct such an attack, the attacker does not ever need to communicate with the TLS server, and the victim does not ever need to communicate with the SSLv2 server. In addition, a lot of the work can actually be conducted offline, and such an attack can take place and be successful many weeks or months after the initial TLS session was established.

We have already discussed some of the foundations of the attack, but it is still quite a complicated attack path. Therefore, we will break the attack down into smaller steps.

Step 1: Observe and collect around 1000 TLS handshakes
Suppose that the victim is communicating with a server over a secure TLS implementation that follows all advice to counter Bleichenbacher-style attacks. Further suppose that this server also shares an RSA key with an SSLv2 implementation. The first thing the attacker should do is collect around 1000 TLS connections. Aviram et al. explain that we can expect to be able to decrypt roughly 1 in every 1000 connections using DROWN (Section 3 of [2]).

Step 2: Convert the TLS ciphertext into an SSLv2 conforming ciphertext
The purpose of this step is to find a means of manipulating a TLS ciphertext such that the result is an SSLv2 conforming ciphertext. So, assuming the RSA modulus is 2048 bit, as it stands, the decrypted TLS plaintext m will begin with `0x0002`, then contain a 205 byte random padding string, then a `0x00` delimiter, and finally a 48 byte `PMS` (potentially beginning with the appropriate major and minor version numbers). We need to convert this into a ciphertext such that when it is decrypted, the padding will begin with `0x0002` and the 6^{th} byte from the end is `0x00`. To do

this, they used the idea of trimmers, as introduced in Section 4.2.1.

So, for each of the intercepted TLS ciphertexts c, an attacker should choose a selection of the best trimmers $s \equiv ut^{-1} \bmod n$ (see Chapter 8), and then construct modified versions of the TLS ciphertexts, c_1, such that $c_1 \equiv c \cdot s^e \bmod n$. We now need to discover whether or not any of these newly constructed TLS ciphertexts are SSLv2 conforming. Note that if c_1 is SSLv2 conforming, then we also know that $m_1 \equiv c_1^d \bmod n = m \cdot \frac{u}{t}$.

Step 3: Checking for SSLv2 conformance
So we have a selection of modified TLS ciphertexts but we need to find out if any of them are SSLv2 conforming. To do this, the attacker should begin an SSLv2 session with the server which shares the RSA encryption key. They should ask for a 40 bit export cipher, and as a result, the SSLv2 server will expect 88 bits of the symmetric key to be sent in the clear and 40 bits of the symmetric key to be encoded and encrypted using the RSA public key.

The attacker can choose the 88 bits of $\texttt{MK}_{clear}$ to be sent over, and this is followed by any one of the modified TLS ciphertexts. Suppose that the modified TLS ciphertext is not SSLv2 conforming. In this scenario, the SSLv2 server will decrypt $\texttt{MK}_{secret}$, discover that the padding is invalid, and then generate a random 40 bit $\texttt{MK}_{secret}$. This will be concatenated with $\texttt{MK}_{clear}$ to form `MK`, and the server will then calculate the `ServerVerify` message by encrypting r_c under the 128 bit key. The result will be sent to the attacker.

Once the attacker has received the `ServerVerify` message, they know it is the encryption of r_c, so they know the plaintext, the ciphertext, and the first 88 bits of the 128 bit key. As a result, they can brute force the remaining 40 bits to establish `MK` (or more specifically, the `server_write_key`). Now the attacker should begin a new session with the same set-up and send the same modified TLS ciphertext. Once again, the server will reply with a `ServerVerify` message. This time the attacker can test to see if the encryption key used to produce this message is the same as the key they have just discovered. However, since the modified TLS ciphertext was not SSLv2 conforming, the server will have generated a new random 40 bit $\texttt{MK}_{secret}$, and so the keys used to encrypt will not be the same. Based on this, the attacker knows that the modified TLS ciphertext was not SSLv2 conforming, and they move onto the next one.

Now suppose that the modified TLS ciphertext is SSLv2 conforming. As before, the attacker sends it to the SSLv2 server and then, when they receive the `ServerVerify` message, they can use an exhaustive key search to establish $\texttt{MK}_{secret}$ and construct `MK`. However, when they send the same modified TLS ciphertext for the second time, since it is SSLv2 conforming, on both occasions the server will have identified the valid padding and extracted the 5 least significant bytes. As a result, the same key will be used to produce the `ServerVerify` message on the second run of the protocol, and the attacker can quickly check this. Thus, the attacker knows that the modified TLS ciphertext is indeed SSLv2 conforming. In addition to this, they now also know that the respective plaintext begins with `0x0002`, the 6^{th} byte from

the end is `0x00`, and the 40 bit key that they have just brute forced represents the last 5 bytes.

Since the attacker has found a modified TLS ciphertext which is SSLv2 conforming (denoted $c_1 = m_1^e \bmod n$), they know 8 bytes of the corresponding plaintext, and we have almost bootstrapped Bleichenbacher's original algorithm. Note that an attacker should be expected to obtain an SSLv2 conforming ciphertext after roughly 10,000 oracle queries (or 20,000 connections to the server) [2], so Aviram et al. recommend testing 10 trimmers per TLS ciphertext. Once we find a TLS ciphertext and trimmer pair that produce an SSLv2 conforming ciphertext, we can stop searching and this TLS ciphertext is the one that we will be able to decrypt using DROWN.

Step 4: Shifting known bytes
Although it may seem that we are now able to conduct Bleichenbacher's original algorithm, in order to do so, Aviram et al. noted that for a 2048 bit RSA modulus, the original algorithm expects $s_1 \approx 2^{24}$. However, in this situation, the only s value that we have (which could perhaps be denoted s_0) is a fraction between $\frac{2}{3}$ and $\frac{3}{2}$, so clearly this is not the case. Instead they introduced the idea of shifting bytes.

The shifting bytes technique allows them to shift the known plaintext bytes ($\texttt{MK}_{secret}$) from the least to the most significant bytes. For example, we know the first two bytes and last six bytes of m_1. So, by calculating $m_1 \cdot 2^{-48} \bmod n$, we can shift the known six least significant bytes to become the six most significant bytes. Their values must be calculated, but as a result, we have shifted the bytes such that we now know the eight most significant bytes. Similarly, by calculating $m_1 \cdot 2^{-40} \bmod n$, we can shift the five least significant bytes to become the most significant bytes, and this results in knowledge of the first seven bytes and last byte of the plaintext message. We do actually utilise a shift of 2^{-40} for reasons that will soon be explained.

The aim of this is to find a value of s_1 such that $m_2 \equiv m_1 \cdot 2^{-40} \cdot s_1 \bmod n$ is also SSLv2 conforming. Since we can calculate the first seven bytes of $m_1 \cdot 2^{-40} \bmod n$, it is easy to ensure that m_2 begins with `0x0002`. However, we cannot ensure that the padding string is non-zero or that the 6^{th} byte from the end is `0x00`. As such, we must query the SSLv2 oracle using increasing values of s_1 that ensure m_2 begins with `0x0002`, and check to see if the result is an SSLv2 conforming message. To do this, we must conduct the same technique as before, whereby the 40 bit secret is found through brute force, and then we query the SSLv2 oracle a second time to see if the same key is used. However, in this scenario we actually only need to brute force a 32 bit key since, due to our shift of 2^{-40}, the last byte can already be calculated. Once we have successfully found s_1 that implies an SSLv2 conforming $c_2 \equiv m_2^e \bmod n$, we definitely know the first two bytes of m_2, the last six bytes of m_2, and bytes three through to seven can be deduced.

Unfortunately, the value of s_1 is still not large enough to switch to the original algorithm, and as a result, we now need to find a value of s_2 such that $m_3 \equiv m_2 \cdot 2^{-40} \cdot s_2 \bmod n$ is SSLv2 conforming. Again, we can be sure that m_3 begins with `0x0002` since we can now calculate the first twelve bytes, but we still cannot ensure that the padding string is non-zero or that the 6^{th} byte from the end is `0x00`. So, as

before, we query the SSLv2 oracle and brute force the 32 unknown bits of the key with increasing values of $s_2 > s_1$ until we find an SSLv2 conforming $c_3 \equiv m_3^e \bmod n$. Once we have found $c_3 \equiv m_3^e \bmod n$, we then definitely know the first two bytes of m_3, the last six bytes of m_3, and bytes three through to twelve can be deduced.

Step 5: Bootstrapping Bleichenbacher's algorithm
That is the end of the byte shifting, but before we can just continue with the original algorithm, we must find s_3 such that $m_4 \equiv m_3 \cdot s_3 \bmod n$ is SSLv2 conforming. Since we know the two most significant bytes and six least significant bytes of m_3, before querying the SSLv2 oracle, we can be sure that the message begins with `0x0002` and that the 6^{th} byte from the end is `0x00`. Unfortunately, we cannot be sure that the padding string is non-zero, but the probability that it is non-zero for a 2048 bit modulus is 0.37; thus, finding such a value of s_3 is a trivial task. So we send $c_3 \cdot s_3^e \bmod n$ to the oracle to see if the result is SSLv2 conforming, and since we already know the six least significant bytes, we no longer need to brute force the secret key. Hence, from this point forward, checking for SSLv2 conformance is a simple task, and on average, we should find a successful value for s_i within three attempts.

We have now successfully bootstrapped the original algorithm and we can continue searching for increasing values of s_i until we have reduced the number of possible options for m_3 to one (i.e the interval containing the SSLv2 conforming message m_3 is of length 1). For clarity, in the DROWN attack, m_3 essentially plays the role of m_0 in the original algorithm. The only major difference is that to test for SSLv2 conformance, instead of using error messages, we query the SSLv2 server twice and then check to see if the same 40 bit $\texttt{MK}_{secret}$ is used both times. If it is, then the message is SSLv2 conforming, if not, then we move onto the next value of s_i.

Step 6: Recovering the TLS session key
After Bleichenbacher's algorithm has finished, we know $m_3 \equiv c_3^d \bmod n$ (an SSLv2 conforming plaintext), but we wish to find $m = c^d \bmod n$ (the plaintext for one of the TLS sessions that was initially observed). To do this, we must first reverse the steps we took to construct m_3 in Step 4 by reversing the byte shifting. As such, it follows that:

$$m_2 = m_3 \cdot 2^{40} \cdot s_2^{-1} \bmod n$$
$$m_1 = m_2 \cdot 2^{40} \cdot s_1^{-1} \bmod n$$

We have now recovered m_1, and this was the first SSLv2 conforming plaintext that we constructed. So, recall from step 2 that we found a successful trimmer $s \equiv ut^{-1} \bmod n$ such that $c_1 \equiv c \cdot s^e \bmod n$ was SSLv2 conforming. From Section 2.1.2, it therefore follows that $m_1 \equiv m \cdot s \bmod n$. We know s since we chose it and we know m_1 as we have just recovered it. Hence we have $m \equiv m_1 \cdot s^{-1} \bmod n$, or more literally:

$$m = m_1 \cdot \frac{t}{u}$$

As a result, we have recovered the plaintext from a TLS handshake, and we can now remove the known padding structure to recover the 48 byte `PMS` and derive the corresponding session key.

5.2.4 Final Notes on DROWN

The DROWN attack works against a server which implements both SSLv2 and later TLS implementations. However, as we have seen above, it is equally effective against a server that does not implement SSLv2, but does share an RSA key with a second server that does implement SSLv2. Another interesting note here is that SSLv2 does not just operate over HTTPS; SSLv2 can also be implemented for other protocols such as SMTP, POP3 and IMAP [2]. Therefore, if these other protocol implementations utilise SSLv2 and the same RSA key, then they can also be exploited as an oracle to decrypt a TLS session key. Similarly, even when other protocol implementations do not use SSLv2, if the RSA key is in any way associated to a protocol that does implement SSLv2, then the servers will be vulnerable to DROWN.

As a result, it should be clear that enabling SSLv2 on any protocol is detrimental to the security of TLS, and it should be disabled in all circumstances. In 2016, Aviram et al. conducted some research into the number of vulnerable servers, and they found that a third of all HTTPS servers were vulnerable to DROWN (around 11.5 million). The full list of results can be found in Table 4 of [2], but this worryingly large number stresses the need for carefully configured SSL/TLS implementations.

One final point that we will mention regarding the DROWN attack is that Aviram et al. also discovered two implementation errors in OpenSSL for SSLv2. The first of these (Section 4.4 of [2]) allowed an attacker to negotiate an export ciphersuite with a server, even if the server did not explicitly allow export ciphersuites. As a result, this enabled the DROWN attack against all OpenSSL implementations of SSLv2. The second of these two errors (denoted Special DROWN in Section 5 of [2]) actually meant that OpenSSL servers would allow the `ClientMasterKey` message to contain `clear_key_data` bytes for non-export ciphers. This means that if the server is expecting a 16 byte encryption key of the form `k[1]`, `k[2]`,..., `k[16]`, the attacker can send:

```
[00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  k[1]]
```

Hence it follows that the attacker now only has to brute force a maximum of 256 possible options for `k[1]` until they have successfully produced the same ciphertext as the `ServerVerify` message. Now they know `k[1]` they can query the server with:

```
[00  00  00  00  00  00  00  00  00  00  00  00  00  00  k[1]  k[2]]
```

The attacker does the same thing to calculate `k[2]`, and they can continue in this way to calculate the entire 16 byte encryption key – with an average of 128 trial encryptions to calculate each byte. This enables an extremely efficient oracle which drastically reduces the complexity of the original DROWN attack – since the majority of the computation is involved in brute forcing the 40 unknown bits of `MK` [2]. In Section 5.2.1 of [2], Aviram et al. noted that the computational costs were so low that they could complete the full attack on a single workstation in under one minute. As a result, it actually optimises the attack in such a way that the server becomes vulnerable to a potentially more devastating man-in-the-middle attack.

Both of these implementation errors have now been fixed, but the Special DROWN

vulnerability scanning performed by Aviram et al. many months after the fixes (Table 5 in [2]) shows that, at the time, 26% of HTTPS servers were still vulnerable to Special DROWN. Therefore, not only should SSL/TLS implementations be carefully configured, but implementations utilising libraries such as OpenSSL should maintain a well-refined patch management system.

5.3 ROBOT

ROBOT is an acronym for Return Of Bleichenbacher's Oracle Threat, and unfortunately it seems that 20 years has not proved long enough to thwart Bleichenbacher-style attacks against RSA in TLS. In December 2017, Böck et al. published the most recent piece of literature on the subject [3], and their research shows that almost one third of the top 100 domains are still vulnerable to variations of the attack – including Facebook and PayPal. Although this paper does not set out to prove anything fundamentally new, they do provide a more up to date overview of Bleichenbacher vulnerabilities in TLS implementations, of which there are many. In this section, we will analyse their findings and provide an example demonstrating how they were able to forge a digital signature using Facebook's private RSA signature key.

5.3.1 Scanning for Bleichenbacher Vulnerabilities

We have previously seen that a server can be vulnerable to a Bleichenbacher-style attack if there are differences in the error messages or the processing times between a correctly padded and an incorrectly padded message. We have also seen that what a server deems to be correctly padded varies across implementations. As such, their research centred around sending each target server a selection of messages, each of which was constructed to test for different vulnerabilities. The five messages that they constructed were as follows:

M1: A correctly padded message according to PKCS #1 v1.5, where the message begins with `0x0002`, contains a non-zero padding string followed by the `0x00` delimiter, then contains the `PMS` which is strictly 48 bytes in length (the first two of which contain the TLS version number).

M2: An incorrectly padded message which does not begin with `0x0002`.

M3: A correctly padded message according to PKCS #1 v1.5; except this time the `0x00` delimiter byte is in the wrong position. Therefore, the length of the `PMS` is invalid. This is identical to the FFT oracle discussed in Section 2.2.3.

M4: The padding starts with `0x0002`, but there is no `0x00` delimiter byte at all.

M5: A correctly padded message according to PKCS #1 v1.5, where the message begins with `0x0002`, contains a non-zero padding string followed by the `0x00` delimiter, then contains the `PMS` which is strictly 48 bytes in length. However, on this occasion, the bytes containing the TLS version numbers are incorrect (similar to the Bad Version Oracle discussed in Section 4.1.1).

Not only did they analyse error messages and processing times, but they also monitored connection state and any timeout issues. Additionally, in Section 4.2 of [3],

they mention that they observed differences based on the constructed TLS protocol flow. For example, recall from Section 2.3 that the third message in the protocol run involves sending the `ClientKeyExchange` message. This could be sent individually or alongside the `ChangeCipherSpec` and `Finished` messages, and they found that such variations did prompt servers to respond differently based on the validity or invalidity of the padding structure. Furthermore, they also explained that varying the requested symmetric mode of operation provided alternative responses. As such, one can see that overall their vulnerability scan was extensive.

With this in mind, for each permutation of the above, they sent all five messages. If the response for each message was not identical then they considered the server to be vulnerable to a Bleichenbacher-style attack; otherwise they tried a different permutation for that server. We will now briefly look at the most notable results of this research and the final statistics presented in their paper.

5.3.2 The Results of the Scan

Böck et al. were successfully able to identify a number of vulnerable SSL/TLS implementations from high-profile vendors. Full details can be found in Section 5 of [3], but we will describe the source of the oracles for each vulnerability here.

Facebook
Firstly, if there was an error in the PKCS #1 v1.5 padding, then Facebook would immediately reset the TCP connection. Such a situation informed them that the padding was invalid, and thus they were able to use this as an oracle to forge a digital signature. Facebook fixed this side-channel, but Böck et al. then identified a second oracle. They noticed that if the `ChangeCipherSpec` and `Finished` messages were not sent with the `ClientKeyExchange` message, then the server would wait for these messages only if the padding was valid. However, certain padding errors would cause the TCP connection to close. Facebook does actually use OpenSSL, but these bugs were a result of custom patches exclusive to Facebook. Therefore, identical oracles do not exist in more generic implementations of OpenSSL.

F5
They found that F5 products instantiated a number of oracles, with the most common type coming from the fact that they would respond with a `handshake_failure` message if the padding was invalid, and the connection could timeout if the padding was valid. Again, their findings allowed for a strong enough oracle to forge a digital signature.

Citrix
Vulnerable implementations run by Citrix were also identified, and the source of these oracles came from an invalid padding causing the connection to timeout. As this was the only difference, they did explain that the attack is inefficient, since accurately detecting timeouts is difficult and requires repeated querying to the oracle.

Radware
The oracle that the Radware implementation allowed for was very strong, whereby

messages not beginning with `0x0002` would reset the TCP connection. If the padding did begin with `0x0002`, then instead the server would respond with a `decrypt_error` message. Hence they were easily able to identify PKCS conforming messages.

Cisco ACE
Cisco ACE devices are no longer sold or supported, but they are vulnerable to Bleichenbacher's attack. Böck et al. discovered that different error types were answered with different error messages. These findings suffice as a padding oracle, and since all the ciphersuites offered by Cisco ACE devices utilise an RSA key exchange, it is not possible to implement a secure TLS configuration on these devices.

Erlang
Erlang is a programming language [3], and they noticed that different errors prompted different TLS error messages. Such differences allowed for a similar oracle to the one observed in the Radware implementation. They also found that WhatsApp utilised Erlang, and so it too was vulnerable to Bleichenbacher-style attacks.

Bouncy Castle
Bouncy Castle is a cryptographic API for Java [3], and their research identified that TLS implementations of Bouncy Castle also leaked enough information to provide a padding oracle. More specifically, if the `0x00` delimiter was not in the correct position within the padding, then the server would respond with a distinguishable error message.

WolfSSL
WolfSSL is an SSL/TLS library for embedded devices [3]. They discovered that if the `ClientKeyExchange` message was sent without the `ChangeCipherSpec` and the `Finished` messages, then a timeout would occur for a correctly padded message and an error would occur if the padding was invalid in any way. Although they explain that such an oracle is extremely weak, it is insecure nonetheless.

They also discovered old vulnerabilities in JSSE and MatrixSSL. We have already discussed the vulnerabilities of JSSE in Section 5.1.1 and 5.1.3 from 2014. However, in 2016, Juraj Somorovsky had also previously discovered padding oracle vulnerabilities in the open-source SSL/TLS library MatrixSSL [16]. As such, although the previously described vulnerable TLS implementations have mainly been fixed, again we are seeing a lack of good patch-management programmes. Therefore, it should not be assumed that these padding oracle vulnerabilities are no longer a problem. Furthermore, since this piece of literature was published, Böck et al. have released vulnerabilities for additional vendors; including Palo Alto Networks, IBM Domino and IBM WebSphere MQ. The details of these vulnerabilities can be found on their website (`https://robotattack.org/#patches`), and since the research is ongoing, more vendors will be added once fixes become available.

Of the 1 million hosts that they tested, 27,965 were vulnerable to a Bleichenbacher-style padding oracle attack. Furthermore, of the top 100 domains (according to Alexa), 27 of them were vulnerable – a worrying statistic for these reputable hosts. All things considered, although conducting Bleichenbacher's attack is not as straight-

forward as it was in 1998, this research demonstrates that even after 20 years, thwarting the attack is far from trivial. Therefore, great attention should be paid to ensure that an SSL/TLS implementation does not allow an attacker to in any way distinguish between valid and invalid padding structures.

5.3.3 Forging a Digital Signature

As a proof of concept, during their research, Böck et al. forged a digital signature of a chosen message using Facebook's private signature key. Throughout this project (and most of the literature), much of the discussion has revolved around decrypting an intercepted ciphertext. However, Bleichenbacher's algorithm is equally capable of forging the digital signature for any chosen message without knowledge of the private signature key. Thus, for completeness and to aid understanding of this proof of concept, we will now walk through an example that will allow us to forge such a signature. It is important to note that this only works if the server uses the same RSA key pair for encryption/decryption and signing purposes.

We will look to utilise the same 1024 bit RSA parameters from our example in Section 3.3. However, the major difference here is that the construction of the digital signature that we wish to forge will not be PKCS conforming to begin with. Therefore, in the blinding step, it no longer suffices to set $s_0 = 1$. This means that we require many more queries to the oracle, so for efficiency, let us suppose that the oracle is a TTT oracle. Again, we will only display the first 10 digits of the numbers in this section, but the full numbers can be found in Appendix D. To begin with we have:

$$n = \texttt{1584975239...} \quad e = \texttt{65537}$$

As mentioned previously, when we are looking to forge a digital signature, we replace the ciphertext c from the algorithm with the encoded message that we wish to sign. Thus, we first need to encode the message, and to do this we follow the advice in Section 9.2 of [6], which we briefly discussed in Section 2.2.1.

As such, the encoded message `EM` is of the form:

```
EM = 0x0001 || ff ff ...  ff ff || 0x00 || T
```

Here, `T` represents the Distinguished Encoding Rules described in [6]. To produce `T`, we hash the message we wish to sign using a standardised hash function, and then we append this digest to the algorithm ID for that hash function, as defined in Section 9.2 of [6].

Suppose the message that we wish to sign is "This is our message", and our chosen hash function is `SHA-256`. Our message digest `D` is:

```
D = 0xceda6028170b54f1c8e9cf7c58c4f0783bbb368b468a6314a2d5977e6dce2d90
```

The identifier `I` for `SHA-256` is `0x3031300d060960864801650304020105000420`, thus it follows that `T = I || D`.

As a result our encoded message is as follows:

```
EM = 0001ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffff003031300d060960864801650304020105000420ceda6028
170b54f1c8e9cf7c58c4f0783bbb368b468a6314a2d5977e6dce2d90
```

For clarity, we will now take the decimal value of `EM` and set this to be c. Hence we have:

$$c = 5486124068...$$

So we have our "ciphertext" c, and now all that differs between a digital signature forgery and decrypting a message is the value of s_0 in step 1 (and of course the value of s_0^{-1} in step 4).

Now we must begin the search for s_0 such that $c_0 \equiv c \cdot s_0^e \bmod n$ is PKCS conforming (so with a TTT oracle, this is when c_0 begins with `0x0002`). It turns out that $s_0 = 37448$, and as a result we have a PKCS conforming c_0, and we can continue the algorithm as discussed in Section 3.3.

Upon completion of the algorithm, we have successfully "decrypted" the "ciphertext" c_0 to produce m_0, and since the decryption and signing keys are the same, this also means that m_0 represents the digital signature for c_0. Hence we have:

$$m_0 = 5597129977...$$

However, we want the digital signature for $c \equiv c_0 \cdot (s_0^e)^{-1} \bmod n$. If we denote m as the digital signature for c, using equation (2.1) from Section 2.1.2, it follows that $m \equiv m_0 \cdot s_0^{-1} \bmod n$. We know m_0, s_0, and n, and so we can calculate m.

$$m = 1230124984...$$

We must now convert from decimal to hexadecimal, and as a result, it follows that the forged digital signature `DS` of the message "This is our message" is:

```
DS = 0xaf2cf4f06ad4ea7eb3d5c629180a3ea60fe7371b59268442774bcda21e80
a23d740b5a623479bb7569ecde829aa4cbb056b2333cb621b04cf63da7d3dc3074414
509ef6cadc1698ce6dd6fdb54df1289e802d5141581910d4a862d40fba337d7dba1d3
c8fbaf6312da9bcf54c0e26b5f992b60092fbba5e063e6f62464cd9b2b
```

With a strong TTT oracle and using the most optimised version of the algorithm, it took a total of 39644 queries to the oracle to forge this digital signature. 37448 of these queries were spent searching for s_0, so one can see that forging a digital signature is much more difficult than simply decrypting an already PKCS conforming ciphertext. However, if the oracle is strong, then such an attack is still perfectly feasible, as we have demonstrated here.

Chapter 6

Additional Applications

Throughout this project we have continually referred to Bleichenbacher-style attacks on SSL/TLS implementations. However, there are other applications that also employ RSA encryption with the PKCS #1 v1.5 padding scheme; so in this section we will briefly look at the XML document standards and the QUIC protocol as examples of this. We will also look at how the changes in TLS 1.3 aim to prevent Bleichenbacher-style attacks, and ultimately whether such changes are sufficient.

6.1 Exploiting the Format of XML

In 2012, Jager et al. published a paper demonstrating practical Bleichenbacher attacks against the PKCS #1 v1.5 key transport mechanism of XML encryption [17]. XML is an acronym for Extensible Markup Language, and XML encryption provides a means of securely transporting encrypted data, with the result being represented in the XML format. Full details of the structure can be found in [18], but for the purposes of this project, we will just observe the basics.

The structure of an XML encrypted message will consist of a header and a body. The header will contain message specific data such as timestamps and user information, then the body will contain the payload [17]. If the payload is to be encrypted, then the header will also contain the symmetric key for this encryption – which will be encrypted under the public encryption key of the receiver. So once the message has been received, the receiver can decrypt the symmetric key, and then decrypt the payload.

However, the symmetric key will need to be padded before encryption, and in the original XML standards, PKCS #1 v1.5 padding was mandatory [17]. We are no longer exchanging multiple protocol messages, as was the case in SSL/TLS, but instead just one message is sent containing both the header and body. Based on this set-up, Jager et al. demonstrated two different types of Bleichenbacher-style attacks which enabled an attacker to determine the symmetric key encrypted within the XML header, and thus determine the payload encrypted within the XML body. We will now briefly describe these attacks.

6.1.1 A Timing Attack

Their first attack stems from the fact that a web service only attempts to decrypt the XML-formatted payload if the decrypted symmetric key has a valid padding structure [17]. Additionally, if the symmetric key is valid but the decrypted payload is invalid, this will only be reported after all of the payload has been decrypted. As such, the larger the payload, the longer it will take to decrypt it. As a result, an attacker can purposely increase the size of the payload, and one can now see that a header containing a valid (and correctly padded) symmetric key will lead to a longer processing/response time than a header containing an invalid (and incorrectly padded) symmetric key.

Using this intuition, Jager et al. measured the minimum response time one should expect from a valid key. They then concluded that if the response time of any oracle query fell below the minimum response time, then the padded symmetric key contained within the XML-formatted header was not PKCS conforming. Hence it is clear that they had discovered a practical oracle which would distinguish between PKCS conforming ciphertexts and non-PKCS conforming ciphertexts. Using such an oracle, they were able to invoke Bleichenbacher's algorithm to deduce the symmetric key contained within the header, and in turn decrypt the payload contained within the body.

Their experiments demonstrated that such an attack could be successful on a local machine and via the internet. On a local machine they required a total of 321870 oracle queries, and this was able to recover the symmetric key in 200 minutes [17]. Although in general it seems unlikely that the victim would be on a local machine, they did comment on the fact that this is realistic if an attacker can rent a virtual machine in a cloud environment which uses the same physical hardware as the victim. Their evaluation of the attack via the internet is much less efficient due to network delays. However, they deduced that an attacker could realistically recover the symmetric key in less than one week. This may seem like a long time, but once the body of the XML message has been intercepted, the contents will not change. As such, unless the information is time-sensitive, an attacker will still be able to recover the potentially valuable payload.

6.1.2 Exploiting CBC Mode of Operation

Their second attack looks to exploit a weakness in the CBC mode of operation for symmetric encryption. As mentioned previously, first the symmetric key (denoted c_{key}) will be decrypted, and then the payload (denoted c_{data}) will be decrypted using c_{key}. They observed that if an error occurs during decryption, then an attacker will be informed of this. Such an error can be the result of an error in decrypting c_{key}, an error in decrypting c_{data} after successfully decrypting c_{key}, or the decryption of c_{data} is successful but it cannot be parsed (it contains non-printable characters or badly placed special characters) [17]. Note, the attacker would not be able to identify the exact reason for the error, and they would only be sure that a modified c_{key} and its padding is valid if no error was returned at all – the chances of which are small.

To avoid this problem, Jager et al. demonstrated that it is possible to modify

c_{data} to ensure that it will always be parsed successfully. To do this, they changed the contents of c_{data} to contain two randomly generated 16 byte blocks such that $c_{data} = (iv, C^{(1)})$. The ciphertext $c = (c_{key}, c_{data})$ is then submitted as an oracle query, where c_{key} represents $c_0(s_i)^e \bmod n$ in Bleichenbacher's algorithm. They also adjusted the metadata to inform the web service that the payload was encrypted in CBC mode. As such, first $C^{(1)}$ is decrypted, and then the result of this decryption is `XOR`'ed to the iv (initialisation vector) to produce the plaintext (denoted $data^{(1)}$). When working in CBC mode, the last byte of $data^{(1)}$ corresponds to the number of padding bytes that must be removed to decode the message [17]. If the last byte happens to be `0x10`, then this tells us that we should remove 16 bytes of padding, which of course means that the message is simply an empty string. This is what the attacker needs, since an empty string can always be parsed successfully.

Based on this, if an attacker can modify the iv in such a way that ensures the last byte of $data^{(1)}$ is `0x10`, then the receipt of an error is confirmation that the padding in c_{key} is invalid, and hence it is not PKCS conforming. Since the chosen iv is simply being `XOR`'ed to the decryption of $C^{(1)}$, it is possible to manipulate the last byte of $data^{(1)}$ by altering the last byte of the iv. Thus, by iterating through all possible options for the last byte of the iv, one such scenario must ensure that $data^{(1)}$ decodes to the empty string. If an error is returned on all of the 256 possible options for the last byte of the iv, then c_{key} is not PKCS conforming. However, if an error is not returned for any one of the 256 possible options for the last byte of the iv, then we have confirmation that c_{key} is PKCS conforming. As a result, we can move onto the next iteration of Bleichenbacher's algorithm.

Once again they demonstrated that this variation of the attack could be successful, and it required the attacker to send around 322000 oracle queries. However, since each query potentially requires 256 server requests, the total number of queries may be as high as 82 million [17]. As such, although we no longer have the issue of network delays, the sheer number of oracle queries required meant that such an attack would take around 5 days. That being said, as was the case with the timing-based attack via the internet, unless the payload contains time-sensitive information, such an attack could be devastating. A final point worth noting is that the countermeasure whereby we generate a random key if c_{key} is invalid does not thwart this variation of the attack. This is because if c_{key} is valid, then a random key will never be generated regardless. However, if c_{key} is invalid and a random key is generated, the chances of a successful decryption and parsing of the payload are slim, and so it is likely that we will still receive an error. Should a randomly generated key lead to successful decryption and parsing, then the attacker can simply submit the same query again. As a result, if the attacker receives no error for a second time, then they can almost guarantee that c_{key} is PKCS conforming.

6.2 QUIC Protocol

QUIC is an acronym for Quick UDP Internet Connections and it is a key-exchange protocol over UDP developed by Google [19]. It has reduced latency when compared to SSL/TLS by establishing a symmetric key using less protocol messages, but ultimately the end goal is the same. However, in 2015, Jager et al. demonstrated that

the QUIC protocol is devastatingly vulnerable to Bleichenbacher-style attacks, even though the protocol itself only allows the use of RSA for signing purposes and not encryption [20].

The reasoning behind this vulnerability boils down to the fact that although QUIC will utilise an RSA-signed Elliptic Curve Diffie-Hellman (ECDH) key share to establish a symmetric key with a client, if RSA keys are shared with SSL/TLS implementations, then an attacker can forge a digital signature (as shown in Section 5.3.3) [20]. Unfortunately, the only assurance a client has with regard to the authenticity of the connected server is a valid RSA digital signature on an ECDH key share, and a timestamp denoting the expiration of that signature. Furthermore, such digital signatures in QUIC are independent of the client connection, and so they can be calculated in advance. This means that if an attacker is able to compute such a digital signature, then a man-in-the-middle attack becomes trivial.

Although an attacker cannot exploit the QUIC protocol to forge a digital signature, when an RSA key is shared with other SSL/TLS implementations that allow for RSA encryption, they can be exploited to forge the required digital signature for the QUIC protocol. Since the attacker may choose the ECDH value and the expiration of the signature, the digital signature can not only be calculated in advance, but also be used on many occasions to instigate multiple man-in-the-middle attacks. As a result, an attacker may well be motivated to forge such a digital signature, even if the SSL/TLS padding oracle is very weak. Furthermore, since the digital signature is independent of the client connection, it does not matter if this takes days, weeks or even months; the end result will still allow for a man-in-the-middle attack.

6.3 Changes for TLS 1.3

TLS 1.2 has now been superseded by TLS 1.3, and therefore this should be the standard to follow for all SSL/TLS implementations. Full details can be found in [21], but for the interests of this project, the most intriguing change which looks to prevent Bleichenbacher-style attacks is the deprecation of RSA encryption. Instead, the standard recommends Diffie-Hellman and Elliptic Curve Diffie-Hellman (see Section 7.4 of [21]). However, despite RSA encryption being dropped from the standard, Jager et al. were able to demonstrate that TLS 1.3 implementations are still vulnerable to Bleichenbacher-style attacks [20].

The reasoning is very similar to that of the QUIC protocol. In TLS 1.3, first the client will send a `ClientHello` message (as before) and a `ClientKeyShare` message. The `ClientKeyShare` is the client's contribution to an (EC)DH key. The server then responds with a `ServerHello` message and a `ServerKeyShare` message (which is the server's contribution to an (EC)DH key) [21]. The server will then send a public key certificate containing the RSA verification key, and this is followed by a `CertVerify` message, which contains an RSA signature over all the messages that have been exchanged so far in the protocol.

As with QUIC, the only assurance a client has with regard to the authenticity of the connected server is a valid RSA digital signature. Since it is likely that RSA

keys will be shared with earlier versions of SSL/TLS implementations, if one of these implementations suffices as a padding oracle, then again such a digital signature can be forged. However, in TLS 1.3, this digital signature is dependent on the client connection, and more specifically, it is dependent on the contents of `ClientHello` and `ClientKeyShare`, which cannot be known in advance. As a result, the major difference between the vulnerability in QUIC and the vulnerability in TLS 1.3 is that an attacker wishing to conduct a man-in-the-middle attack must wait for the client to initiate a connection with the vulnerable server before conducting Bleichenbacher's attack to forge the digital signature. Since forging a digital signature can take a several hours at best, in general one would hope that the client would not wait such a length of time, rendering this attack highly unlikely. However, the existence of very strong and efficient padding oracles, such as those which were vulnerable to Special DROWN in Section 5.2.4, clearly indicates that such a vulnerability should not be ignored.

Nonetheless, removing RSA encryption from the standard will certainly help to reduce the impact of Bleichenbacher-style attacks. Furthermore, another advantage of deprecating RSA encryption is that RSA does not provide forward secrecy, whereas (EC)DH does. From a key management perspective, it is desirable to have this property because it provides assurance that a compromised long-term private key will not compromise all previously established short-term sessions keys.

Chapter 7

Attack Performance

Following on from Bleichenbacher's original algorithm discussed in Section 3.2 and the improvements discussed in Chapter 4, we will now present the results of our simulations to demonstrate how effective the optimisations have been since 1998.

The figures that we will present in this section are a result of simulated attacks against a 1024 bit RSA public key using our own source code (Python 2.7.15). The oracles that we investigated were all of those discussed in Section 2.2.3, with the exception of an FFF oracle. Unfortunately, under the time constraints, an FFF oracle is too strict to obtain substantial results for this project. In an attempt to avoid statistical anomalies, for each oracle we conducted 10000 different simulated attacks using the original algorithm, the improvements discussed in Section 4.1, the improvements discussed in Section 4.2, then with all improvements simultaneously.

7.1 Setting the Parameters

To ensure that our simulations accurately reflect any randomly chosen parameters, each of the 10000 tests for the above permutations utilised different prime factors of the modulus n, a different private decryption key d, a different `PMS` and a different non-zero padding string. We did, however, ensure that the public encryption key e remained unchanged. This value does not need to be random or secret, and setting $e = 65537 = 2^{16} + 1$ allows for efficient RSA encryption. Each test utilised a unique seed, and the random parameters for that test were generated using this seed and a deterministic pseudo-random number generator. Therefore, random generation worked as follows:

Defining the seed
Instead of being random, we gave the seeds a structure to ensure reproducibility of attack simulations. The structure was of the form:

$$x \,||\, y \,||\, z \,||\, 99 \,||\, n$$

where x represented the version of the algorithm, y represented the oracle type, z represented the test batch number and n represented the specific test number. For example; x was assigned 6, 7, 8 or 9, depending on the algorithm, y was a three character binary number where a 1 represented an F and a 0 represented a T, z was

between 0 and 9, then n ran from 1 to 1000. The 99 was simply a means of separating these identifiers, and the use of a batch number allowed us to easily manage the limited time available to conduct testing. As an example, the seed 9110299146 corresponds to test number 146 in batch number 2, implemented with the most optimised algorithm against an FFT oracle. Note that in Chapter 8 we set $z = 5$ throughout. Greater detail regarding the structure of these seeds can be found in Appendix E, and overall, this structure allowed for a total of 10000 tests against each of the 4 oracles using each of the 4 variations of the algorithm.

Generating p and q
From chapter 9.2.2 of [22], we know that in order to create a 1024 bit RSA modulus n, we must have $2^{511.5} < p, q < 2^{512}$. If p or q are smaller than this then we cannot guarantee that the RSA modulus will be 1024 bits in length. Based on this, for each test, the seed is used to randomly generate 2 integers between $2^{511.5}$ and 2^{512}, and then we simply set p and q respectively to be the smallest primes which are greater than or equal to these 2 randomly chosen values.

Generating d
Once we have generated p and q, calculating the private RSA decryption key is simply a case of finding $d = 65537^{-1} \bmod (p-1)(q-1)$. This can be done using the Extended Euclidean Algorithm, and since p and q are unique, so too is d.

Generating the pre-master secret and the non-zero padding string
To generate the `PMS`, we use the unique seed to generate a 48 byte string. There are no rules regarding the structure of the `PMS`, so its generation is simple since any 48 byte string is valid. Similarly, to generate the non-zero padding string, we can also use the unique seed. With a 1024 bit RSA modulus, the non-zero padding string contains 77 bytes. Therefore, we can generate a 77 byte string as mentioned above. However, since the padding string is non-zero, we must check to see if it contains a `0x00` byte. If it does then we generate a new padding string and check for `0x00` bytes again. The probability of a 77 byte padding string not containing a `0x00` byte is $(\frac{255}{256})^{77} \approx 0.74$, so randomly generating such a padding string is not difficult.

7.2 The Results

For each version of the algorithm, we will present the results of our attack simulations against the four oracles discussed previously. Table 7.1 contains the results using Bleichenbacher's original algorithm, Table 7.2 contains the results when optimising step 2b (as discussed in Section 4.1), Table 7.3 contains the results when we optimise step 2a and trim M_0 (as discussed in Section 4.2), then Table 7.4 contains the results when we implement all previously discussed optimisations simultaneously. The source code for the most optimised version of the algorithm can be found in Appendix F, and the individual optimisations are simply a subset of this source code. All 16 permutations of oracle and algorithm type will utilise different seeds, and as such, the parameters will vary across all 16 sets of 10000 tests. However, 10000 tests is a large sample size and provides good statistics. Therefore, we would expect similar results if we used the same seeds across all 16 sets of simulations.

Oracle	Original algorithm		
	Mean	Median	% that entered 2b
FFT	222806	174057	62.36%
FTT	215886	166855	61.52%
TFT	45840	26896	18.99%
TTT	42350	25534	17.39%

Table 7.1: Simulations using the original algorithm proposed in 1998.

Oracle	Klíma et al. improvements		
	Mean	Median	% that entered 2b
FFT	124767	86553	62.57%
FTT	123985	85245	61.77%
TFT	41720	26896	18.74%
TTT	35116	26027	17.48%

Table 7.2: Simulations using the improvements to step 2b proposed in 2003.

Oracle	Bardou et al. improvements			
	Mean	Median	% that entered 2b	Trimmers
FFT	97257	16293	22.59%	1500
FTT	85917	13240	20.01%	2000
TFT	20067	3906	3.48%	600
TTT	18911	3449	3.06%	500

Table 7.3: Simulations using the improvements to step 1 and step 2a proposed in 2012, along with the recommended limits for the trimming phase discussed in Section 4.2.1.

Oracle	Bardou et al. and Klíma et al. improvements			
	Mean	Median	% that entered 2b	Trimmers
FFT	61423	16036	22.33%	1500
FTT	56511	13478	20.13%	2000
TFT	12602	3860	3.68%	600
TTT	12347	3537	3.10%	500

Table 7.4: Simulations using all optimisations and the recommended limits for the trimming phase discussed in Section 4.2.1.

The results in Table 7.1, 7.2, 7.3 and 7.4 clearly demonstrate that, as the oracles become stricter, the mean number of queries required to conduct the attack increases. That being said, it is the median number of oracle queries that is of most interest to us. This is because throughout most of the sets of data, there are a few sets of parameters that require an abnormally large number of queries to conduct the attack. For example, one of the tests in Table 7.4 against an FFT oracle required 4402441 queries. As a result, the mean can be heavily influenced by just a handful of such examples. Instead, the conclusion we should make from Table 7.4 is that when using the most optimised algorithm against an FFT oracle, we can expect half of all attacks to succeed in less than 16036 oracle queries. Of course we can deduce similar results for an FTT, TFT and TTT oracle.

Furthermore, by comparing the results in Table 7.1 to the results in Table 7.2, we can see that the probability of entering step 2b is almost the same. In fact, had we used exactly the same seeds to produce these statistics, then we would have found that the probabilities of entering step 2b were identical. This is because the difference between the algorithm for Table 7.1 and the algorithm for Table 7.2 is the way in which step 2b works. So, although we have modified how step 2b works, it does not change the likelihood of entering step 2b in the first place. On the other hand, the results in Table 7.3 and Table 7.4 show a significant reduction in the percentage of tests entering step 2b. Hence we can conclude that trimming M_0 and "skipping" holes in step 2a decreases the probability of entering the computationally more expensive step 2b, and therefore aids in reducing the total number of oracle queries to complete the attack. We should also take note of the difference between the results in Table 7.3 and Table 7.4. Table 7.3 shows that the median number of queries is unaffected by the absence of the Klíma et al. optimisation when compared to Table 7.4. However, the mean is affected, and it is consistently higher. Therefore, when we combine both methods, although in general it seems that the Bardou et al. optimisation is more efficient, the Klíma et al. optimisation certainly helps to reduce the average number of oracle queries overall.

Despite the above analysis, the increased efficiency brought about by these optimisations may be visually clearer when presented graphically. As a result, Figure 7.1 illustrates the density distribution of oracle calls for an FFT oracle using the original algorithm, the improvements from 2003, and with the additional improvements from 2012. For clarity, we have omitted the sparse data points past 400000 oracle queries. The original graph simply consisted of much longer tails converging towards the x-axis.

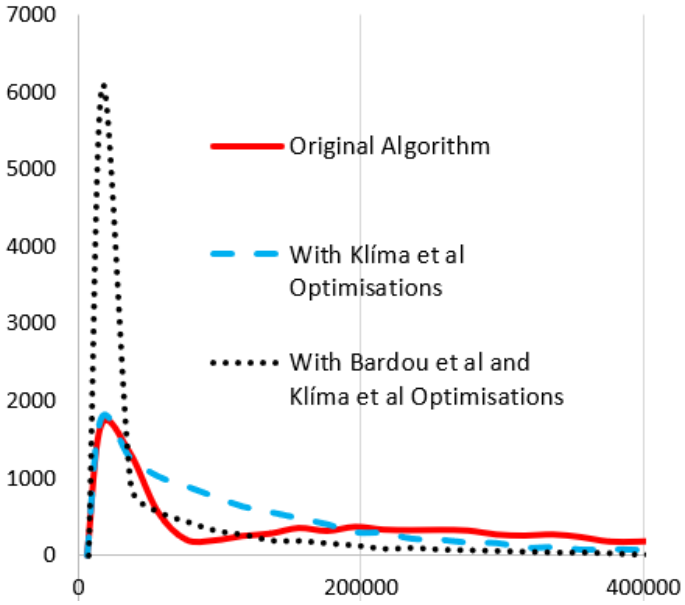

Figure 7.1: The density distribution across the 3 versions of the algorithm.

As you can see in Figure 7.1, the shape of the curve for the most optimised version

of the algorithm has a thick head and a very thin tail. Conversely, the curve for the original algorithm has a thinner head and a much thicker tail. As expected, this indicates that in the original algorithm, a greater volume of tests require a much larger number of oracle queries in comparison to the optimised version. The curve representing the Klíma et al. improvements does not really differ to the original algorithm, although we can see that it does reduce the number of tests which require greater than 200000 oracle queries. This is also expected, since tests that require such a large number of queries are more likely to enter step 2b, and therefore take advantage of the optimisation that this will invoke.

We can also graphically demonstrate the individual effects of the two separate optimisations against the original algorithm for both a TTT and FFT oracle. Figure 7.2 compares the Klíma et al. optimisation to the original algorithm, and Figure 7.3 compares the Bardou et al. optimisation (without the Klíma et al. optimisation) to the original algorithm. Again, the long tails converging towards the x-axis have been omitted.

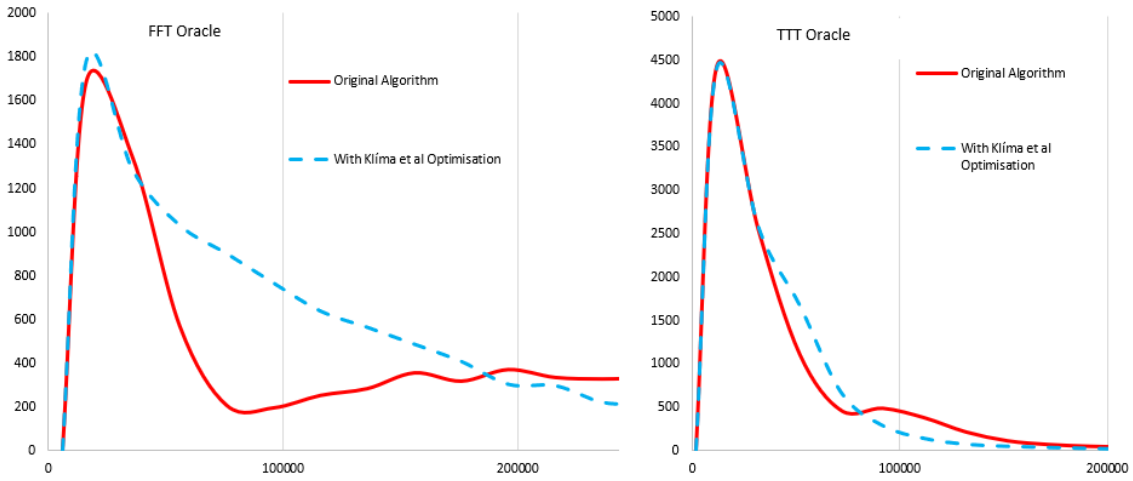

Figure 7.2: The density distribution of the Klíma et al. optimisation and the original algorithm against an FFT and TTT oracle.

By analysing Figure 7.2, we can see that the Klíma et al. improvement has less of an effect as the strength of the oracle increases. We know this because the two curves illustrated by the graph on the right (TTT oracle) are extremely similar. However, the left graph (FFT oracle) shows that with this optimisation, the head of the curve is slightly thicker, and the tail of the curve is thinner once we reach tests requiring more than 200000 oracle queries. This is the behaviour we would expect since a stronger oracle is less likely to enter step 2b, and hence the statistical differences for a TTT oracle are smaller.

Conversely, the Bardou et al. improvement has more of an effect as the strength of the oracle increases. The graph on the left of Figure 7.3 (FFT oracle) shows that with this optimisation, the head of the curve is much thicker and the tail of the curve is much thinner. However, the graph on the right (TTT oracle) shows that with this optimisation, the head of the curve almost hits a maximum thickness of 10,000, and the tail of the curve shrinks to a near-minimum thickness of 0 almost immediately.

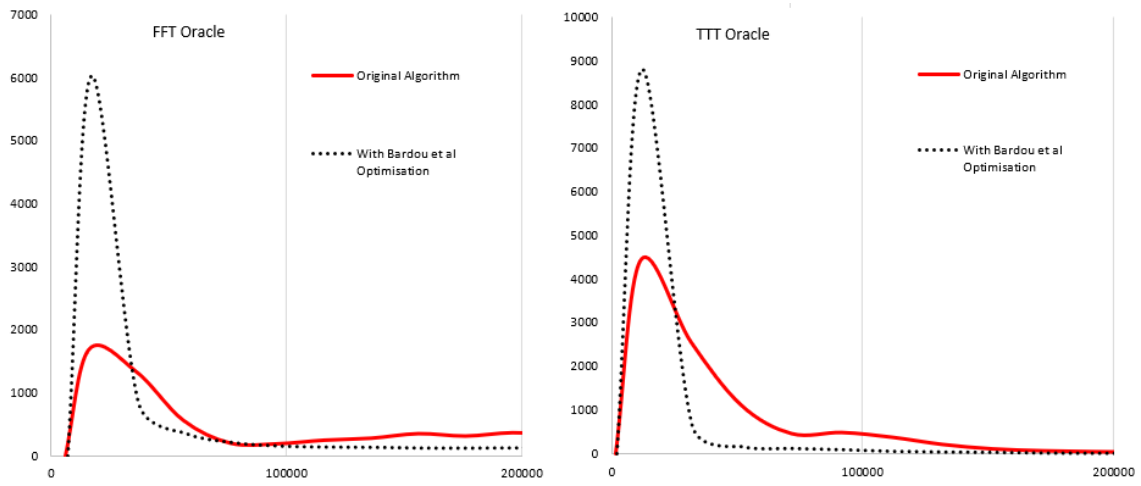

Figure 7.3: The density distribution of just the Bardou et al. optimisation and the original algorithm against an FFT and TTT oracle.

Clearly this optimisation provides an efficient attack against an FFT oracle, but the effects are visually clearer and stronger against a TTT oracle. Again, this is the behaviour we would expect because it is easier to find trimmers with a stronger oracle, and the more we trim M_0, the higher our starting value of s_1 becomes, and the larger the holes containing invalid values of s_1 become (as discussed in Section 4.2.2). Thus, we require less oracle queries overall to complete the attack.

Both of the above observations would be clearer if we were able to graphically demonstrate simulations against an FFF oracle, but unfortunately we do not have the computational power to present this. That being said, testing conducted by Bardou et al. against an FFF oracle in Section 2.5 of [8] shows that these observations hold true. They found that, although the addition of their optimisation did improve the attack when compared to the Klíma et al. optimisation alone, it only reduced the mean and median number of oracle calls by 13.4% and 6% respectively. However, in our FFT oracle experiments, when compared to the Klíma et al. optimisation, the addition of the Bardou et al. optimisation in Table 7.4 reduced the mean and median number of oracle calls by 50.8% and 81.5% respectively.

Finally, we can also visually demonstrate how the strength of an oracle can affect the attack. The graph in Figure 7.4 was produced using the full set of simulation results from Table 7.4, and it displays how the most optimised version of the attack performs against all 4 of the oracles discussed above. Again, we have omitted the tails converging towards the x-axis.

We can now clearly see that the full optimisation makes the attack extremely efficient across all four oracles, with very few attack simulations requiring more than 50,000 oracle queries. Furthermore, we can also see that the FFT curve and the FTT curve are extremely similar. As is the case with the TFT curve and the TTT curve. From this we can deduce that not allowing a `0x00` byte in the first 8 padding bytes has a relatively low impact on the number of oracle queries. This makes sense since the probability of there not being a `0x00` byte in the first 8 bytes of padding

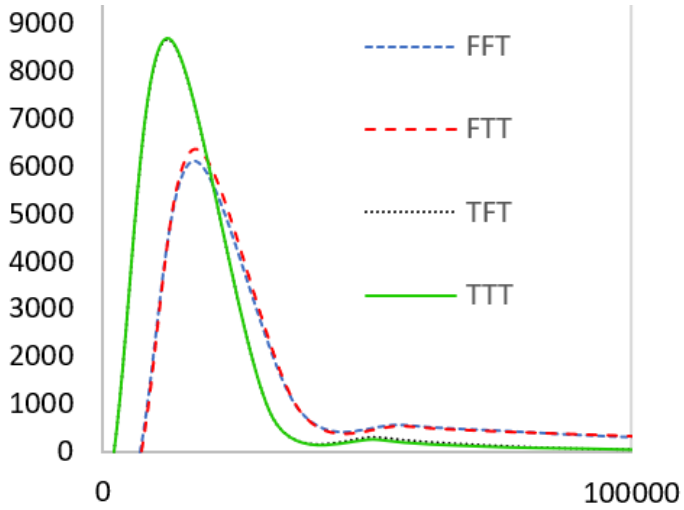

Figure 7.4: The density distribution across the 4 oracles using the most optimised version of the algorithm.

is $(\frac{255}{256})^8 \approx 0.97$. We can make a similar observation by comparing the TTT curve to the FTT curve. The only difference between these oracles is that an FTT oracle requires a `0x00` delimiter byte after the first 8 padding bytes. As is shown in Figure 7.4, this has a slightly larger impact on the number of oracle queries – which also makes sense, since the probability of there being such a `0x00` delimiter byte is $(1 - (\frac{255}{256})^{118}) \approx 0.37$.

With this in mind, it should now be abundantly clear that allowing an attacker access to any one of these four oracles will enable a practical attack against an SSL/TLS implementation. Therefore, it is imperative that servers perform strict padding checks and do not leak any information which may suffice as a padding oracle. Of course, the best way to ensure the security of an SSL/TLS implementation is to follow appropriate standards – including PKCS #1 v2.2 [6], TLS 1.2 [10] and TLS 1.3 [21].

Chapter 8

Searching for Trimmers

Up until this point, our discussion around trimmers has been fairly vague. This is also the case in the literature, and although the idea of trimmers was introduced in [8], they did not provide much of an indication as to the best way to search for them. In this section, we will present our own research around finding trimmers for the most optimised algorithm, and we will also illustrate our most effective methods to maximise the trimming of M_0 for the four oracles presented in Table 7.4.

Recall from Section 4.2.1 that if we can find m_0 and $m_0 \cdot ut^{-1} \bmod n$ that are PKCS conforming then t divides m_0. Furthermore, if $m_0 \cdot ut^{-1} \bmod n$ is PKCS conforming, then $m_0 \cdot ut^{-1} \bmod n = m_0 \frac{u}{t}$. Hence we have:

$$2B \leq m_0 \frac{u}{t} < 3B$$

As a result, it holds that $m_0 \frac{u}{t}$ begins with `0x0002`, and such a trimmer will allow us to trim the interval containing m_0.

The methodology behind our research includes varying the number of oracle calls we allow for trimming, the way in which we search for the initial trimming values u and t, then the way in which we seek the best numerators u_l and u_h to maximise trimming efficiency. We will begin this section by looking at the TTT oracle, but when we experiment with the other oracles, maximising the trimming of M_0 is not so straightforward. This is due to the fact that depending on the oracle, a valid query has a chance of being returned as invalid. For example, suppose it holds that t divides m_0 and $2B \leq m_0 \frac{u}{t} < 3B$. However, further suppose that we are working with a TFT oracle and the first 8 padding bytes of $m_0 \cdot ut^{-1} \bmod n$ contains a `0x00` byte. This means that although the values of u and t are valid trimmers for m_0, the oracle will return $m_0 \cdot ut^{-1} \bmod n$ as non-PKCS conforming, and so we will discard u and t. One can see that similar situations will arise with an FTT and FFT oracle. Based on this, we will look to find a balance between reducing the chances of such an error, whilst minimising the number of oracle queries required to conduct the attack.

From Section 2.2 of [8], we know that our search for u and t can be restricted to $\frac{2}{3} < \frac{u}{t} < \frac{3}{2}$. Furthermore, they explained that assuming m_0 is uniformly distributed in the original interval $[2B, 3B-1]$, the probability that $m_0 \frac{u}{t}$ is also in the interval is $(1/t)(3-2(t/u))$ if $2/3 < u/t < 1$, and $(1/t)(3(t/u)-2)$ if $1 < u/t < 3/2$. With this in mind, we know that trimmers are more likely to be successful when t is

small and when u and t are close together. Therefore, we can use this information as our starting point, and we will add to this in the following sections.

8.1 TTT Oracle

Working with a TTT oracle is simple because if t divides m_0 and $2B \leq m_0 \frac{u}{t} < 3B$, then $m_0 \cdot ut^{-1} \bmod n$ begins with `0x0002` and $m_0 \cdot ut^{-1} \bmod n$ will always be PKCS conforming. As a result, there is no situation whereby a valid pair of trimmers u, t will result in $m_0 \cdot ut^{-1} \bmod n$ not being PKCS conforming. Therefore, we do not need to worry about dealing with any false negatives, and in this section we simply need to establish the most efficient way to find the lowest and highest possible trimmers.

If $t > 4$ and t divides m_0, it must hold that either $\frac{(t+1)}{t}$ is a valid trimmer or $\frac{(t-1)}{t}$ is a valid trimmer. The former is true if $2B \leq m_0 < 2.5B$, and the latter is true if $2.5B \leq m_0 < 3B$. For values of $t \leq 4$, the only possible valid trimmers are $\frac{4}{3}$, $\frac{3}{4}$ and $\frac{5}{4}$. As a result, we should start our search with $\frac{4}{3}$ and $\frac{3}{4}$, and if $\frac{3}{4}$ is not valid then try $\frac{5}{4}$. Thereafter, starting from $t = 5$, we should try $\frac{(t-1)}{t}$, followed by $\frac{(t+1)}{t}$ if and only if $\frac{(t-1)}{t}$ is not a valid trimmer. At this point we set $t = t+1$ and repeat. So for each denominator t, we will either find one valid numerator u and add $\frac{u}{t}$ to our list of trimmers, or we find no valid numerator. However, at this stage of our search, we are most interested in finding as many possible values for t as possible, hence if $\frac{(t-1)}{t}$ is a valid trimmer then we gain no information if we were to also discover that $\frac{(t+1)}{t}$ is valid, hence we do not test it. Thus, this is the most efficient way to search for trimmers with a TTT oracle, and we can continue to do this until we reach our pre-determined trimming query limit – in [8], Bardou et al. recommended a limit of 500 queries for a TTT oracle. At the end of this step, we will have a list of valid trimmers $\frac{u_1}{t_1}, \frac{u_2}{t_2}, \ldots, \frac{u_n}{t_n}$.

Once we have reached our limit and compiled a set of valid trimmers, it remains to deduce the highest and lowest possible trimmers from that set. First, we calculate the lowest common multiple of our set of trimmer denominators $t_1, t_2, \ldots, t_n$. This is denoted t'. Then we must find the lowest and highest possible numerators, denoted u_l and u_h respectively, such that both $\frac{u_l}{t'}$ and $\frac{u_h}{t'}$ are valid trimmers.

We know that u_l must be greater than $\frac{2t'}{3}$ and u_h must be less than $\frac{3t'}{2}$. Furthermore, we can produce an upper bound for u_l by taking the lowest fraction $\frac{u_a}{t_a}$ from our list of trimmers and multiplying it by t'. We can use a similar method to produce a lower bound for u_h by taking the highest fraction $\frac{u_b}{t_b}$ from our list of trimmers and multiplying it by t'. As a result we have:

$$\frac{2t'}{3} < u_l \leq \frac{u_a t'}{t_a}$$
$$\frac{u_b t'}{t_b} \leq u_h < \frac{3t'}{2}$$

Once we have established the intervals that contain u_l and u_h, all that remains is to conduct a binary search over the intervals to find the lowest possible valid numerator

u_l, and the highest possible valid numerator u_h.

To find u_l using this method, we try the value that represents the middle of the interval containing u_l, denoted u_m. If such a value u_m provides a valid trimmer $\frac{u_m}{t'}$, then u_m becomes the new upper bound for u_l. If $\frac{u_m}{t'}$ is not a valid trimmer then instead u_m becomes the new lower bound for u_l. We then continue to half the size of the interval with each iteration until there is only one possible option for u_l. By searching in this way, it guarantees that we find the lowest possible valid trimmer $\frac{u_l}{t'}$. We can follow a similar technique to find u_h, but the slight difference is that if the middle value u_m implies a valid trimmer $\frac{u_m}{t'}$, then u_m becomes the new lower bound for u_h. Conversely, if $\frac{u_m}{t'}$ is not a valid trimmer, then u_m becomes the new upper bound for u_h. As before, this guarantees that we find the highest possible valid trimmer $\frac{u_h}{t'}$. As explained in Section 4.2.1, we then have:

$$2B \cdot \frac{t'}{u_l} \leq m_0 < 3B \cdot \frac{t'}{u_h}$$

Since we do not have to worry about false negatives when we query the oracle, the only variable we now need to consider is the trimming query limit. Although Bardou et al. recommended a limit of 500 for a TTT oracle, in Table 8.1 we will vary this limit and present the results of 1000 tests against a TTT oracle using the above method to search for trimmers.

Trimming Limit	Mean	Median
100	12326	4222
200	10823	3588
300	10518	3428
350	10342	3425
375	10231	3386
400	10079	3360
425	10071	3371
450	10053	3386
475	10010	3369
500	10034	3394
525	9973	3405
550	9920	3399
600	9904	3432
700	9842	3498
800	9624	3555
900	9638	3612

Table 8.1: The mean and median number of oracle queries required for a TTT oracle over 1000 tests as we vary the trimming limit.

From Table 8.1, we can see that as the trimming limit increases, the mean number of oracle queries decreases. However, this is not the case with the median, and it seems that a trimming limit between 375 and 500 ensures the median number of oracle queries is below 3400; with the lowest appearing when the trimming limit is 400. As a result, we can conclude that 400 is the optimal trimming limit for a TTT oracle – achieving a median of 3360 queries. We will now consider a TFT oracle.

8.2 TFT Oracle

When working with a TFT oracle, we can use the same intuition as Section 8.1, except here we must consider the possibility of false negatives. That is, when t divides m_0 and $m_0 \cdot ut^{-1} \bmod n$ begins with `0x0002`, but $m_0 \cdot ut^{-1} \bmod n$ is not PKCS conforming. This will result in a valid trimmer being absent from our trimming set, and it happens when there is a `0x00` byte in the first 8 padding bytes of $m_0 \cdot ut^{-1} \bmod n$. The probability of a valid trimmer being declared valid is:

$$P(\text{TFT}) = \left(\frac{255}{256}\right)^8 \approx 0.97$$

Hence, we can see that there is a 3% chance that the TFT oracle will provide a false negative. Although this chance is small, we must still consider whether it has an impact on the attack efficiency. There are 2 situations where such an error could result in a false negative. The first is when we are searching for trimming pairs u and t, and the second is when we are conducting our binary search to find the values u_l and u_h. A false negative is either scenario could reduce the efficiency in trimming M_0.

In order to reduce the chances of a false negative when searching for trimming pairs u and t, should we find that the oracle declares $\frac{(t-1)}{t}$ and $\frac{(t+1)}{t}$ as invalid, then we could also try $\frac{(t-2)}{t}$ and $\frac{(t+2)}{t}$. Suppose that both $\frac{(t-1)}{t}$ and $\frac{(t-2)}{t}$ are valid trimmers. Although there is a 3% chance that $\frac{(t-1)}{t}$ will be declared invalid by the oracle, the chances of both $\frac{(t-1)}{t}$ and $\frac{(t-2)}{t}$ being declared invalid is 0.09%. Hence, we can almost eliminate the possibility of losing out on a valid denominator t. However, by reducing this chance of error, it does cost additional queries to the oracle. Therefore, it is important to find a balance between the two. For the purposes of notation, the number of trimming samples will be denoted (x, y), where x represents the number of tested numerators below t, and y represents the number of tested numerators above t. For example, a trimming sample size of $(1, 2)$ means that for a given denominator t, we will consider 1 numerator below t and 2 numerators above t. Thus, the numerators to be considered in this example are $(t-1)$, $(t+1)$ and $(t+2)$.

We can look to use a similar technique when conducting our binary search. Suppose that during our search for u_l, we find that u_m is declared an invalid numerator by the oracle. It could be that it is simply invalid, or it may be part of the 3% of false negatives. If it is declared invalid then we can try $u_m + 1$ (or $u_m - 1$ when searching for u_h). If u_m was falsely declared invalid then $u_m + 1$ will definitely be a valid numerator. As a result, the chances of both u_m and $u_m + 1$ both being falsely declared invalid is 0.09%. Hence, this method of sampling around the middle value for each interval significantly reduces the inefficiencies brought about by false negatives, but of course it also costs additional queries to the oracle. Again, we will look to try and balance this.

Table 8.2 illustrates 72 sets of results as we vary the size of the trimming samples, binary search samples and the trimming limit. We conducted 1000 tests for each permutations, and for clarity, note that a binary search sample of size 0 and

a trimming sample of size $(1, 1)$ implements the same version of the algorithm that we discussed in Section 8.1.

0 Binary Samples				
Trimming Samples	(1,1)	(2,1)	(1,2)	(2,2)
Trimming Limit				
300	3976	4125	4130	4232
400	3866	4051	4045	4160
500	3900	3967	3966	4093
600	3934	4004	4005	4067
700	3940	4036	4036	4163
800	3998	4081	4120	4234
1 Binary Sample				
Trimming Samples	(1,1)	(2,1)	(1,2)	(2,2)
Trimming Limit				
300	3777	3877	3889	4042
400	3692	3867	3864	3930
500	3743	3792	3792	3924
600	3774	3849	3849	3892
700	3754	3890	3891	3980
800	3827	3901	3901	4025
2 Binary Samples				
Trimming Samples	(1,1)	(2,1)	(1,2)	(2,2)
Trimming Limit				
300	3773	3857	3874	4028
400	3699	3863	3863	3904
500	3745	3799	3799	3915
600	3784	3853	3853	3899
700	3767	3893	3892	3981
800	3835	3911	3911	4030

Table 8.2: The median number of queries for a TFT oracle over 1000 tests as we vary the size of the trimming samples, binary search samples and trimming limit.

By observing the results in Table 8.2, we see that a trimming sample of size $(1, 1)$ allows for the most efficient trimming of M_0. This means that for each denominator t, we should only tests to see if $\frac{(t-1)}{t}$ and $\frac{(t+1)}{t}$ are valid trimmers. Furthermore, the optimal size for our binary search sample is 1. Although a binary search sample of size 2 would further reduce the chances of a false negative, this reduction in error is outweighed by the additional query costs. Finally, under the aforementioned optimal conditions, it appears that 400 is the optimal trimming limit. With these parameters, we were able to achieve a median of 3692 queries to the oracle.

8.3 FTT Oracle

As with a TFT oracle, again we must consider false negatives arising during our implementation. With an FTT oracle, this will happen if t divides m_0 and $m_0 \cdot ut^{-1} \bmod n$ begins with `0x0002`, but there is no `0x00` delimiter byte after the first

8 padding bytes. The probability of a valid trimmer being declared valid is:

$$P(\text{FTT}) = 1 - \left(\frac{255}{256}\right)^{118} \approx 0.37$$

Hence, there is a 63% chance that the FTT oracle will respond with a false negative. Again, we will look to balance trimming efficiency against the median number of oracle queries required to conduct the attack, and we will invoke the same notation and experimental techniques as Section 8.2. Furthermore, in Section 8.2 we found that a binary search sample of size 1 provided the optimal results. With a TFT oracle, recall that such a sample size means that the chances of both u_m and $u_m + 1$ being falsely declared invalid numerators is 0.09%. However, with an FTT oracle, we have just shown that there is a 63% chance that the oracle will respond with a false negative. Therefore, in order to get close to the low error rate of 0.09%, we require a binary search sample of size 15; since $(0.63)^{15} \approx 0.00098 = 0.098\%$. As a result, we used this as a starting point for our experimentation. Also, due to the large number of possible permutations, we begin by following the advice of Bardou et al. in [8] by setting the trimming limit to 2000. Our initial set of results can be found in Table 8.3.

Binary Samples	Trimming Samples			
	(1,1)	(1,2)	(2,2)	(2,3)
12	12095	11901	11916	12053
13	12060	11877	11901	11998
14	12053	11828	11902	11997
15	12053	11837	11877	12013
16	12060	11877	11877	12011

Table 8.3: The median number of queries for an FTT oracle over 1000 tests with a trimming limit of 2000, and varying sizes of trimming and binary search samples.

Table 8.3 illustrates 20 sets of results as we vary the size of the trimming and binary search samples, with a trimming limit of 2000. The results show that with such a trimming limit, a trimming sample of size $(1, 2)$ allows for the most efficient trimming of M_0. This means that for each denominator t, we should only test to see if $\frac{(t-1)}{t}$, $\frac{(t+1)}{t}$ and $\frac{(t+2)}{t}$ are valid trimmers. Furthermore, the optimal size for our binary search sample is 14. This value ensures that the chance of a false negative is reduced to 0.155%.

Although Table 8.3 provides an optimal median of 11828 queries for an FTT oracle, these figures only hold true when we set the trimming limit to 2000. As a result, we will now utilise the strongest parameters to conduct additional experiments with alternative trimming limits – both above and below 2000.

As we can see from Table 8.4, 2000 is perhaps not the the optimal trimming limit. Instead the results show that the higher trimming limit of 2600 actually leads to a reduced median of 11575 oracle queries.

Trimming Limit	Median
1300	12353
1500	11954
1700	11850
1900	11832
2000	11828
2100	11790
2300	11597
2400	11595
2500	11579
2600	11575
2700	11615
2800	11713

Table 8.4: The median number of oracle queries for an FTT oracle over 1000 tests as we vary the trimming limit – utilising a trimming sample of size $(1, 2)$ and a binary sample of size 14.

Finally, we took the optimal trimming limit and binary search sample size, then tested it with a trimming sample of size $(2, 1)$. Based on our research with a TFT oracle, the results are likely to be similar to when the trimming sample size is $(1, 2)$. We found that this slight alteration provided a median of 11795 oracle queries. As a result, we can confirm that a trimming limit of 2600, a trimming sample of size $(1, 2)$, and a binary sample of size 14 are the optimal parameters for an FTT oracle – achieving a median of 11575 oracle queries.

8.4 FFT Oracle

Finally, we consider an FFT oracle. Such an oracle will return a false negative if t divides m_0 and $m_0 \cdot ut^{-1} \bmod n$ begins with `0x0002`, but there is no `0x00` delimiter byte after the first 8 padding bytes, or there is a `0x00` byte in the first 8 padding bytes. Here the probability of a valid trimmer being declared valid is:

$$P(\text{FFT}) = \left(1 - \left(\frac{255}{256}\right)^{118}\right) \cdot \left(\frac{255}{256}\right)^{8} \approx 0.36$$

Hence, there is a 64% chance that an FFT oracle will respond with a false negative – very similar to that of an FTT oracle. By using the same intuition as we did in Section 8.3, we require a binary search sample of size 16 to get close to the low error rate of 0.09%. This is because $(0.64)^{16} \approx 0.00079 = 0.079\%$. Once again, we will use this as a starting point for our experimentation, and we will follow the advice of Bardou et al. in [8] by setting the trimming limit to 1500.

Table 8.5 presents 15 sets of results as we vary the size of the trimming and binary search samples, with a trimming limit of 1500. These results show that with a trimming limit of 1500, a trimming sample size of $(1, 2)$ allows for the most efficient trimming of M_0. So for each denominator t, we should only test the validity of $\frac{(t-1)}{t}$, $\frac{(t+1)}{t}$ and $\frac{(t+2)}{t}$. We can also see that the optimal size for our binary search is 16,

Binary Samples	Trimming Samples		
	(1,1)	(1,2)	(2,2)
14	16190	15430	15785
15	16105	15395	15715
16	16052	15324	15669
17	16052	15324	15669
18	16052	15324	15669

Table 8.5: The median number of queries for an FFT oracle over 1000 tests with a trimming limit of 1500, and varying sizes of trimming and binary search samples.

since an increase to 17 or 18 does not improve the median. This means that the chances of a false negative is approximately 0.079%. As we did with the FTT oracle, we will now take these optimal parameters to conduct additional experiments as we vary the trimming limit both above and below 1500.

Trimming Limit	Median
1300	15655
1500	15324
1900	15172
2000	14768
2100	14868
2200	14787
2300	14752
2400	14740
2500	14840
2600	14807
2700	14907
2800	14869
2900	14942
3000	14940

Table 8.6: The median number of oracle queries for an FFT oracle over 1000 tests as we vary the trimming limit – utilising a trimming sample of size $(1, 2)$ and a binary sample of size 16.

Table 8.6 illustrates that in our experiments, a trimming of limit of 2400 actually provides a lower median than a trimming limit of 1500 – reducing the median number of oracle queries from 15324 to 14740. Again, we took the optimal trimming limit and binary search sample size, and tested it with a trimming sample of size $(2, 1)$. As with an FTT oracle, we found that this slightly increased the median number of oracle queries over 1000 tests to 14828. Therefore, when working with an FFT oracle, it appears that a trimming limit of 2400, a trimming sample of size $(1, 2)$, and a binary search sample of size 16 are the optimal parameters – achieving the aforementioned median of 14740 oracle queries.

8.5 Summary

Having conducted extensive research into optimising the search for trimmers, we have demonstrated that the most efficient way to do so varies, depending on the strength of the oracle. When working with a TTT oracle, the search is intuitive. However, with other oracles that may reject valid trimming pairs u and t, false negatives must be considered. Our research illustrates that the optimal binary search error rates for TFT, FTT and FFT oracles are 0.098%, 0.155% and 0.079% respectively. That being said, we were only able to conduct 1000 tests for each of our experiments. Therefore, with larger sample sizes, it is possible that we could discover alternative optimal error rates. Nonetheless, for a TTT, TFT, FTT and FFT oracle, we achieved a median of 3360, 3692, 11575 and 14740 oracle queries respectively.

When compared to the statistics presented in Section 2.5 of [8], unfortunately we have not been able to reach their level of optimisation for an FTT or FFT oracle; where they achieved a median of 11276 and 14501 respectively. However, we have been able to improve upon their median number of oracle queries for a TTT (3768) and TFT (4014) oracle.

In an attempt to demonstrate that these improvements are not the result of statistical flukes, we conducted some additional experimentation; this time utilising 10000 test simulations. For a TTT oracle, this larger test size produced a median of 3420 oracle queries. Then, for a TFT oracle, this larger test size produced a median of 3713 oracle queries. Hence, it is likely that our trimming search techniques for both a TTT and TFT oracle are more efficient than those established by Bardou et al. in 2012. However, more work is required on our part to improve upon the statistics for an FTT and FFT oracle.

Chapter 9

Conclusion

In this project, we have provided a comprehensive overview with significant depth to allow for a strong technical understanding of Bleichenbacher-style attacks. As a result, we have been able to bridge the gaps between the available literature, and this project should suffice as a single point-of-reference to both a technical and non-technical reader. We demonstrated examples of precisely how the algorithm works, and were able to quantitatively investigate the different optimisations; along with their individual contributions to the attack. We also provided details on how Bleichenbacher-style attacks still exist within modern SSL/TLS implementations, and throughout our attack simulations, we were able to provide analysis on the efficiency of the attack under different parameters. In Section 3.2.2, we uncovered and corrected an error that appears in the original attack publication [1] and all subsequent publications. We then went on to present the results of a significant number of attack simulations, of which provides a more thorough overview for the attack optimisations than any other available literature.

In addition to this, our research on searching for trimmers provides optimal parameters with justification – something which is absent from the literature since the introduction of trimmers in 2012. We were able to delve deeper into the findings of Bardou et al. in [8], and in doing so, for a TTT and TFT oracle, we were able to improve upon their optimal results – reducing the medians from 3768 and 4014 to 3360 and 3692 respectively. Unfortunately, we were not able to do the same with an FTT or FFT oracle, so this is an area that warrants further research and experimentation. Our findings may be due to statistical flukes, or it may be that some of the methods presented in this project are simply more or less efficient. However, all of the results are consistent with Bardou et al., and so we hope that our findings in Chapter 8 complement those in [8], and perhaps provide the reader with a greater level of understanding and appreciation for their optimisation.

In Section 6.3, we mentioned that RSA encryption has been deprecated in TLS 1.3. However, although its deprecation significantly hinders Bleichenbacher-style attacks, the protocol is not completely immune. Therefore, great care must still be taken when implementing the protocol. For versions of SSL/TLS 1.2 and earlier, although RSA encryption is permitted, perhaps a more sensible approach would be to utilise (Elliptic Curve) Diffie-Hellman key agreements for encryption. Not only does this make the protocol immune to all known Bleichenbacher-style attacks, but

it also has the added benefit of providing forward secrecy. Based on this, in many ways it seems that the deprecation of RSA encryption is a good decision to help provide assurance of SSL/TLS security.

We have also looked at an alternative padding scheme known as RSA-OAEP. However, despite the weaknesses that we can attribute to the PKCS #1 v1.5 padding scheme, the chosen ciphertext attack by James Manger [7] is comparatively more damaging to RSA-OAEP. With a 1024 bit RSA key, such an attack requires around 1000 oracle queries to decrypt a chosen ciphertext. As a result, one can see that switching to a different padding scheme does not eliminate the security threats. Therefore, instead of trying to avoid the vulnerabilities of PKCS #1 v1.5 padding, perhaps we should look to ensure that all appropriate countermeasures have been taken to thwart Bleichenbacher's attack. By following Section 7.4.7.1 of TLS 1.2 [10] and Appendix E.7 of TLS 1.3 [21], such a task is perfectly achievable. In general, it is imperative that we do not allow for any of the four oracles discussed in this project, or the weaker FFF and BVO oracles that we discussed in Section 2.2.3 and Section 4.1.1 respectively. In doing so, we will ensure that incorrectly formatted messages (including incorrect SSL/TLS version numbers) will be treated in a way that is indistinguishable to those that are correctly formatted. This must include error messages, timing differences, connection time-outs and any other means of information leakage that we have discussed throughout this project.

Finally, the common practice of sharing static RSA keys across multiple servers and SSL/TLS protocol versions also leaves websites vulnerable to Bleichenbacher-style attacks. Coupled with the fact that many implementations favoured weak encryption over no encryption was precisely what lead to the pervasiveness of the DROWN attack discussed in Section 5.2. Therefore, all things considered, it is clear that the SSL/TLS standards should be carefully followed, padding schemes must be properly implemented, patches for SSL/TLS implementations should be applied quickly, export/weak ciphersuites and SSLv2 should be avoided, and good public key management is paramount.

Looking into the future, the use of RSA encryption is becoming less common – and this will continue to be the case as more servers look to implement TLS 1.3. However, these servers will still need to support earlier versions of the protocol for backwards compatibility. Therefore, the option to naively enable RSA encryption may remain for many years to come. Furthermore, it is likely that alternative means of confirming side-channel leakage will be or may have already been discovered. This may lead to alternative oracles, and ultimately another way of enabling this 20-year-old attack. That being said, using RSA solely for the purpose of producing digital signatures is secure – provided the key is not utilised elsewhere for encryption purposes. As a result, the ground-breaking findings of Rivest, Shamir and Adleman in 1978 [4] are likely to be practically implemented for many years to come.

Bibliography

[1] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. *Advances in Cryptology: Proceedings of CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12, 1998.

[2] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Parr, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. *Proceedings of the 25th USENIX Security Symposium*, pages 689–706, 2016.

[3] Hanno Böck, Juraj Somorovsky, and Craig Young. Return Of Bleichenbacher's Oracle Threat (ROBOT). *Cryptology ePrint Archive*, report 2017/1189, 2017.

[4] Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, volume 21(2), pages 120–126, 1978.

[5] Burt Kaliski. PKCS #1: RSA Encryption Version 1.5. *RFC* 2313, pages 1–19. `https://doi.org/10.17487/RFC2313`, 1998.

[6] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. *RFC* 8017, pages 1–78. `https://doi.org/10.17487/RFC8017`, 2016.

[7] James Manger. A chosen ciphertext attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS #1 v2.0. *Advances in Cryptology: Proceedings of CRYPTO 2001*, pages 230–238, 2001.

[8] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. *Advances in Cryptology: Proceedings of CRYPTO 2012*, pages 608–625, 2012.

[9] Keith Martin. *Everyday Cryptography (1st Edition)*. Oxford University Press, 2012.

[10] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. *RFC* 5246. `https://rfc-editor.org/rfc/rfc5246.txt`, 1998.

[11] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. *Cryptographic Hardware and Embedded Systems: Proceedings of CHES 2003*, pages 426–440, 2003.

[12] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. *23rd USENIX Security Symposium (USENIX Security 14)*, pages 733–748, 2014.

[13] Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. *RFC* 2246, pages 1–80. `https://doi.org/10.17487/RFC2246`, 1999.

[14] Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. *RFC* 4346, pages 1–87. `https://doi.org/10.17487/RFC4346`, 2006.

[15] Dr. Taher Elgamal and Kipp Hickman. The SSL Protocol. *Netscape Communications Corporation*. `https://datatracker.ietf.org/doc/html/draft-hickman-netscape-ssl-00`, 1995.

[16] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[17] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. *Proceedings of the 17th European Symposium on Research in Computer Security*, pages 752–769, 2012.

[18] Donald Eastlake, Joseph Reagle, Frederick Hirsch, Thomas Roessler, Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, and Magnus Nyström. XML Encryption Syntax and Processing Version 1.1. *W3C Recommendation*. `https://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/`, 2013.

[19] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. QUIC: A UDP-based secure and reliable transport for HTTP/2. *Google*. `https://tools.ietf.org/html/draft-hamilton-early-deployment-quic-00#section-3`, 2015.

[20] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1185–1196, 2015.

[21] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.3 (Draft). *Internet Engineering Task Force*. `https://tools.ietf.org/html/draft-ietf-tls-tls13-28`, 2018.

[22] Keith Mayes and Konstantinos Markantonakis. *Smart Cards, Tokens, Security and Applications (2nd Edition)*. Springer, 2017.

Appendix

A An Example Implementation of the Original Algorithm

```
from __future__ import division
import gmpy2
from gmpy2 import *
import math
import random
from random import randint
import time
import numpy as np

# This is for an FFT oracle

p = 12774566182638508651066048521819586493381800658856066616264070095136686959270020000164522965039319084386478190190419560331300700277241768836582806004529129
q = 12407272516930135069018312377913966218219495485579857780039745404167360117666305470818640640372546403429975909724368921508290782020989305312628056890880939
n = p * q
e = 65537

def eea(a, b):
    if b == 0:
        return (1, 0)
    (q, r) = (a // b, a % b)
    (s, t) = eea(b, r)
    return (t, s - (q * t))

def find_inverse(x, y):
    inv = eea(x, y)[0]
    if inv < 1:
        inv += y
    return inv

d = find_inverse(e, phi)

def zeroinpaddingcheck(x):
    for j in range(3, 18):
        if j % 2 == 1:
            if hex(x)[j:(j + 2)] == "00":
                return 1
                break
            else:
```

```
                continue
        else:
            continue

def zeroinpadding(x):
    for j in range(0, (len(x) - 1)):
        if j % 2 == 0:
            if x[j:(j + 2)] == "00":
                return 1
                break
            else:
                continue
        else:
            continue

def delimiterchecker(x):
    a = len(hex(x))
    b = a - 2
    for j in range(19, (b + 1)):
        if j % 2 == 1:
            if hex(x)[j:(j + 2)] == "00":
                return 1
                break
            else:
                continue
        else:
            continue

def oracle(query):
    global counter
    counter += 1
    v = powmod(query, d, n)
    if bottom <= v and v <= top and delimiterchecker(v) == 1 and
       zeroinpaddingcheck(v) != 1:
        return(1)
        # The padding is correct
    else:
        return(0)
        # we get an error message

def range(start, stop):
    while start < stop:
        yield start
        start += 1

def range_overlap_adjust(list_ranges):
    # https://stackoverflow.com/questions/15273693/python-union-of-
       multiple-ranges
    overlap_corrected = []
    for start, stop in sorted(list_ranges):
        if overlap_corrected and start - 1 <= overlap_corrected
           [-1][1] and stop >= overlap_corrected[-1][1]:
            overlap_corrected[-1] = min(overlap_corrected[-1][0],
               start), stop
        elif overlap_corrected and start <= overlap_corrected
           [-1][1] and stop <= overlap_corrected[-1][1]:
            break
        else:
```

```
            overlap_corrected.append((start, stop))
    return overlap_corrected

def ceildiv(a, b):
    # http://stackoverflow.com/a/17511341
    return -(-a // b)

def floordiv(a, b):
    # http://stackoverflow.com/a/17511341
    return a // b

def PMS():
    global message
    a = pow(16, 95)
    b = (pow(16, 96)) - 1
    f = random.randint(a, b)
    message = hex(f)[2:-1]

def pad():
    global padding
    global encoding
    global decimal_of_encoding
    global ciphertext
    r = (k - 3 - ((len(message)) / 2)) * 2
    a = pow(16, (r - 1))
    b = pow(16, r) - 1
    f = random.randint(a, b)
    padding = hex(f)[2:-1]
    while zeroinpadding(padding) == 1:
        g = random.randint(a, b)
        padding = hex(g)[2:-1]
    encoding = "0002" + padding + "00" + message
    decimal_of_encoding = int(encoding, 16)
    ciphertext = powmod(decimal_of_encoding, e, n)

def step_1(input):
    global i
    global c_0
    i = 0
    for s in range(1, int(n)):
        w = int(powmod(s, e, n))
        binding = int((w * ciphertext) % n)
        if oracle(binding) == 1:
            list_s.append(s)
            break
        else:
            continue
    c_0 = binding
    M = [(2 * B, (3 * B) - 1)]
    list_M.append(M)
    i = i + 1

def step_2a():
    global i
    global c_0
    global s
    for s in range((ceildiv(n, (3 * B))), n):
        x = int(powmod(s, e, n))
```

```
        attempt2a = int((x * c_0) % n)
        if oracle(attempt2a) == 1:
            list_s.append(int(s))
            break
        else:
            continue

def step_2bc():
    global i
    global c_0
    global s
    i = i + 1
    # This is step 2b
    if i > 1 and len(list_M[i - 1]) > 1:
        for s in range(list_s[i - 1] + 1, n):
            y = int(powmod(s, e, n))
            attempt2b = int((y * c_0) % n)
            if oracle(attempt2b) == 1:
                list_s.append(int(s))
                break
            else:
                continue

    # This is step 2c
    elif i > 1 and len(list_M[i - 1]) == 1:)
        found = False
        r = ceildiv(2 * (((list_M[i - 1][0][1] * list_s[i - 1]) -
           (2 * B))), n)
        while not found:
            for s in range(ceildiv(((2 * B) + (r * n)), list_M[i -
               1][0][1]), (ceildiv(((3 * B) + (r * n)), list_M[i -
               1][0][0]))):
                z = int(powmod(s, e, n))
                attempt2c = int((z * c_0) % n)
                if oracle(attempt2c) == 1:
                    found = True
                    list_s.append(int(s))
                    break
            r = r + 1
    else:
        print("error")

def step_3():
    global i
    global c_0
    list_temp = []
    for j in range(0, len(list_M[i - 1])):
        for r in range((ceildiv(((list_M[i - 1][j][0] * list_s[i])
           - (3 * B + 1)), n)), (floordiv((list_M[i - 1][j][1] *
           list_s[i] - 2 * B), n)) + 1):
            z = ceildiv((2 * B + r * n), list_s[i])
            list_temp.append(((int(max(list_M[i - 1][j][0], min(
               ceildiv((2 * B + r * n), list_s[i]), list_M[i - 1][j
               ][1])))), int(min(list_M[i - 1][j][1], max(floordiv
               (((3 * B - 1) + r * n), list_s[i]), list_M[i - 1][j
               ][0])))))
    list_M.append(range_overlap_adjust(list_temp))
```

```
def step_4():
    global i
    global c_0
    inverse = find_inverse(list_s[0], n)
    message = list_M[i][0][0] * inverse % n
    print("The decrypted ciphertext is {}".format(message))
    if message != decimal_of_encoding:
        print "Error - Decryption failed!"

def main(ciphertext):
    global counter
    global bottom
    global top
    global list_s
    global B
    global list_M
    B = int(pow(2, 8 * (k - 2)))
    bottom = 2 * B
    top = ((3 * B) - 1)
    counter = 0
    list_s = []  # this is where we store our s values
    list_M = []  # this is where we store the possible intervals
    t0 = time.time()
    step_1(ciphertext)
    step_2a()
    step_3()
    while len(list_M[i]) != 1 or list_M[i][0][0] != list_M[i
       ][0][1]:
        step_2bc()
        step_3()
    else:
        step_4()
    t1 = time.time()
    duration = t1 - t0
    print("It took {} seconds, and required {} calls to the Oracle
       ".format(duration, counter))

def test(x):
    Oracle_times = []
    u = 1
    for u in range(1, (x + 1)):
        global t
        t = int("9110991000")
        random.seed(t)
        print ("Seed {}: {}".format(u, t))
        PMS()
        pad()
        main(ciphertext)
        Oracle_times.append(counter)
        u = (u + 1)

test(1)
```

B Details for the Example in Section 3.3

```
n = 158497523913555876747567486455334446943697697850525957180228510
509398100752881436105780492049427802220707002137152229434111991799 0
918350753264367062123042487366644057367440411667898349892395924939 6
```

```
940576872415025116462794511904689281816894186939748682944275185968459447582138190274294534224155921297269637213 1

e = 65537

c = 526782076432733192013113050697510338947448548695162256597979552108671432282696081693562188354972235391626492207454128727168812708642144750713930055027926363900963557106438633904094341890596040841442892225793582986403831598196739465622246016952992874375914276707086600919933210985396608310042500233282900900 33

M_0 = {[54861240687936886832559362511872092700743926359323320701120019884561973817596729471651756995363627936132847253378721117449581838627446479032241037182456702996144987007100062645355901977919340246415125412623597951915939539289081689902927585003914562122604525965755095898421400738061436860606493020515205 12,
82291861031905330248839043767808139051115889538984981051680029826842960726395094207477635493045441904199270880068081676174372757941169718548361555773685054494217480510650093968033852966879010369622688118935396927873909308933622534854391377505871843183906788948632643847632101107092155290909739530772807 67]}

M_1 = {[57984032420551969261799367758338458391605776226353735000640893058982105780870199133014894375218943116262986366219380059075388129438371803015329960702473140891444769429782675874231332303873735676596897606086667046312916947663077810697335176924174483500340286083976630563630752740479561267046001864303413 29,
57984366919915958877596164047360541630997890999394155239167209259335038906739460743462255217588282568905728472476641071947270634052960105231530388975366150200041782691089956528231974678740987181628581132246124540355669925771359556210961802907864526269452673161142448227772563736155057038260321762437798 47],
[77311820227826632605201292818436555695879625119778033175170313278407518957247424437054951938712364519922153750787666070185929169508103535876306328754702181649528350398838714062974681488586813232108074457342584399055387278145249250587362485243105949487712123393858295642079796323522417841918455886981627 59,
77312154727190622220998089107458638935271739892818453413696629478760452083116686047502312781081703972564895857044927083057811674122691838092506757027595190958125363660145994716975323863454064737139757983502041893098140256253530996100989111226795992256824510471024113306221607319197913613132775785116012 77]}

M_2 = {[57984170496221909925877810386226547764811134524722693215300948882208401439276670926720160118256624161237329881376563468670909561148771741852185034089215268262842566619381210273068948460826408096660116800176592035056475284451191596894484229313007919500408554770592915278000281605391066736630158080825637 31,
57984247689251742312569887458420676769784668782296696557518504369627720031279813730215115319954375628980401003798131690965369756254228920031617527934789337924533396453808307847105106169545361918853168761653290600134316511772261875410219239102743988787623340439587748210925612364838399178971477799206928 64]}

M_3 = {[57984170496221909925877810386226547764811134524722693215300948882208401439276670926720160118256624161237329881376563468670909561148771741852185034089215268262842566619381210273068948460826408096660116800176592035056475284451191596894484229313007919500408554770
```

```
592915278000281605391066736630158080825637 31,
579841986953434583636091861551154053800737771207978586829240634634 0
098278842235493124245485012653596160858682491598590140613947951557 0
247462025839578562019690410899238854659158782927041878692202347023 5
404319671356111920298141585237128614648999347513291597529296490889 1
785979448079242750002250910679459915]}

M_989 = {[579841920589406700723097951151738986844836670449757912367
054444235659537243763227900123637103641595191709903054542208387271 4
097402039129691230441175707294279678482833255778893661222434370847 6
436268191586933957339224568664805896224765892815644236165962822188 0
783291029946017265476439229817208759766557315 8,
579841920589406700723097951151738986844836670449757912367054444235 6
595372437632279001236371036415951917099030545422083872714097402039 1
296912304411757072942796784828332557788936612224343708476436268191 5
869339573392245686648058962247658928156442361659628221880783291029 9
460172654764392298172087597665573158]}
```

C Details for the Example in Section 4.2.3

```
M_0 = {[54861240687936886832559362511872092700743926359323320701120
019884561973817596729471651756995363627936132847253378721117449581 8
386274464790322410371824567029961449870071000626453559019779193402 4
641512541262359795191593953928908168990292758500391456212260452596 5
755095898421400738061436860606493020515205 12,
677697679086279218947245099852534910872079054659841934767443944441 6
700497112138852761355004322157052863912480770614350546857645695410 0
998645465359226297705635822424041904024306403673018745370266392112 4
185508116914254528071272123444934981712974459137239125366250402987 0
620843711886340644011054725825822720]}
```

D Details for the Example in Section 5.3.3

```
n = 158497523913555876747567486455334446943697697850525957180228510
509398100752881436105780492049427802220707002137152229434111991799 0
918350753264367062123042487366644057367440411667898349892395924939 6
940576872415025116462794511904689281816894186939748682944275185968 4
594475821381902742945342241559212972696372131

c = 54861240687936886832559362511872092700743926359323320701120019 8
845619738175967294716517569953636279361328472533787211174495818 3862
744647903224103718245670299614498700710006264535421091908069935709 3
034032722424995315810616521937069048546621003595387222723579445410 5
739905153764964929088137271733758552820 8

m_0 = 55971299770253176123548884844479247656061065860293103011756 14
103689321006906137263446896798592525449650626338964215747425532842 3
629751864766375850357988020346895918445493180341753718928371971158 3
491646923356778275963421556719864874241556876714572640262541039381 1
528990759818194305695537320357475953232000

m = 123012498414353716588779615431871631339979562484880180664267029
764493711385683269223577800800348922301195090786285482141562878580 2
803337158796929850991240045150933647463768600998957967929564037887 7
173480819042865128755953205976806398381518022077914239362984786037 3
943806259389937543771569445743812385065245483
```

E The Seeds for our Attack Simulations

The structure of a seed is

$$x \,||\, y \,||\, z \,||\, 99 \,||\, n$$

where x represents the version of the algorithm, y represents the oracle type, z represents the test batch number and n represents the specific test number.

The version of the algorithm (x)
The original algorithm is specified by a 7, the Klíma et al. optimisation is specified by an 8, then the most efficient algorithm with both the Klíma et al. and Bardou et al. optimisations is specified by a 9. Testing with just the Bardou et al. optimisation came later, and as such, this is specified by a 6.

The oracle type (y)
A TTT oracle is denoted 000, a TFT oracle is denoted 010, an FTT oracle is denoted 100, and an FFT oracle is denoted 110.

The test batch number (z) and test number (n)
We decided to test in batches of 1000 to ensure that equal attention would have be paid to all variations of algorithm and oracle had we ran out of time. As a result, the first 1000 tests had a batch number of 0 and the test numbers ran from 1 to 1000. Then, the second 1000 tests had a batch number of 1 and again the test numbers ran from 1 to 1000. This continued until we had obtained results for 10000 tests under each permutation of oracle and algorithm. Of course we could have also simply ran the test numbers from 1 to 10000 for each permutation, but under the time constraints this could have provided incomplete results.

F An Optimised Implementation of the Algorithm

```
from __future__ import division
import gmpy2
from gmpy2 import *
import math
import random
from random import randint
import time
import numpy as np

# This is for a TTT oracle

def eea(a, b):
    if b == 0:
        return (1, 0)
    (q, r) = (a // b, a % b)
    (s, t) = eea(b, r)
    return (t, s - (q * t))

def find_inverse(x, y):
    inv = eea(x, y)[0]
    if inv < 1:
```

```
        inv += y
    return inv

def zeroinpaddingcheck(x):
    for j in range(3, 18):
        if j % 2 == 1:
            if hex(x)[j:(j + 2)] == "00":
                return 1
                break
            else:
                continue
        else:
            continue

def zeroinpadding(x):
    for j in range(0, (len(x) - 1)):
        if j % 2 == 0:
            if x[j:(j + 2)] == "00":
                return 1
                break
            else:
                continue
        else:
            continue

def delimiterchecker(x):
    a = len(hex(x))
    b = a - 2

    for j in range(19, (b + 1)):
        if j % 2 == 1:
            if hex(x)[j:(j + 2)] == "00":
                return 1
                break
            else:
                continue
        else:
            continue

def oracle(query):
    global counter
    counter += 1
    v = powmod(query, d, n)
    if bottom <= v and v <= top:
        return(1)
        # The padding is correct
    else:
        return(0)
        # we get an error message

def trimmer_oracle(query):
    global counter2
    counter2 += 1
    v = powmod(query, d, n)
    if bottom <= v and v <= top:
        return(1)
    else:
        return(0)
```

```
def test1(u, t):
    if gcd(u, t) == 1:
        return 1
    else:
        return 0

def test2(u, t):
    tinv = find_inverse(t, n)
    a = powmod(u, e, n)
    b = powmod(tinv, e, n)
    g = (c_0 * a * b) % n
    if trimmer_oracle(g) == 1:
        return 1
    else:
        return 0

def lcm(a):
    # https://stackoverflow.com/questions/37237954/calculate-the-
       lcm-of-a-list-of-given-numbers-in-python/37238140
    lcm = a[0]
    for i in a[1:]:
        lcm = int((lcm * i) / (gcd(lcm, i)))
    return lcm

def trimming():
    global trimmers
    global mintrim
    global maxtrim
    trimmers = [1]
    trimmersfrac = [1]
    trimmer_limit = 500
    for t in range(3, 4097):
        if counter2 < trimmer_limit:
            for u in range((t - 1), (t + 1) + 1):
                if counter2 < trimmer_limit and (u/t) > (2/3) and (
                   u/t) < (3/2):
                    if test1(u, t) == 1:
                        if test2(u, t) == 1:
                            trimmers.append(u / t)
                            trimmersfrac.append(t)
                            break
                        else:
                            continue
                    else:
                        continue
                else:
                    break
        else:
            break
    print ("It conducted {} trimmer queries to the oracle and found
        {} trimmers".format(counter2, (len(trimmers) - 1)))
    mintrim1 = min(min(trimmers), 1)
    maxtrim1 = max(max(trimmers), 1)
    newtrimmers = []
    newtrimmers.append(mintrim1)
    newtrimmers.append(maxtrim1)
    denom = lcm(trimmersfrac)
```

```
    lowerbottom = floordiv((2 * denom), 3)
    lowertop = denom
    while (lowertop - lowerbottom) != 1:
        u = ceildiv((lowerbottom + lowertop), 2)
        if test2(u, denom) == 1:
            lowertop = u
        else:
            lowerbottom = u
    ulower = lowertop
    mintrim = (ulower / denom)
    upperbottom = denom
    uppertop = ceildiv((3 * denom), 2)
    while (upperbottom + 1) != uppertop:
        u = floordiv((upperbottom + uppertop), 2)
        if test2(u, denom) == 1:
            upperbottom = u
        else:
            uppertop = u
    uupper = upperbottom
    maxtrim = (uupper / denom)

def range(start, stop):
    while start < stop:
        yield start
        start += 1

def range_overlap_adjust(list_ranges):
    # https://stackoverflow.com/questions/15273693/python-union-of-
       multiple-ranges
    overlap_corrected = []
    for start, stop in sorted(list_ranges):
        if overlap_corrected and start - 1 <= overlap_corrected
           [-1][1] and stop >= overlap_corrected[-1][1]:
            overlap_corrected[-1] = min(overlap_corrected[-1][0],
               start), stop
        elif overlap_corrected and start <= overlap_corrected
           [-1][1] and stop <= overlap_corrected[-1][1]:
            break
        else:
            overlap_corrected.append((start, stop))
    return overlap_corrected

def ceildiv(a, b):
    # http://stackoverflow.com/a/17511341
    return -(-a // b)

def floordiv(a, b):
    # http://stackoverflow.com/a/17511341
    return a // b

def primes():
    global p
    global q
    global e
    global d
    global k
    global n
    x = random.randint(int(pow(2, 511.5)), int(2 ** (512)))
```

```
    y = random.randint(int(pow(2, 511.5)), int(2 ** (512)))
    p = next_prime(x)
    q = next_prime(y)
    n = p * q
    phi = (p - 1) * (q - 1)
    byte_length_n = (len(hex(n)) - 2) / 2
    k = int(byte_length_n)
    e = 65537
    d = find_inverse(e, phi)

def PMS():
    global message
    a = pow(16, 95)
    b = (pow(16, 96)) - 1
    f = random.randint(a, b)
    message = hex(f)[2:-1]

def pad():
    global padding
    global encoding
    global decimal_of_encoding
    global ciphertext
    r = (k - 3 - ((len(message)) / 2)) * 2
    a = pow(16, (r - 1))
    b = pow(16, r) - 1
    f = random.randint(a, b)
    padding = hex(f)[2:-1]
    while zeroinpadding(padding) == 1:
        g = random.randint(a, b)
        padding = hex(g)[2:-1]
    encoding = "0002" + padding + "00" + message
    decimal_of_encoding = int(encoding, 16)
    ciphertext = powmod(decimal_of_encoding, e, n)

def step_1(input):
    global i
    global c_0
    i = 0
    global a
    global b
    for s in range(1, int(n)):
        w = int(powmod(s, e, n))
        binding = int((w * ciphertext) % n)
        if oracle(binding) == 1:
            list_s.append(s)
            break
        else:
            continue
    c_0 = binding
    M = [(2 * B, (3 * B) - 1)]
    trimming()
    a = int(ceil((2 * B) * (1 / mintrim)))
    b = int(floor(((3 * B) - 1) * (1 / maxtrim)))
    M = [(a, b)]
    list_M.append(M)
    i = i + 1

def step_2a():
```

```
    global i
    global c_0
    global s
    s = ceildiv((n + (2 * B)), b)
    found = False
    while not found:
        r = floordiv(((s * a) - (3 * B)), n)
        if s >= ceildiv(((2 * B) + ((r + 1) * n)), b):
            x = int(powmod(s, e, n))
            attempt2a = int((x * c_0) % n)
            if oracle(attempt2a) == 1:
                list_s.append(int(s))
                found = True
                break
            else:
                s = s + 1
                continue
        else:
            s = ceildiv(((2 * B) + ((r + 1) * n)), b)

def step_2b():
    global i
    global c_0
    global s
    if i > 1 and len(list_M[i - 1]) > 1:
        iteration = 0
        found = False
        r_values = []
        s_ranges = []
        for j in range(0, len(list_M[i - 1])):
            r_values.append(ceildiv(2 * (((list_M[i - 1][j][1] *
               list_s[i - 1]) - (2 * B))), n))
        for w in range(0, len(list_M[i - 1])):
            s_ranges.append((ceildiv(((2 * B) + (r_values[w] * n)),
                list_M[i - 1][(iteration % len(list_M[i - 1]))][1])
               , (ceildiv(((3 * B) + (r_values[w] * n)), list_M[i -
                1][(iteration % len(list_M[i - 1]))][0])))
        while not found:
            if s_ranges[(iteration % len(list_M[i - 1]))][0] >
               s_ranges[(iteration % len(list_M[i - 1]))][1]:
                h = (r_values[(iteration % len(list_M[i - 1]))]) +
                   1
                r_values[(iteration % len(list_M[i - 1]))] = h
                y = (ceildiv(((2 * B) + (h * n)), list_M[i - 1][(
                   iteration % len(list_M[i - 1]))][1]), (ceildiv
                   (((3 * B) + (h * n)), list_M[i - 1][(iteration %
                    len(list_M[i - 1]))][0])))
                s_ranges[(iteration % len(list_M[i - 1]))] = y
                s = s_ranges[(iteration % len(list_M[i - 1]))][0]
                z = int(powmod(s, e, n))
                attempt2bnew = int((z * c_0) % n)
                if oracle(attempt2bnew) == 1:
                    found = True
                    list_s.append(int(s))
                    break
                else:
                    t = s + 1
```

```
                    s_ranges[(iteration % len(list_M[i - 1]))] = (t
                        , ((ceildiv(((3 * B) + (r_values[(iteration
                        % len(list_M[i - 1]))] * n)), list_M[i -
                        1][(iteration % len(list_M[i - 1]))][0])))
                    iteration = iteration + 1
            else:
                s = s_ranges[(iteration % len(list_M[i - 1]))][0]
                z = int(powmod(s, e, n))
                attempt2bnew = int((z * c_0) % n)
                if oracle(attempt2bnew) == 1:
                    found = True
                    list_s.append(int(s))
                    break
                else:
                    t = s + 1
                    s_ranges[(iteration % len(list_M[i - 1]))] = (t
                        , ((ceildiv(((3 * B) + (r_values[(iteration
                        % len(list_M[i - 1]))] * n)), list_M[i -
                        1][(iteration % len(list_M[i - 1]))][0])))
                    iteration = iteration + 1

def step_2c():
    global i
    global c_0
    global s
    if i > 1 and len(list_M[i - 1]) == 1:
        found = False
        r = ceildiv(2 * (((list_M[i - 1][0][1] * list_s[i - 1]) -
            (2 * B))), n)
        while not found:
            for s in range(ceildiv(((2 * B) + (r * n)), list_M[i -
                1][0][1]), (ceildiv(((3 * B) + (r * n)), list_M[i -
                1][0][0]))):
                z = int(powmod(s, e, n))
                attempt2c = int((z * c_0) % n)
                if oracle(attempt2c) == 1:
                    found = True
                    list_s.append(int(s))
                    break
            r = r + 1
    else:
        print("error")

def step_3():
    global i
    global c_0
    list_temp = []
    for j in range(0, len(list_M[i - 1])):
        for r in range((ceildiv(((list_M[i - 1][j][0] * list_s[i])
            - (3 * B + 1)), n)), (floordiv((list_M[i - 1][j][1] *
            list_s[i] - 2 * B), n)) + 1):
            z = ceildiv((2 * B + r * n), list_s[i])
            list_temp.append(((int(max(list_M[i - 1][j][0], min(
                ceildiv((2 * B + r * n), list_s[i]), list_M[i - 1][j
                ][1])))), int(min(list_M[i - 1][j][1], max(floordiv
                (((3 * B - 1) + r * n), list_s[i]), list_M[i - 1][j
                ][0])))))
    list_M.append(range_overlap_adjust(list_temp))
```

```
def step_4():
    global i
    global c_0
    inverse = find_inverse(list_s[0], n)
    message = list_M[i][0][0] * inverse % n
    if message != decimal_of_encoding:
        print "Error - Decryption failed!"

def main(ciphertext):
    global counter
    global counter2
    global bottom
    global top
    global list_s
    global B
    global list_M
    global i
    B = int(pow(2, 8 * (k - 2)))
    bottom = 2 * B
    top = ((3 * B) - 1)
    counter = 0
    counter2 = 0
    list_s = []  # this is where we store our s values
    list_M = []  # this is where we store the possible intervals
    step_1(ciphertext)
    t0 = time.time()
    step_2a()
    step_3()
    while len(list_M[i]) != 1 or list_M[i][0][0] != list_M[i
       ][0][1]:
        i = i + 1
        if i > 1 and len(list_M[i - 1]) > 1:
            step_2b()
            if t not in list2b:
                list2b.append(t)
        elif i > 1 and len(list_M[i - 1]) == 1:
            step_2c()
        else:
            print("Error")
            break
        step_3()
    else:
        step_4()
    t1 = time.time()
    duration = t1 - t0
    print("It took {} seconds, and required {} calls to the Oracle
       ".format(duration, (counter + counter2)))
    print("It called the oracle for trimming {} times".format(
       counter2 - 500))

def test(x):
    global list2b
    Oracle_times = []
    trimmercount = []
    list2b = []
    u = 1
    for u in range(1, (x + 1)):
```

```
        global t
        t = int("90000" + "99" + str(u))
        random.seed(t)
        print ("Seed {}: {}".format(u, t))
        primes()
        PMS()
        pad()
        main(ciphertext)
        Oracle_times.append(counter + counter2)
        trimmercount.append(counter2 - 500)
        u = (u + 1)
        print("Mean: {}, Median: {}".format(np.mean(Oracle_times),
           np.median(Oracle_times)))
        print
    print("The list of Oracle query numbers is: {}".format(
       Oracle_times))
    print("{} seeds have entered step 2b".format(len(list2b)))
    print("The seeds that entered step 2b are {}".format(list2b))
    print("The list of oracle trimmer calls is {}".format(
       trimmercount))
    print("Maximum: {}, Mean: {}, Median: {}".format(max(
       trimmercount), np.mean(trimmercount), np.median(trimmercount
       )))

test(1000)
```