Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Experimenting with the Bleichenbacher Attack

Bachelor Thesis

Livia Capol

March 01, 2021

Advisor: Prof. Dr. Kenny Paterson

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

In 1998, Daniel Bleichenbacher introduced an adaptive chosen ciphertext attack against RSA-based encryption using the PKCS #1 v1.5 padding standard. The attack makes use of a padding oracle based on side-channel information. We describe the attack in detail, conduct a theoretical analysis and analyze the results of large-scale experiments. Hereby, we consider different oracle types and sizes of the RSA modulus to show how these parameters influence the attack.

Over the last twenty years, different optimization methods for the so-called "Million Message Attack" have been devised. We provide precise explanations of their underlying mathematical properties and describe how to implement them in practice. For the implementation, we introduce new adjustments to the methods to make them more efficient. Furthermore, we analyze their individual and combined effect on the attack complexity experimentally. The experimental results show that applying the different improvement methods together significantly lowers the number of oracle calls required to decrypt a ciphertext.

Finally, we introduce a novel method for improving a part of the algorithm to further enhance its complexity for some unfortunate cases. This new technique is also tested and its experimental results are discussed. From the simulations, it follows that this method allows us to increase the efficiency of Bleichenbacher's attack while only slightly lowering its overall success probability. To show the effect that the different optimizations have on the Bleichenbacher attack, all results are visualized and tables containing the relevant complexity results are provided. Lastly, we give a comprehensive overview of the influence of all tested improvements on the attack.

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Dr. Kenny Paterson, for his continuous support and guidance during this thesis. Our discussions were always very insightful and inspired me to dive deeper into the subject. Furthermore, I am grateful that you always made time to respond to all my questions and ideas.

Next, I would like to thank the whole Applied Cryptography Group at ETH for granting me access to their daisen server, making it possible to perform large-scale experiments.

Finally, a huge thank you to all my family and friends who supported me throughout writing this thesis and my whole Bachelor studies.

Contents

Chapter 1

Introduction

1.1 Motivation and Relevance

In 1998, Daniel Bleichenbacher published a paper that describes an adaptive chosen ciphertext attack against RSA-based encryption using the PKCS #1 v1.5 padding standard [3]. Chosen ciphertext attacks are usually based on the assumption that an attacker has access to a decryption device that returns the complete or part of the decryption for a chosen ciphertext. Hence, if a cryptographic system was vulnerable to a chosen ciphertext attack, this was only considered a theoretical weakness. However, the attack introduced by Bleichenbacher showed the practical relevance of this attack model. It uses a padding oracle based on side-channel information to gradually learn more details about an encrypted message. The PKCS #1 v1.5 encryption format, exploited by the attack, was widely deployed, including in a version of the Secure Sockets Layer (SSL) used by thousands of web servers at the time. Furthermore, the attack is also known as the "Million Message Attack" because it needs to send around one million ciphertexts to a decryption device, i.e., an SSL server, to successfully decrypt a ciphertext.

Since Bleichenbacher's publication, a great effort has been put into developing mitigation techniques to thwart the attack. However, these techniques often introduce new side-channels themselves due to careless implementations, making the application again vulnerable to the attack [8]. Additionally, slight variations of the vulnerabilities still exist in many modern servers [4, 10] and new Bleichenbacher-like attacks have been discovered recently exploiting different types of side-channel information [5, 16, 17].

Furthermore, various optimizations have been devised over the last twenty years to improve the attack presented by Bleichenbacher [2, 8]. Hence, for the famous "Million Message Attack", one nowadays requires significantly fewer chosen ciphertexts.

This thesis aims to provide a detailed and intuitive description of Bleichenbacher's attack. Additionally, we want to give a comprehensive overview of the different improvements published over the years. For this, we describe them carefully, as often some details are omitted in the papers, and then compare their performance to the original algorithm by performing large-scale experiments. The last objective is to conduct research into devising new methods for improving the algorithm. Hence, we introduce slight enhancements to the existing optimizations and present a new heuristic to lower the attack complexity for some unfortunate cases.

1.2 Thesis Structure

The thesis starts by introducing the concepts relevant to Bleichenbacher's attack. Hence, Chapter 2 provides an overview of RSA encryption, the Public Key Cryptography Standards (PKCS), padding oracles and SSL/TLS. Chapter 3 then presents the Bleichenbacher attack. We first describe the attack in detail, followed by a theoretical analysis and conclude the chapter by explaining the complexity results of the attack in practice. In Chapter 4, the different improvements are presented and their effect on the attack is analyzed. Based on the performed experiments, we derived a novel heuristic to further reduce the attack complexity for some unfortunate cases. This heuristic is shown in Chapter 5 alongside its results in practice. Finally, Chapter 6 gives an overview of the experimental results obtained from large-scale experiments when applying the different optimizations. The thesis finishes with a conclusion on the conducted research on Bleichenbacher's attack. Furthermore, in the appendix, we provide the code of our implementation of the Bleichenbacher attack containing all improvements and the heuristic.

Chapter 2

Preliminaries and Notation

2.1 RSA Encryption

Until the 1970s, only symmetric cryptographic systems existed. In these methods, two parties that want to communicate securely have to share a secret key. This key is used to both encrypt and decrypt the exchanged messages and must not be known by any other party. In 1978, Rivest, Shamir and Adleman introduced the RSA algorithm [15]. This algorithm presented a novel way of encrypting and decrypting messages, which is nowadays widely known as asymmetric encryption. In asymmetric encryption, there exist two different keys for encryption and decryption. One of them is the so-called public key which is known publicly and can be given to anyone. This key is used for encryption. The decryption key is called private key and must be kept secret. A message is encrypted using the public key and can only be decrypted with the corresponding private key. Additionally, it is hard to compute the private key from the public key. In RSA, this is based on the fact that prime factorization is hard. More precisely, for a large number n whose factors are two large prime numbers p and q it is hard to find those primes. The larger the value n is, the more difficult it is to find the corresponding prime factors.

Generating a public and private key pair using the RSA algorithm is done in the following steps:

1. Generate two different large random prime numbers p and q. These values are kept secret.
2. Calculate $n = pq$. This value n is called the modulus for this key pair and will be shared in the public key. However, because it is hard to factorize n, the prime numbers remain unknown to the public. The larger the random prime numbers p and q are, the harder it is to factorize n.
3. Calculate the totient $\phi(n) = (p-1)(q-1)$.

4. Choose an integer e that is co-prime to $\phi(n)$ and satisfies $1 < e < \phi(n)$. This value e will be shared to be used for encryption.
5. Compute the multiplicative inverse of e in the ring of integers modulo $\phi(n)$. In other words, we want to find the integer d for decryption that satisfies: $e \cdot d \equiv 1 \pmod{\phi(n)}$. It will be a part of our private key and should be hidden.

After this procedure, we can construct the private and public keys of this RSA instance. We set the public key to (n, e) and the private key to (p, q, d). All parts of the public key have to be shared and all parts constituting the private key have to remain secret.

As soon as the public and private keys have been set up, an integer message m smaller than n can be encrypted by computing $c \equiv m^e \pmod{n}$. The resulting ciphertext c can be decrypted by the party holding the private key in the following way: $m \equiv c^d \pmod{n} \equiv m^{ed} \pmod{n} \equiv m \pmod{n}$. The last step follows from the Euler-Fermat theorem.

In practice, this implementation of RSA is usually referred to as textbook RSA and well-known to be insecure. For example, if we encrypt a single message twice using the same RSA public key, the encryption will result in two identical ciphertexts. This means that the textbook RSA algorithm is a deterministic encryption scheme and therefore not IND-CPA secure [7]. Hence, RSA must be used in combination with a padding scheme to meet standard security notions. In the next section, we will present an encoding scheme created to achieve this objective. Using a padding scheme also directly allows us to split a message m which is larger than n into a sequence of messages smaller than n and encrypt each one separately with RSA.

2.2 Public Key Cryptography Standards

The Public Key Cryptography Standards (PKCS) are a set of standards devised and introduced by the RSA Laboratories. They provide specifications for implementing RSA-based public key cryptography with the aim to eliminate the security issues of textbook RSA. PKCS #1, the first version of the standards, was published in the early 1990s. We will focus on PKCS #1 v1.5 [11] as Bleichenbacher's attack [3] exploits the structure of this padding scheme.

2.2.1 Public Key Cryptography Standards #1 v1.5

We describe PKCS #1 v1.5 by showing how a message M gets transformed into a ciphertext c by the sender and how the receiver can extract the message from an incoming ciphertext. For this, assume that we have an RSA public key (n, e) as well as a private key (p, q, d). The message M the sender wants to exchange is a sequence of bits and consists of $|M|$ bytes. The PKCS #1 v1.5 standard specifies how this message gets padded to encrypt it using the public key. Let k be the byte length of the modulus n. The data block M to be encrypted can have at most $k - 11$ bytes. If the message M consists of more bytes, the sender can split it into a sequence of smaller messages and encrypt each message separately. To pad the message, the sender first generates a pseudo-random padding string PS of $k - 3 - |M|$ nonzero bytes. The constraint that M can consist of at most $k - 11$ bytes means that the padding has to consist of at least 8 bytes. Now, the sender can build an encryption block EB of exactly k bytes in the following way:

$$EB \texttt{ = 0x00 || 0x02 || PS || 0x00 || M.}$$

Here, `||` denotes that two bytes or sequences of bytes are concatenated. The second byte of the encryption block EB is called the block type. There exist three different block types: 0, 1 and 2. For Bleichenbacher's attack, only block type 2 is relevant as it is used for encryption. The other two block types are reserved for digital signatures.

Next, the sender converts the encryption block EB into an integer m. This integer can then be encrypted with RSA, resulting in the ciphertext $c \equiv m^e \pmod{n}$.

The receiver, who owns the private key, can easily extract M from the ciphertext c. First, the receiver deciphers c with his RSA private key exponent d. As a result, he obtains the message m and converts it into an encryption block EB. In this block, he searches for the first zero byte after the initial two bytes, indicating the end of the padding. Having identified this byte, the receiver can directly extract the message.

This padding scheme is nowadays widely deployed in practice. Although it was created to provide a secure way to implement RSA-based encryption, this encoding scheme is known to introduce security problems. In 1998, Daniel Bleichenbacher published a paper [3] that shows a now well-known attack that exploits the padding structure combined with side-channel information to decrypt an arbitrary ciphertext. Bleichenbacher's attack is based on an oracle that returns whether a given ciphertext corresponds to a correctly padded plaintext according to the PKCS #1 v 1.5 standard. In the next section, we introduce different padding oracles for PKCS #1 v 1.5 depending on how restrictive the checks of the oracle on the plaintext are.

2.2.2 Padding Oracles based on PKCS #1 v1.5

A padding oracle is an oracle that receives a ciphertext and returns whether the corresponding plaintext has the correct format or not. Hereby, the definition of correct depends on the used padding scheme for the cryptographic protocol and how rigorously the oracle checks the specification. In our case, the oracles are based on the PKCS #1 v1.5. In order to capture the behavior of real devices, where some receivers perform more careful checks than others, we consider stronger and weaker oracles. Overall, we describe four different oracles. Each oracle type always checks that the first two bytes of the plaintext are `0x00 || 0x02`. As we will see, this allows Bleichenbacher's attack to be used with all four oracle types to decrypt an arbitrary ciphertext. However, the complexity of deciphering a ciphertext depends on how restrictive the remaining oracle checks are.

We adopt the convention of characterizing the different oracles with three Booleans [2]. Each of the three Booleans corresponds to one test the oracle applies or skips on a decrypted ciphertext. F denotes that the test is conducted and T that the test is skipped. We also keep the same order of the Booleans for the different checks as presented by Bardou et al. [2]. The first Boolean stands for testing the existence of at least one zero byte after the first ten bytes. This property must hold for any plaintext that conforms to the PKCS #1 v1.5 standard as there is one zero byte directly after the padding string. The second Boolean stands for checking the existence of eight nonzero bytes in the padding. This check is necessary because the padding has at least eight bytes and can only contain nonzero bytes. Finally, the third Boolean corresponds to assuring that the message contained in the plaintext is of some particular length. For our analysis, we consider the four following oracles: TTT, TFT, FFT and FFF. This set of oracle types is sensible since we assume that in practice, a decrypted ciphertext is parsed from left to right, performing increasingly more checks. We note that for the FFF oracle, we will analyze a particular instance that is of practical relevance. This oracle is referred to as "Bad Version Oracle" and described in Section 2.3.3.

Let us now describe the four different oracle types:

- TTT Oracle: The TTT oracle is the most permissive oracle as it only checks that the first two bytes of the plaintext are `0x00` and `0x02`. The oracle does not perform any other checks on the plaintext for a given ciphertext.
- TFT Oracle: The TFT oracle checks for the `0x00` and `0x02` bytes and the presence of at least eight bytes of non-zero padding following the two initial bytes. It returns `True` on any plaintext with these two properties and does not test for the existence of a zero byte in the remaining $k - 10$ bytes.
- FFT Oracle: The FFT oracle returns `True` on a correctly padded plaintext of any length. It accepts any ciphertext that when deciphered results in a plaintext conforming to the PKCS #1 v1.5 format without requiring the `0x00` byte to be at a specific location.

- FFF Oracle: The FFF oracle is the most restrictive oracle and only accepts correctly padded plaintexts with a fixed message length. This means that we know exactly how many bytes the padding block has and at which position in the plaintext the zero byte has to be located.

2.2.3 Access to a Padding Oracle

Bleichenbacher's attack relies on the access to an oracle that returns `True` on any ciphertext whose corresponding plaintext conforms to the PKCS #1 v1.5 padding format [3]. It is important to note that the assumption of such an oracle also has practical relevance. In practice, an oracle can be in the form of error messages [4, 8] resulting from incorrectly padded plaintexts. However, this is not the only side-channel information that can serve as an oracle. Since its publication, various Bleichenbacher-like attacks have been discovered, also exploiting other side-channel information such as timing variations [5, 10], memory access patterns [17] and microarchitectural side-channels [16].

We also want to describe the behavior which results in the different oracle types in practice. A TTT oracle arises if some server only checks the first two bytes and if they are correct, directly extracts a message of known length. The same holds for a TFT oracle, which also checks for eight nonzero bytes following the first two bytes. As these oracles do not check for a zero byte after the padding string, they must have foreknowledge about the length of the message to extract it. On the other hand, the FFT oracle arises for a server that checks all properties of a PKCS #1 v1.5 conforming plaintext. It extracts a message of unknown length after the first discovered zero byte in the last $k - 10$ bytes. Finally, we obtain a FFF oracle if a server checks all specifications, including the zero byte at a fixed position, and extracts the message of known length.

2.3 SSL/TLS

2.3.1 Description

Transport Layer Security (TLS), often still referred to as Secure Sockets Layer (SSL), is a cryptographic protocol whose aim is to provide privacy and data integrity for applications communicating over a computer network [14]. The protocol is widely established nowadays and used for different applications such as banking or email. Its main goal is to set up a secure channel between a client and a server. TLS consists of several subprotocols, the two most important being the TLS Record Protocol and the TLS Handshake Protocol. The Handshake Protocol initiates the connection by establishing keys to communicate securely and making some assurances about the identities of the communicating parties. The Record Protocol concerns the exchange of application data between the entities. It describes how data is compressed, authenticated and encrypted. It is important to note that TLS is the successor and improved version of SSL. Nevertheless, the two terms are still used interchangeably in the industry. There also exist different versions of TLS, the most recent being TLS 1.3 [13] published in 2018 with several new features providing more security and reduced latency compared to TLS 1.2. In the remaining part of this section, we consider TLS 1.2 as this is the version whose structure Bleichenbacher's attack [3] exploits and the most widely used version to date. However, also the earlier version SSLv2 is highly vulnerable to a variation of the attack that enables decryption of RSA ciphertexts [1].

2.3.2 TLS 1.2 Handshake Protocol

We now take a closer look at the Handshake Protocol of TLS 1.2. Bleichenbacher's attack can be used to decipher one of the exchanged messages in this protocol. Figure 2.1 shows the transmitted messages between a client C and a server S to set up a secure channel [12]. The messages marked with an asterisk are optional or situation-dependent and the messages enclosed in square brackets are encrypted with the keys generated during the protocol run. First, we look at the different steps on a high level and then break down one particular message relevant to Bleichenbacher's attack. We consider the case where RSA is used for key agreement and authentication in the TLS 1.2 Handshake Protocol. Furthermore, the TLS 1.2 protocol uses the PKCS #1 v1.5 standard for RSA-based sessions. Both are requirements for Bleichenbacher's attack. We note that other key agreement modes exist for the TLS 1.2 Handshake Protocol, such as Diffie-Hellman. In TLS 1.3, the Diffie-Hellman key exchange protocol is used exclusively because of its better security guarantees, providing perfect forward secrecy.

The first two messages `ClientHello` and `ServerHello` are used to agree on the cipher suite and exchange nonce values for later key derivation. The server usually also sends a `Certificate`, which binds his public key to his identity, to authenticate itself. Optionally, the server can require the client to provide a `Certificate`, but this is rarely done in practice. If RSA is used for key agreement and authentication, the client then generates a so-called `premaster secret` consisting of 48 pseudo-random bytes. This `premaster secret` is padded, encrypted under the server's public key from the obtained certificate and sent as the `ClientKeyExchange` message. The server can decrypt this message and obtain the `premaster secret` from which both communicating parties derive a `master secret`. This `master secret`, in turn, is used to derive the keys for encryption and authentication of this TLS session. In the `Finished` messages, a MAC comprises the hash of all previously exchanges messages to guarantee the integrity of the handshake. The client and server check these messages. To establish a TLS session, both

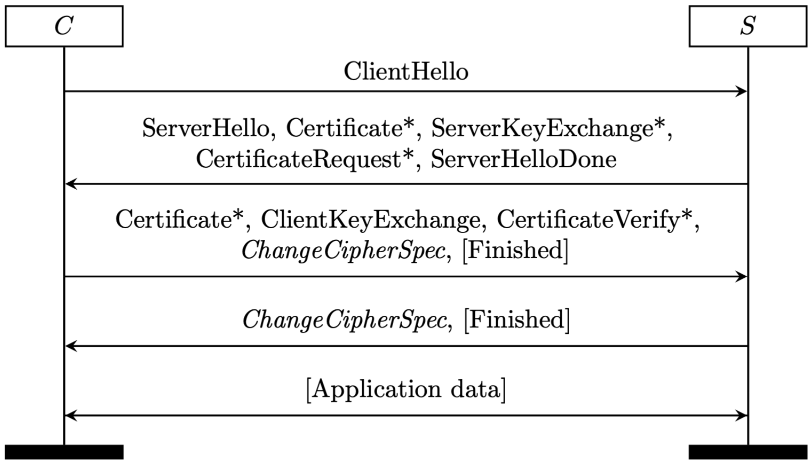

C
S
ClientHello
ServerHello, Certificate*, ServerKeyExchange*, CertificateRequest*, ServerHelloDone
Certificate*, ClientKeyExchange, CertificateVerify*, *ChangeCipherSpec*, [Finished]
ChangeCipherSpec, [Finished]
[Application data]

Figure 2.1: TLS 1.2 Handshake Protocol [12]

of the `Finished` messages have to be successfully verified. Afterwards, the negotiated keys can be used for encryption and authentication of application data.

Bleichenbacher's attack concerns the `ClientKeyExchange` message of the TLS 1.2 Handshake Protocol. If an attacker can decrypt this message, he obtains the `premaster secret` and can derive the session keys, breaking all security guarantees. As the protocol uses PKCS #1 v1.5 for padding the message, we know that the structure of a `ClientKeyExchange` message looks as follows:

```
ClientKeyExchange = 0x00 || 0x02 || PS || 0x00 || premaster secret.
```

The `premaster secret` has a fixed length of 48 bytes, where the first two bytes specify the version number of TLS. For TLS 1.2, these bytes (also referred to as `major` and `minor` version numbers) are both `0x03`. The remaining bytes are randomly generated by the client and build a pseudo-random string R of 46 bytes. So, the `premaster secret` has the following structure:

```
premaster secret = major || minor || R.
```

If we have access to an oracle that gives us information about the padding of a decrypted ciphertext, this oracle can be used in Bleichenbacher's attack. In the next section, we introduce an oracle relevant in practice: the Bad Version Oracle (BVO). It is based on side-channel information revealed due to poor implementation of the checks on the padding structure.

2.3.3 Bad Version Oracle (BVO)

Bleichenbacher's attack exploits side-channel information about the correct format of a plaintext to decrypt an arbitrary ciphertext [3]. As a countermeasure to his attack, it was recommended not to reveal any information about the decoding process and where a potential error occurs. Additionally, security architects were advised to carefully verify all specifications of the padding structure relevant for TLS 1.2. In particular, they were instructed to not only check the correctness of the first two bytes, the nonzero padding string of prescribed length and the zero byte at a fixed position but also that the version numbers of the `ClientKeyExchange` message are correct. These version numbers were introduced to thwart so-called version rollback attacks. As it was not adequately described how these checks should be done, many architects simply issued an error message whenever the version numbers were incorrect. The side-channel information provided by this uncareful implementation led to a new possible oracle referred to as the Bad Version Oracle (BVO). The attack based on this oracle was discovered and published in 2003 by Klíma, Pokorný and Rosa [8].

According to our description in Section 2.2.2, an FFF oracle only accepts plaintexts that conform to the PKCS #1 v1.5 standard and where the message M has a fixed length. The BVO, introduced by Klíma, Pokorný and Rosa, describes a particular version of a FFF oracle. We now describe the properties of the oracle.

First, recall that a plaintext is PKCS #1 v1.5 conforming if:

- The first byte of the plaintext is `0x00`.
- The second byte of the plaintext is `0x02`.
- The next eight bytes of the plaintext are not `0x00`.
- There exists a `0x00` byte in the remaining k - 10 bytes.

In the TLS 1.2 Handshake Protocol, the message, i.e., the `premaster secret`, has a fixed length of 48 bytes. The oracle hence only accepts a plaintext if the length of the message M is exactly 48 bytes.

We call such a plaintext S-PKCS conforming. So, a plaintext is S-PKCS conforming if:

- The first byte of the plaintext is `0x00`.
- The second byte of the plaintext is `0x02`.
- The next $k - 51$ bytes of the plaintext are not `0x00`.
- The byte $k - 48$ is `0x00`.

To describe which plaintexts are accepted by the BVO, we first have to introduce the servers' behavior that results in the side-channel information. It is known that the server should not disclose any information whether the decrypted ciphertext is S-PKCS conforming or not. Thus, in practice, a server is recommended to continue with a randomly chosen value of the `premaster secret` if the resulting plaintext has the wrong format. The communication will still break down after sending the `Finished` message because the client and server will use different session keys due to this countermeasure. However, the attacker does not know whether the communication broke down because he sent an invalid ciphertext or the incorrect `premaster secret`.

We assume that the server applies all checks to assure that the plaintext is S-PKCS conforming and the countermeasure to thwart Bleichenbacher's attack. Additionally, all S-PKCS conforming plaintexts are processed by the server to check for the correct version number. This results in the following behavior of the server:

1. The server checks if the decrypted plaintext is S-PKCS conforming. If the plaintext is not S-PKCS conforming, the server generates a new `premaster secret` randomly and breaks down the communication with the client after receiving the `Finished` message.
2. The server checks each S-PKCS conforming plaintext to see whether the bytes $k-47$ and $k-46$ contain the expected `major` and `minor` version numbers. If the test fails, the server issues a distinguishable error message. Note that the test is never done for plaintexts that are not S-PKCS conforming.

We can now give the definition of the Bad Version Oracle. The BVO accepts any ciphertext if:

- The corresponding plaintext is S-PKCS conforming.
- Either byte $k-47$ of the plaintext is not equal to the `major` version number or byte $k-46$ is not equal to the `minor` version number.

We can see that the probability of an S-PKCS conforming plaintext also being accepted by the oracle is high. A random S-PKCS conforming plaintext passes the oracle with probability $1-2^{-16}$ because the probability of both version numbers being correct is 2^{-16}.

It is important to note that receiving an error message in practice is interpreted as a `True` from the BVO. Additionally, if this oracle returns `True`, we know with certainty that the deciphered ciphertext is S-PKCS conforming and hence also PKCS #1 v1.5 conforming. This knowledge implies that we can apply Bleichenbacher's attack with this oracle. Obviously, the more restrictive checks of this oracle significantly influence the complexity of the attack, which becomes evident in the theoretical analysis of the attack and the experimental results.

Chapter 3

Bleichenbacher's Attack

Bleichenbacher's paper describes an adaptive chosen ciphertext attack against RSA-based encryption using the PKCS #1 v1.5 padding standard [3]. A chosen-ciphertext attack is an attack model where the attacker can select a ciphertext and obtain the complete or part of its decryption. In the case of Bleichenbacher's devised algorithm, the attacker has access to an oracle that tells him, for any chosen ciphertext c, whether the corresponding plaintext has the correct format. An adaptive chosen ciphertext attack means that the attacker can choose the ciphertexts based on previous outcomes of the oracle.

3.1 Description

First, we give an overview of the attack and then describe each step in detail. Let (n, e) be the RSA public key and (p, q, d) the corresponding private key. Assume the attacker does not have access to the private key and wants to find $m \equiv c^d \pmod{n}$ for an arbitrary integer c. He chooses integers s, computes $c' \equiv cs^e \pmod{n}$ and sends c' to the oracle. Let $B = 2^{8(k-2)}$, where k is the byte length of the modulus n. If the oracle returns `True` for c', the attacker knows that the first two bytes of the plaintext, i.e., $ms \pmod{n}$, are `0x00` and `0x02`. Hence, the attacker can derive that $2B \leq ms \pmod{n} \leq 3B - 1$. By testing different values for s, the attacker collects several such pieces of information restricting the possible values of m further until he can derive m.

The attack is divided into three phases. In the first phase, the message is blinded, resulting in a ciphertext c_0 corresponding to an unknown message m_0. This step is only needed for computing signatures where the message m does not conform to the proper padding. For our consideration of decrypting an encrypted message, this phase can be skipped as the initial message already has the correct format. In the second phase, the attacker tries to find small values s_i for which the oracle accepts $c_0(s_i)^e \pmod n$. For each successful value s_i, he narrows down the range of possible values for m_0 by computing a set of intervals where one of them must contain m_0. To do this, he uses the previously collected knowledge about m_0. We stay in the second phase until only a single interval is left. As soon as this happens, we continue with the last phase. The attacker now has enough information about m_0 to choose the s_i such that the probability of $c_0(s_i)^e \pmod n$ being accepted by the oracle is a lot higher than for a randomly chosen ciphertext. During this procedure, the size of s_i is increased gradually to restrict the possible range of m_0 further until he ends up with only one possible value.

Now, we describe the different phases of the attack in detail.

Step 1: Blinding. Given an integer c, choose different random integers s_0 until $\mathrm{c}(s_0)^e \pmod n$ is accepted by the oracle. For the first successful value s_0, set $c_0 \leftarrow c(s_0)^e \pmod n$, $M_0 \leftarrow \{[2B, 3B-1]\}$ and $i \leftarrow 1$. When decrypting a ciphertext c corresponding to a correctly padded message m, set $s_0 = 1$ and $c_0 = c$.

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 oracle accepts the ciphertext $c_0(s_1)^e$ (mod n). This means start by trying the value $s_1 = \lceil n/(3B) \rceil$ and increment it by one until the ciphertext is accepted by the oracle.

Step 2b: Searching with more than one interval left. If $i > 1$ and the number of intervals in M_{i-1} is at least 2, search for the smallest integer $s_i > s_{i-1}$, such that the ciphertext $c_0(s_i)^e \pmod n$ is accepted by the oracle. Similarly to step 2a, start with $s_i = s_{i-1} + 1$ and increment s_i until the ciphertext is accepted by the oracle.

Step 2c: Searching with one interval left. If $i > 1$ and M_{i-1} contains exactly one interval (i.e., $M_{i-1} = \{[a, b]\}$), choose small integer values r_i, s_i such that

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

and

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

until the ciphertext $c_0(s_i)^e \pmod n$ is accepted by the oracle.

In other words, start with $r_i = \lceil 2\frac{bs_{i-1}-2B}{n} \rceil$. For this r_i, compute the possible s_i values and test them with the oracle, starting with the lowest. If the interval of possible s_i values for this r_i is empty or none of the tested values work, increment r_i by one and continue searching.

Step 3: Narrowing the set of solutions. For each successful s_i, compute the set M_i by setting:

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

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

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

Before showing the correctness and analyzing the complexity of this attack, we make four remarks concerning the algorithm:

- Bleichenbacher's attack is based on the fact that we want to find a message m_0, which we know is contained in the interval $[2B, 3B - 1]$. This is why blinding is required if we are not sure that the message m conforms to the PKCS #1 v1.5 standard.
- We start step 2a with $s_1 = \lceil n/(3B) \rceil$ as for smaller values of s_1 the first two bytes of the message $m_0 s_1 \pmod n$ are never `0x00 || 0x02`. This follows because $m_0 \in [2B, 3B - 1]$ and hence for $1 < s_1 < n/(3B)$ we have $3B < m_0 s_1 \pmod n < n$.
- We want to find the smallest possible value for s_1 in step 2a as with increasing s_1, it gets more likely that we have to perform step 2b. The larger s_1, the more r values are possible in step 3. For each r value, we then introduce an interval in M_1. These intervals do not overlap as their length is upper bounded by $\lceil \frac{B}{s_1} \rceil$ and two consecutive intervals are $\lceil \frac{n}{s_1} \rceil$ apart from each other. Note that for a known value n, we could actually determine the value s_{lim} such that if $s_1 > s_{lim}$, we have to perform step 2b.
- For step 2c, we use $r_i \geq 2\frac{bs_{i-1} - 2B}{n}$ because we want to at least double the value of s_i with every round of 2c. By doing that, we can roughly half the length of the remaining interval, computed in step 3, in each iteration. Hence, we want $2s_{i-1} \leq s_i$. Combined with the constraint that $\frac{2B + r_i n}{b} \leq s_i$, we get $2s_{i-1} \leq \frac{2B + r_i n}{b}$. This leads to $r_i \geq 2\frac{bs_{i-1} - B}{n}$. We see that a slightly higher lower bound on r_i can be used compared to that in Bleichenbacher's original presentation. However, this is not relevant in practice as step 2c is already a heuristic procedure.

This attack works for any oracle that returns `True` on messages where the first two bytes are `0x00 || 0x02`. Hence, it can be used in combination with all of the previously introduced oracle types.

We now prove the correctness of the attack. To show that Bleichenbacher's attack finds the desired message m, it suffices to show that the attack finds m_0. We prove this by showing that $m_0 \in M_i$ for all i by induction over i. By construction, m_0 is accepted by the oracle and hence $2B \leq m_0 \leq 3B - 1$. This shows that $m_0 \in M_0$. Now, assume that $m_0 \in M_{i-1}$. We want to show that $m_0 \in M_i$. We know that there exists an interval $[a, b] \in M_{i-1}$ such that $a \leq m_0 \leq b$. Additionally, since $m_0 s_i \pmod{n}$ is accepted by the oracle, there exists an integer r such that $2B \leq m_0 s_i - rn \leq 3B - 1$. Combining these two insights, we get the following two equations:

$$as_i - (3B - 1) \leq rn \leq bs_i - 2B$$

and

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

Because of how the set of intervals in step 3 is updated, we know that there exists some interval $[a', b'] \in M_i$ containing m_0.

This proves that Bleichenbacher's attack finds the desired message m for an arbitrary ciphertext c. The complexity analysis of the attack is performed in Section 3.3 separately for each oracle type as the more or less restrictive checks on the decrypted ciphertexts influence its performance.

3.2 Noisy Oracles

Before we continue with the theoretical analysis of the attack, it is important to note that Bleichenbacher's attack [3] assumes a noiseless oracle. The attack relies on the correctness of the response it receives from the oracle. In practice, however, these oracles base on side-channel information, which is often noisy. An example of such an oracle is remotely timing the decryption process. But, noisy oracles are also relevant for more recently introduced attacks based on the leakage from microarchitectural side-channels [16].

Bleichenbacher's attack actually does not require the oracle to be completely noiseless. The attack still works in the presence of false negative returns by the oracle. In this context, a false negative means that the oracle rejects a valid ciphertext. The attack would continue searching for the next conforming multiplier s_i until the oracle eventually accepts a ciphertext. Although the correctness of the attack is not affected, its performance can be influenced a lot by this behavior. On the other hand, if the oracle returned false positives, Bleichenbacher's attack could fail. An oracle that returns `True` on an invalid ciphertext results in the attack narrowing the possible solutions down in an incorrect way.

The current solution to this problem is to repeat the timing measurements to reduce the false positive probability. Hereby, false negatives get ignored as they do not influence the correctness of the algorithm. This countermeasure clearly increases the complexity of the attack. In [16], Ronen et al. issue at most six queries with the same ciphertext. The ciphertext only gets accepted if at least five of the issued queries return `True`. Their simulation results show that the presence of errors at most doubles the number of queries required for the attack.

As noisy oracles are a challenge and of importance in practice, it would be of great interest to discover a version of Bleichenbacher's attack that copes with an oracle returning noisy results.

3.3 Theoretical Analysis

For the four different oracle types introduced in Section 2.2.2, we analyze the complexity of Bleichenbacher's attack separately. In this context, the complexity always denotes the number of oracle calls needed to complete the attack successfully. The theoretical analyses of the attack are conducted analogously to Bleichenbacher's paper [3].

Let (n, e) be the RSA public key and (p, q, d) the corresponding private key. Our analysis considers different lengths for the modulus n: 512, 1024 or 2048 bits. In practice, a modulus n is usually generated to have one of these lengths and we assume that n has exactly the specified number of bits. Depending on the generating algorithm, the bit length of the modulus can also be one bit more or less than the desired size. For simplicity, we exclude this fact from the theoretical analysis. Let k be the byte length of n. From our assumption it follows that: $2^{8k-1} < n < 2^{8k}$. We note that this is a tighter bound compared to Bleichenbacher's paper, where he lower bounds n by $2^{8(k-1)}$.

For the analyses of all oracles, we define two probabilities:

- Let $\Pr(P)$ be the probability that a randomly chosen integer $0 \leq m < n$ passes all checks conducted by the oracle. This describes the probability that a randomly chosen ciphertext c is accepted by the oracle.

- Let $\Pr(A)$ be the probability that the first two bytes of the random plaintext are `0x00 || 0x02`. In other words, $\Pr(A)$ is the probability that the first two bytes of the plaintext corresponding to a randomly chosen ciphertext c are `0x00 || 0x02`. $\Pr(A)$ can be described by $\Pr(A) = B/n$ for $B = 2^{8(k-2)}$. From our assumption on the modulus n it follows that: $2^{15}B < n < 2^{16}B$. Finally, we obtain: $2^{-16} < \Pr(A) < 2^{-15}$. This holds independently of the oracle type.

3.3.1 TTT Oracle

The TTT oracle is the most permissive version of the oracles. For a received ciphertext c, it only checks whether the first two bytes of the corresponding plaintext m are `0x00 || 0x02`. Intuitively, it should be the easiest for an attacker to randomly produce a ciphertext accepted by the oracle and thus find s_i values that work. This observation already suggests that if a device only checks the first two bytes of a plaintext, it is the most vulnerable to the Bleichenbacher attack. To approximate how many ciphertexts are needed to find the desired plaintext m, we conduct a complexity analysis of the attack under the TTT oracle.

For the TTT oracle, $\Pr(P)$ and $\Pr(A)$ are equal. We will see that this is not the case for the oracle types which perform more rigorous checks. Furthermore, we analyze the expected complexity of the Bleichenbacher attack for keylengths of 512, 1024 and 2048 bits to see how this parameter influences the attack performance.

keylength = 512 bits

We consider the attack on an encrypted message, which is already a valid ciphertext. Hence, step 1 is not necessary and we do not incorporate it into the theoretical analysis.

In step 2a, we want to find a small value s_1 that results in a valid ciphertext, but we do not have any additional information about m_0. Thus, we start at the first value that could result in a valid ciphertext and increase it by one until we find a conforming multiplier for s_1. To approximate the complexity of step 2a, we use $\Pr(P)$. The first multiplier for which the ciphertext is accepted by the oracle has roughly the size $1/\Pr(P) = n/B < 2^{16}$. Hence, we approximate the complexity of step 2a by 2^{16}.

For one round of step 2b, we again need to perform roughly $1/\Pr(P)$ calls to the oracle. However, the complexity of step 2b depends on how many times we have to repeat this step until we end up with a single interval. So, we want to bound the number of intervals in M_i for each value i. For this goal, Bleichenbacher used the following heuristic in his paper [3]: Let ω_i denote the number of intervals in M_i, then

$$\omega_i \leq 1 + 2^{i-1} s_i (\frac{B}{n})^i.$$

Before applying this formula to bound the number of intervals in M_i, we want to explain the heuristic. We start by looking at M_1. After having found s_1, we go to step 3 to narrow down the solution. For s_1, there exist at most $\lceil \frac{(b-a)s_1+B-1}{n} \rceil$ possible r values. As we are in the first iteration, $a = 2B$ and $b = 3B - 1$. So, we approximate the number of different values for r by

$\lceil \frac{Bs_1}{n} \rceil$. For each possible r, we introduce an interval $I_r = [\lceil \frac{2B+rn}{s_1} \rceil, \lfloor \frac{3B-1+rn}{s_1} \rfloor]$ in M_1. Because these intervals do not overlap, we expect M_1 to contain $\lceil \frac{Bs_1}{n} \rceil$ intervals, meaning that $\omega_1 \leq 1 + s_1(\frac{B}{n})$.

To show that the bound also holds for $i > 1$, assume that it holds for $i-1$. For each $[a', b'] \in M_{i-1}$, the possible number of r values can be approximated by $\lceil \frac{(b'-a')s_i}{n} \rceil$. Altogether, the number of intervals I_r can be bounded by $\lceil \frac{Bs_i}{n} \rceil$. Furthermore, because of how we construct the intervals, each interval I_r or part of it is included in M_i if it overlaps with the corresponding interval of M_{i-1}. No interval can overlap with two intervals in M_{i-1}. To provide a heuristic, assume that the intervals I_r are randomly distributed. The probability that one interval I_r, whose length is $\lceil \frac{B}{s_i} \rceil$, intersects with an interval in M_{i-1} of length $\lceil \frac{B}{s_{i-1}} \rceil$ is approximately: $\frac{B/s_i + B/s_{i-1}}{B} = \frac{1}{s_i} + \frac{1}{s_{i-1}}$. Overall, we expect one interval I_r to be present in M_i with a probability of $(\frac{1}{s_i} + \frac{1}{s_{i-1}})\omega_{i-1}$. As we have assumed that the heuristic holds for $i-1$ and by considering that one interval must contain m_0, we get:

$$\omega_i \leq (\frac{1}{s_i} + \frac{1}{s_{i-1}})(1 + 2^{i-2} s_{i-1} (\frac{B}{n})^{i-1}) \frac{Bs_i}{n} \leq 1 + 2^{i-1} s_i (\frac{B}{n})^i.$$

Using this heuristic and the fact that s_1 is approximately $1/\Pr(P) = n/B$, we bound the number of intervals in M_1 by $\omega_1 \leq 1 + 1 = 2$. As this is a bound, we assume for the remaining part of the analysis that $|M_1| = 1$ and we do not have to do a single call to step 2b. We can also analyze how many intervals we expect M_2 to contain. Because s_2 has roughly the value $2/\Pr(P) = 2n/B$, we have $\omega_2 \leq 1 + 4\frac{n}{B}(\frac{B}{n})^2 = 1 + 4\frac{B}{n} < 1 + 2^{-13}$. With a high probability at most one round of step 2b is necessary.

Next, we bound the number of oracle calls needed for step 2c. If we are in step 2c and searching for s_i we know that M_{i-1} only has one interval left, i.e., $M_{i-1} = [a, b]$ and $a \leq m_0 \leq b$. Hence, if $m_0 s_i \pmod{n}$ is accepted by the oracle, we know:

$$\frac{2B + r_i n}{b} \leq \frac{2B + r_i n}{m_0} \leq s_i \leq \frac{3B - 1 + r_i n}{m_0} \leq \frac{3B - 1 + r_i n}{a}.$$

The length of the outer interval is:

$$\frac{3B - 1 + r_i n}{a} - \frac{2B + r_i n}{b} \geq \frac{3B - 1 + r_i n}{m_0} - \frac{2B + r_i n}{m_0} \geq \frac{B-1}{m_0} \geq \frac{B-1}{3B}.$$

We can deduce that we can find an s_i value for each third value r_i that is tried. Additionally, Bleichenbacher approximates the probability that this s_i is also contained in the interval $[\frac{2B+r_i n}{m_0}, \frac{3B-1+r_i n}{m_0}]$ and hence accepted by our oracle by roughly $1/2$ [3]. It follows that we find an s_i such that the oracle accepts the decoded message after trying about two chosen ciphertexts. Because of how we choose s_i based on the previous s_{i-1}, the remaining interval is divided in half in each step. Using this, we expect step 2c to finish after around $16k = 2^{10}$ steps.

In total, to decode an encrypted message, approximately $2^{16} + 2^{10} = 66\,560$ calls to the oracle are necessary.

keylength = 1024 bits

The analysis of steps 2a and 2b does not depend on the keysize. Therefore their complexity is the same.

For step 2c, we need $16k$ calls to the oracle. For a keylength of 1024 bits, this corresponds to $16 \cdot 128 = 2^{11}$ oracle calls.

In total, to decode an encrypted message, approximately $2^{16} + 2^{11} = 67\,584$ calls to the oracle are necessary.

keylength = 2048 bits

The analysis of steps 2a and 2b does not depend on the keysize. Therefore their complexity is the same.

For step 2c, we need $16k$ calls to the oracle. For a keylength of 2048 bits, this corresponds to $16 \cdot 256 = 2^{12}$ oracle calls.

In total, to decode an encrypted message, approximately $2^{16} + 2^{12} = 69\,632$ calls to the oracle are necessary.

3.3.2 TFT Oracle

For the remaining three oracles, we conduct the analyses the same way, using the heuristics introduced for the TTT oracle.

$\Pr(P)$ again denotes the probability that a randomly chosen plaintext is accepted by the oracle. For the TFT oracle, this means that in addition to the first two bytes being `0x00` and `0x02`, the next eight bytes have to be nonzero. We have

$$\Pr(P \mid A) = \frac{255}{256}^8$$

and by using $\Pr(A)$, we deduce:

$$0.96 \cdot 2^{-16} < \Pr(P) < 0.97 \cdot 2^{-15}.$$

We observe that this additional check decreases the probability of producing a valid plaintext by randomly guessing ciphertexts only by a small fraction.

keylength = 512 bits

We approximate the complexity of step 2a by $1/\Pr(P) < 1.05 \cdot 2^{16}$.

Each round of step 2b needs $1/\Pr(P) < 1.05 \cdot 2^{16}$ calls to the oracle. The additional checks influence the probability of having to conduct a round of step 2b. Because s_1 is roughly $1/\Pr(P) < 1.05n/B$, the number of intervals in M_1 is $\omega_1 < 1 + 1.05 = 2.05$. As this is a bound and based on heuristics, we assume that $|M_1| = 1$ and we can skip step 2b. We approximate s_2 by $2/\Pr(P) < 2 \cdot 1.05n/B$ and get $\omega_2 < 1 + 1.05 \cdot 2^{-13}$. With a high probability, at most one round of step 2b is needed.

In step 2c, it is again easy to find values r_i, s_i, which satisfy the conditions. For each possible s_i, we want to know the probability that multiplying it with m_0 results in a message accepted by the oracle. This happens if the first two bytes are `0x00 || 0x02` and the bytes three to ten are nonzero. We know that the first case happens roughly with probability $1/2$ and $\Pr(P \mid A) = \frac{255}{256}^8$. Overall, each s_i we try is accepted by the TFT oracle with a probability of $(\frac{255}{256}^8)/2$. Hence, we expect to find the next s_i after $2 \cdot 1.05$ oracle calls. Therefore, to complete step 2c it takes $1.05 \cdot 16k = 1.05 \cdot 2^{10}$ calls to the oracle.

In total, to decode an encrypted message, approximately $1.05(2^{16} + 2^{10}) = 69\,888$ calls to the oracle are necessary.

keylength = 1024 bits

The analysis of steps 2a and 2b does not depend on the keysize. Therefore their complexity is the same.

For step 2c, we need $1.05 \cdot 16k$ calls to the oracle. For a keylength of 1024 bits, this corresponds to $1.05 \cdot 16 \cdot 128 = 1.05 \cdot 2^{11}$ calls to the oracle.

In total, to decode an encrypted message, approximately $1.05(2^{16} + 2^{11}) < 70\,964$ calls to the oracle are necessary.

keylength = 2048 bits

The analysis of steps 2a and 2b does not depend on the keysize. Therefore their complexity is the same.

For step 2c, we need $1.05 \cdot 16k$ calls to the oracle. For a keylength of 2048 bits, this corresponds to $1.05 \cdot 16 \cdot 256 = 1.05 \cdot 2^{12}$ calls to the oracle.

In total, to decode an encrypted message, approximately $1.05(2^{16} + 2^{12}) < 73\,114$ calls to the oracle are necessary.

3.3.3 FFT Oracle

For the FFT oracle, $\Pr(P)$ is equal to the probability that a randomly chosen plaintext has the correct format specified by the PKCS #1 v1.5 standards. The oracle checks that the first byte is `0x00`, the second byte `0x02`, the following eight bytes are nonzero and that at least one of the remaining bytes is `0x00`. We have

$$\Pr(P \mid A) = \frac{255}{256}^8 \cdot (1 - (\frac{255}{256})^{k-10}).$$

The constraint that the resulting plaintext of some ciphertext has to have another zero byte in the bytes 11 to k influences the probability of randomly generating an accepted ciphertext a lot. This probability also depends on the keylength. For longer keys, it is more probable that one byte in the plaintext of a randomly generated ciphertext is `0x00`. In the following analysis, we see how these two factors influence the performance of the attack.

keylength = 512 bits

For a keylength of 512 bits we have: $0.184 < \Pr(P \mid A) < 0.185$.

It follows that: $0.184 \cdot 2^{-16} < \Pr(P) < 0.185 \cdot 2^{-15}$.

We approximate the complexity of step 2a by $1/\Pr(P) < 5.44 \cdot 2^{16}$.

Each round of step 2b needs $1/\Pr(P) < 5.44 \cdot 2^{16}$ calls to the oracle. The additional checks influence the probability of having to conduct a round of step of 2b. Because s_1 is roughly $1/\Pr(P) < 5.44n/B$, the number of intervals in M_1 is $\omega_1 < 1 + 5.44 = 6.44$. We can see that if we perform all checks on the plaintext to determine whether it is PKCS #1 v1.5 conforming, it is much more likely that the adversary has to do a round of step 2b. We approximate s_2 by $2/\Pr(P) < 2 \cdot 5.44n/B$ and get $\omega_2 < 1 + 5.44 \cdot 2^{-13}$. With a high probability, at most one round of step 2b is needed. For the FFT oracle with a keylength of 512 bits, we assume that we have to do one round of step 2b. Hence, step 2b requires $1/\Pr(P) < 5.44 \cdot 2^{16}$ oracle calls.

The new checks also influence the number of calls needed for step 2c. The probability that a value s_i which satisfies the conditions in step 2c is also PKCS #1 v1.5 conforming is roughly $\Pr(P \mid A)/2$. For a keysize of 512 bits this means we need around $2 \cdot 5.44$ calls to the oracle to find the next conforming multiplier s_i. Therefore, to complete step 2c it takes $5.44 \cdot 16k = 5.44 \cdot 2^{10} = 1.36 \cdot 2^{12}$ calls to the oracle.

In total, to decode an encrypted message, approximately $5.44(2 \cdot 2^{16} + 2^{10}) = 5.44(2^{17} + 2^{10}) = 1.36(2^{19} + 2^{12}) < 718\,603$ calls to the oracle are necessary.

keylength = 1024 bits

For a keylength of 1024 bits we have: $0.358 < \Pr(P \mid A) < 0.359$.

It follows that: $0.358 \cdot 2^{-16} < \Pr(P) < 0.359 \cdot 2^{-15}$.

We approximate the complexity of step 2a by $1/\Pr(P) < 2.8 \cdot 2^{16}$.

Each round of step 2b needs $1/\Pr(P) < 2.8 \cdot 2^{16}$ calls to the oracle. Because s_1 is roughly $1/\Pr(P) < 2.8n/B$, the number of intervals in M_1 is $\omega_1 < 1 + 2.8 = 3.8$. Compared to the first two oracle types, it is still more likely that the adversary has to do a round of 2b. However, with growing key size, the probability decreases as it is easier to generate a `0x00` in the remaining $k - 10$ bytes. We approximate s_2 by $2/\Pr(P) < 2 \cdot 2.8n/B$ and get $\omega_2 < 1 + 2.8 \cdot 2^{-13}$. It still holds that with a high probability, at most one round of step 2b is needed. For the FFT oracle with a keylength of 1024 bits, we assume that we have to do one round of step 2b. Hence, step 2b requires $1/\Pr(P) < 2.8 \cdot 2^{16}$ oracle calls.

For step 2c, we need $2.8 \cdot 16k$ calls to the oracle. For a keylength of 1024 bits, this corresponds to $2.8 \cdot 16 \cdot 128 = 2.8 \cdot 2^{11}$ calls to the oracle.

In total, to decode an encrypted message, approximately $2.8(2 \cdot 2^{16} + 2^{11}) = 2.8(2^{17} + 2^{11}) = 1.4(2^{18} + 2^{12}) = 372\,736$ calls to the oracle are necessary.

keylength = 2048 bits

For a keylength of 2048 bits we have: $0.599 < \Pr(P \mid A) < 0.6$.

It follows that: $0.599 \cdot 2^{-16} < \Pr(P) < 0.6 \cdot 2^{-15}$.

We approximate the complexity of step 2a by $1/\Pr(P) < 1.67 \cdot 2^{16}$.

Each round of step 2b needs $1/\Pr(P) < 1.67 \cdot 2^{16}$ calls to the oracle. Because s_1 is roughly $1/\Pr(P) < 1.67n/B$, the number of intervals in M_1 is $\omega_1 < 1 + 1.67 = 2.67$. This bound again shows that with growing keylength, it is easier to find an accepted ciphertext and less likely that we have to do a round of step 2b. But, the probability of performing step 2b is still higher than for the more permissive oracles, which perform fewer checks. We approximate s_2 by $2/\Pr(P) < 2 \cdot 1.67n/B$ and get $\omega_2 < 1 + 1.67 \cdot 2^{-13}$. With a high probability at most one round of step 2b is needed. For the FFT oracle with a keylength of 2048 bits, we assume that we have to do one round of step 2b. Hence, step 2b requires $1/\Pr(P) < 1.67 \cdot 2^{16}$ oracle calls.

For step 2c, we need $1.67 \cdot 16k$ calls to the oracle. For a keylength of 2048 bits, this corresponds to $1.67 \cdot 16 \cdot 256 = 1.67 \cdot 2^{12}$ calls to the oracle.

In total, to decode an encrypted message, approximately $1.67(2 \cdot 2^{16} + 2^{12}) = 1.67(2^{17} + 2^{12}) < 225\,731$ calls to the oracle are necessary.

3.3.4 Bad Version Oracle

The BVO is an instance of a FFF oracle. For any FFF oracle, the message length is fixed, which means that this oracle checks for a zero byte at a particular position. This decreases the probability of an attacker generating an accepted plaintext by randomly trying ciphertexts significantly. The probability that a random plaintext has a `0x00` byte at a fixed position is 2^{-8}. For the BVO, we know that the message has a size of 48 bytes. The first two bytes of this message cannot be equal to `major || minor`. This, however, does not influence the performance too much because any random plaintext satisfies this condition with a probability of $1 - 2^{-16}$. So, this analysis for the BVO is almost the same as for a "pure" FFF oracle. Additionally, because of the fixed message length, the nonzero padding is longer the larger the keylength. This indicates that it gets harder to generate a valid ciphertext with growing keylength.

To analyze the complexity of the attack using this oracle, we introduce a new probability. Let $\Pr(S - PKCS \mid A)$ be the probability that the plaintext is S-PKCS conforming, according to our definition in Section 2.3.3, assuming that the first two bytes of the plaintext are `0x00 || 0x02`. We can see that

$$\Pr(S - PKCS \mid A) = \frac{255}{256}^{k-51} \cdot 2^{-8}.$$

After the first two bytes, there have to be $k - 51$ nonzero bytes followed by one `0x00` byte. The probability that we receive an error message if the randomly chosen ciphertext is S-PKCS conforming is

$$\Pr(P \mid S - PKCS) = 1 - 2^{-16}.$$

Combining this, we obtain:

$$\Pr(P|A) = \Pr(P|S - PKCS) \cdot \Pr(S - PKCS|A) = (1 - 2^{-16})(\frac{255}{256}^{k-51} \cdot 2^{-8})$$

and

$$2^{-24}(1 - 2^{-16})(\frac{255}{256}^{k-51}) < \Pr(P) < 2^{-23}(1 - 2^{-16})(\frac{255}{256}^{k-51}).$$

These probabilities depend on the size of the key and are larger the smaller the keylength.

keylength = 512 bits

For a keylength of 512 bits we have: $0.95 \cdot 2^{-8} < \Pr(P \mid A) < 0.96 \cdot 2^{-8}$.

It follows that: $0.95 \cdot 2^{-24} < \Pr(P) < 0.96 \cdot 2^{-23}$.

We approximate the complexity of step 2a by $1/\Pr(P) < 1.06 \cdot 2^{24}$.

Each round of step 2b needs $1/\Pr(P) < 1.06 \cdot 2^{24}$ calls to the oracle. Because s_1 is roughly $1/\Pr(P) < 1.06 \cdot 2^{8} \cdot \frac{n}{B} = 271.36 \cdot \frac{n}{B}$, the number of intervals in M_1 is $\omega_1 < 1 + 271.36 = 272.36$. With a high probability we have to do at least one round of 2b. We approximate s_2 by $2/\Pr(P) < 1.06 \cdot 2^9 \cdot \frac{n}{B}$ and get $\omega_2 \leq 1 + 1.06 \cdot 2^{10} \cdot \frac{B}{n} < 1 + 1.06 \cdot 2^{-5} < 1.04$. It still holds that with a high probability, at most one round of step 2b is needed. For the BVO with a keylength of 512 bits, we assume that we have to do one round of step 2b. Hence, step 2b requires $1/\Pr(P) < 1.06 \cdot 2^{24}$ calls.

The more restrictive checks of the oracle, especially checking for a zero byte at a certain fixed location, also have an influence on the performance of step 2c. The probability that a value s_i which satisfies the conditions in step 2c also results in a ciphertext accepted by the BVO is roughly $\Pr(P \mid A)/2$. This implies that an attacker has to do around $1.06 \cdot 2^9$ calls to the oracle to find the next s_i in step 2c. For a keylength of 512 bits, this corresponds to $1.06 \cdot 2^8 \cdot 16k = 1.06 \cdot 2^{12} \cdot 64 = 1.06 \cdot 2^{18}$ calls to the oracle.

In total, to decode an encrypted message, approximately $1.06(2 \cdot 2^{24} + 2^{18}) = 1.06(2^{25} + 2^{18}) < 35\,845\,571$ calls to the oracle are necessary.

We observe that the number of oracle calls required to decrypt a ciphertext has increased significantly compared to the more permissive oracles. As the BVO is of practical relevance, this motivates finding improvements to the attack, which are presented in Chapter 4.

keylength = 1024 bits

For a keylength of 1024 bits we have: $0.73 \cdot 2^{-8} < \Pr(P \mid A) < 0.74 \cdot 2^{-8}$.

It follows that: $0.73 \cdot 2^{-24} < \Pr(P) < 0.74 \cdot 2^{-23}$.

Each round of step 2b needs $1/\Pr(P) < 1.37 \cdot 2^{24}$ calls to the oracle. Because s_1 is roughly $1/\Pr(P) < 1.37 \cdot 2^8 \cdot \frac{n}{B} = 350.72 \cdot \frac{n}{B}$, the number of intervals in M_1 is $\omega_1 < 1 + 350.72 = 351.72$. With a high probability we have to do at least one round of 2b. We approximate s_2 by $2/\Pr(P) < 1.37 \cdot 2^9 \cdot \frac{n}{B}$ and get $\omega_2 \leq 1 + 1.37 \cdot 2^{10} \cdot \frac{B}{n} < 1 + 1.37 \cdot 2^{-5} < 1.05$. It still holds that with a high probability, at most one round of step 2b is needed. For the BVO with a keylength of 1024 bits, we assume that we have to do one round of step 2b. Hence, step 2b requires $1/\Pr(P) < 1.37 \cdot 2^{24}$ calls.

For step 2c, we need $1.37 \cdot 2^8 \cdot 16k$ calls to the oracle. For a keylength of 1024 bits, this corresponds to $1.37 \cdot 2^8 \cdot 16 \cdot 128 = 1.37 \cdot 2^{19}$ calls to the oracle.

In total, to decode an encrypted message, approximately $1.37(2 \cdot 2^{24} + 2^{19}) = 1.37(2^{25} + 2^{19}) < 46\,687\,847$ calls to the oracle are necessary.

keylength = 2048 bits

For a keylength of 1024 bits we have: $0.44 \cdot 2^{-8} < \Pr(P \mid A) < 0.45 \cdot 2^{-8}$.

It follows that: $0.44 \cdot 2^{-24} < \Pr(P) < 0.45 \cdot 2^{-23}$.

We approximate the complexity of step 2a by $1/\Pr(P) < 2.28 \cdot 2^{24}$.

Each round of step 2b needs $1/\Pr(P) < 2.28 \cdot 2^{24}$ calls to the oracle. Because s_1 is roughly $1/\Pr(P) < 2.28 \cdot 2^8 \cdot \frac{n}{B} = 583.68 \cdot \frac{n}{B}$, the number of intervals in M_1 is $\omega_1 < 1 + 583.68 = 584.68$. With a high probability we have to do at least one round of 2b. We approximate s_2 by $2/\Pr(P) < 2.28 \cdot 2^9 \cdot \frac{n}{B}$ and get $\omega_2 \leq 1 + 2.28 \cdot 2^{10} \cdot \frac{B}{n} < 1 + 2.28 \cdot 2^{-5} < 1.08$. It still holds that with a high probability, at most one round of step 2b is needed. However, we observe that with growing keylength the probability of having to perform two rounds of step 2c increases. For the BVO with a keylength of 2048 bits, we assume that we have to do one round of step 2b. Hence, step 2b requires $1/\Pr(P) < 2.28 \cdot 2^{24}$ calls.

For step 2c, we need $2.28 \cdot 2^8 \cdot 16k$ calls to the oracle. For a keylength of 2048 bits, this corresponds to $2.28 \cdot 2^8 \cdot 16 \cdot 256 = 2.28 \cdot 2^{20}$ calls to the oracle.

In total, to decode an encrypted message, approximately $2.28(2 \cdot 2^{24} + 2^{20}) = 2.28(2^{25} + 2^{20}) = 1.14(2^{26} + 2^{21}) < 78\,894\,859$ calls to the oracle are necessary.

3.4 Experimental Results

In this section, we present the results of Bleichenbacher's attack in practice. We implemented the Bleichenbacher attack and measured its complexity for the different oracle types and keylengths. For the TTT, TFT and FFT oracles, we conducted 1 000 000 simulations and for the BVO 10 000 simulations with each keylength.

In preparation for the experiments, we precomputed 1 000 private and public key pairs for each keylength. These key pairs were used in a round-robin fashion for the different simulations of an experiment. For each simulation of Bleichenbacher's attack, we generated a random message. For the TTT, TFT and FFT oracles, we first determined a random message length and then generated a message of the corresponding length. For the BVO, the message size is fixed, so a random message of 46 bytes was generated, and the version numbers were prepended. We used the version numbers of TLS 1.2, namely `0x03 || 0x03`. For the generation of pseudo-random bit strings, we implemented a pseudo-random number generator based on AES to provide a good source of randomness [7]. Furthermore, we constructed a unique AES key for every simulation of an experiment to avoid reusing outputs. Then, we produced pseudo-random numbers by providing a counter value as an input to the AES instance in ECB mode and incrementing this counter by one after each generated data block.

In the following parts, we look at the different oracles separately. We present the experimental results and compare them to the theoretical analysis. For each oracle type and keylength, we computed the mean and median over the number of simulations. Additionally, we also provide the mean and median values for the individual steps and the number of rounds required for step 2b. Then, we visualize the distributions of the total number of oracle calls and oracle calls for the individual steps. In each figure, the x-axis shows the number of required oracle calls for the attack and the y-axis the frequency of this number of calls over the performed simulations. The x-axis is scaled differently for the BVO compared to the other three oracle types as its complexity is larger. Furthermore, for the TTT, TFT and FFT oracles, outliers above 2^{26} and for the BVO outliers above 2^{33} oracle calls were removed from the plots. We explicitly mention whenever an outlier was removed.

3.4.1 TTT Oracle

The experimental results confirm many of the predictions from the theoretical analysis in Section 3.3.1. Firstly, if they have to be performed, steps 2a and 2b are the most expensive. As we know, step 2a always has to be done. However, step 2b can be skipped most of the time for the TTT oracle. This is visible in Figures 3.1, 3.2 and 3.3, as well as Table 3.5. In the figures, we see a lot fewer orange bars, indicating step 2b, compared to the blue or green bars for steps 2a and 2c. Hence, this step does not have to be performed often. From Table 3.5, we can read out that in the median, we do not have to perform step 2b, and in the mean, we only have to perform 0.21 rounds of step 2b. We note that one round of step 2b corresponds to finding the next suitable integer s_i and updating all intervals.

Additionally, the distributions of the number of oracle calls to steps 2a and 2b do not depend on the keylength. This can be seen in Figures 3.1, 3.2 and 3.3 because the distributions for steps 2a and 2b are identical. Lastly, the number of oracle calls for step 2c increases by a factor of two when doubling the keylength. This is visible in Table 3.4 and by comparing the green bars in Figures 3.1, 3.2 and 3.3. We also observe that the theoretical analysis nicely approximates the median of step 2c.

On the other hand, from the figures and Table 3.2, it follows that we overestimated the complexity of step 2a in the theoretical analysis in Section 3.3.1. This step usually requires fewer oracle calls. Another interesting observation from the figures is that there exists an interval where we know s_1 cannot lie. We can see this as there is a region into which the number of calls required for step 2a never falls, see Figures 3.1, 3.2 and 3.3. This region does not depend on the keylength as it is present for all keylengths. Additionally, if we have to do at least one round of step 2b, the number of oracle calls needed for step 2b is constantly above this "hole". This observation can be explained mathematically and used to improve the attack, as shown in Section 4.4.

What also stands out is that the mean values are always significantly larger than the respective median values, see Tables 3.1, 3.2, 3.3 and 3.4. This can be explained by the fact that the distributions for both the total number of oracle calls and the individual steps have a long tail, as visible in Figures 3.1, 3.2 and 3.3. Thus, there exist rare cases where at least one step of the attack takes abnormally long. Interestingly, these cases exist for steps 2a, 2b as well as 2c.

keylength	median total	mean total	theoretical analysis total
512	21 114	36 272	66 560
1024	23 000	40 815	67 584
2048	26 439	49 767	69 632

Table 3.1: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	theoretical analysis 2a
512	19 178	27 180	65 536
1024	19 318	27 361	65 536
2048	19 642	27 841	65 536

Table 3.2: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	theoretical analysis 2b
512	0	5 634	0
1024	0	5 675	0
2048	0	5 829	0

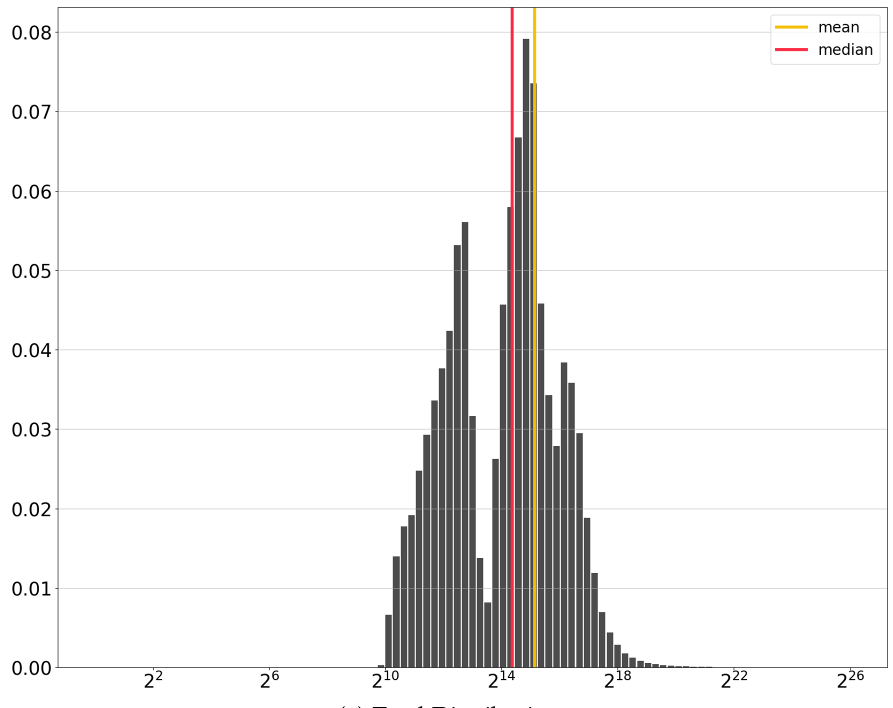
Table 3.3: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – Step 2b

keylength	median 2c	mean 2c	theoretical analysis 2c
512	1 030	3 458	1 024
1024	2 126	7 779	2 048
2048	4 320	16 097	4 096

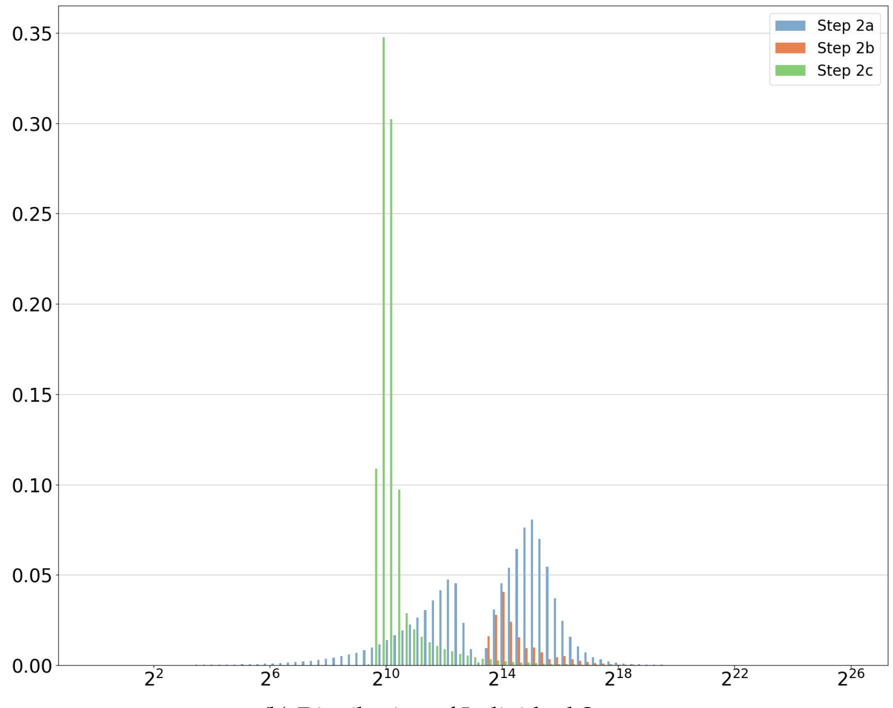
Table 3.4: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – Step 2c

keylength	median rounds 2b	mean rounds 2b	theoretical analysis rounds 2b
512	0	0.21	0
1024	0	0.21	0
2048	0	0.21	0

Table 3.5: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – Rounds of Step 2b

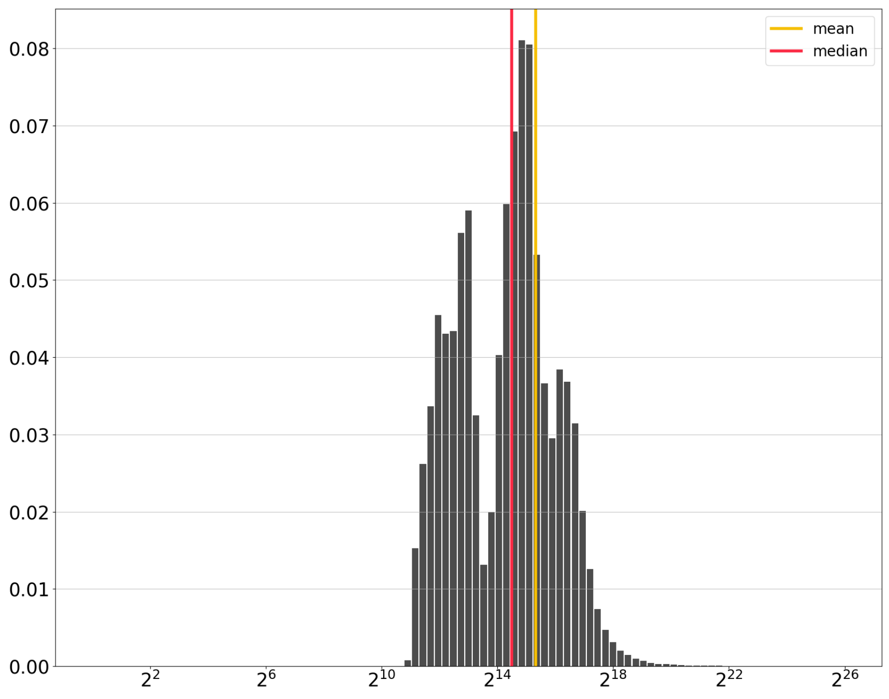


(a) Total Distribution

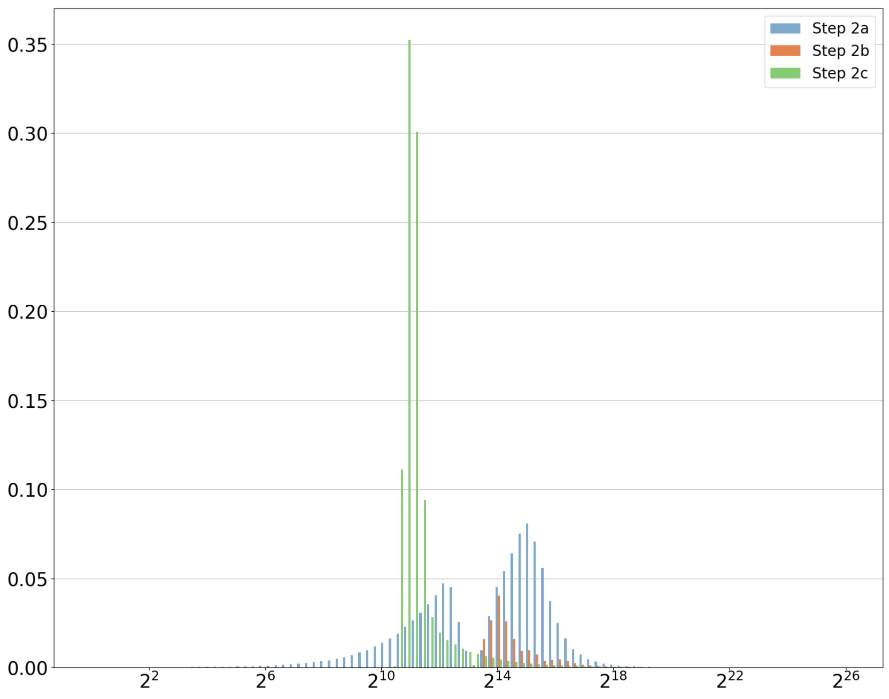


(b) Distribution of Individual Steps

Figure 3.1: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 3 from Total, 3 from Step 2c

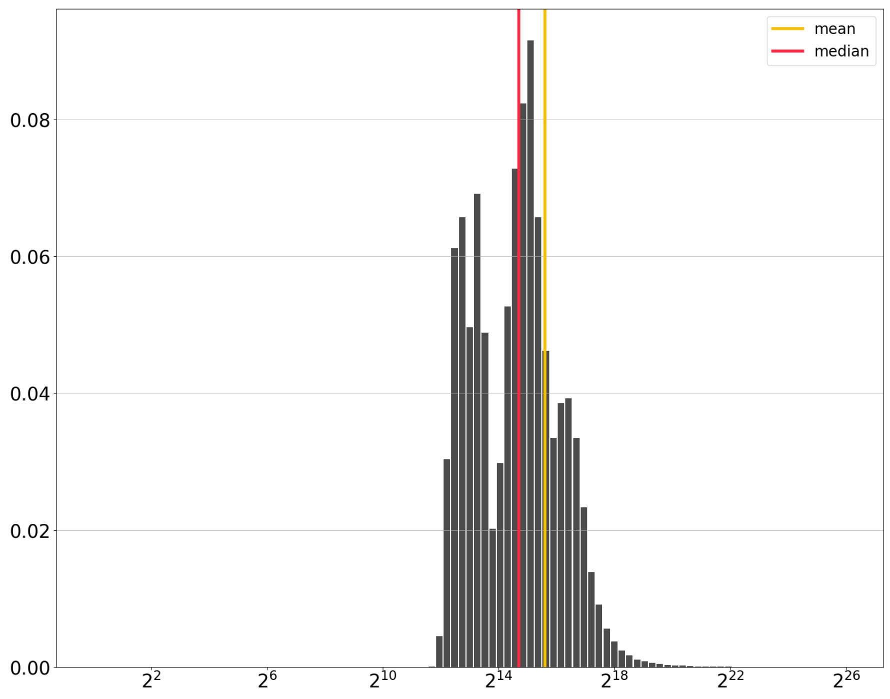


(a) Total Distribution

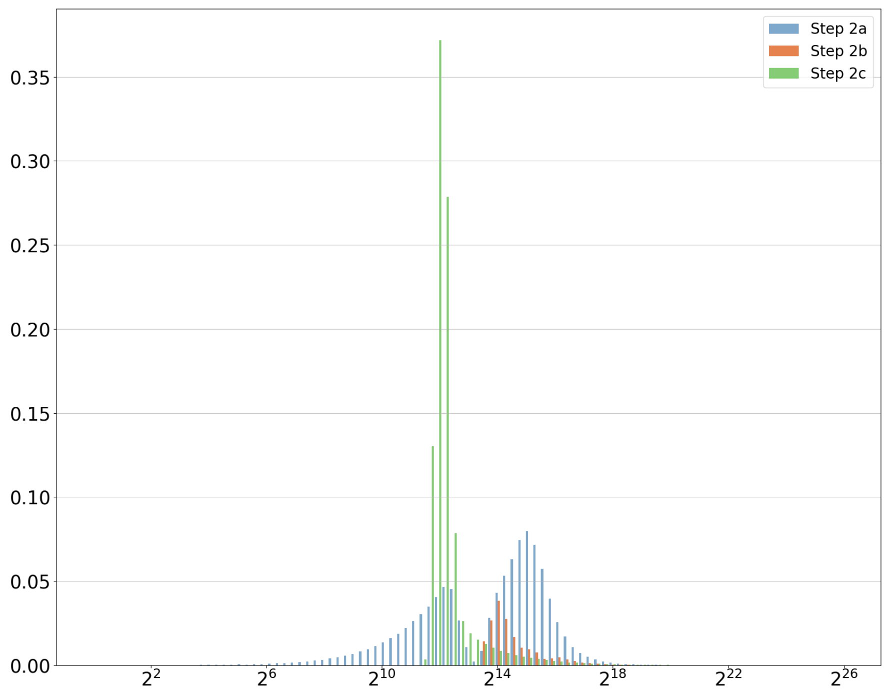


(b) Distribution of Individual Steps

Figure 3.2: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – keylength 1024 – outliers removed: 5 from Total, 5 from Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 3.3: Bleichenbacher's Attack – TTT Oracle – 10^6 Simulations – keylength 2048 – outliers removed: 11 from Total, 11 from Step 2c

3.4.2 TFT Oracle

As already predicted by the theoretical analysis, the additional check performed by the TFT oracle does not significantly change the number of required oracle calls. The distributions of the oracle calls for the different parts of the algorithm stay the same. Hence, we do not provide the figures for them as they look identical to the TTT oracle shifted by a small factor. We can also see from the provided mean and median values in Tables 3.6, 3.7, 3.8 and 3.9 that the factor by which the complexity grows is roughly 1.05, as approximated by the theoretical analysis in Section 3.3.2. The other observations are the same as for the TTT oracle.

keylength	median total	mean total	theoretical analysis total
512	22 077	39 530	69 888
1024	24 010	42 981	70 964
2048	27 486	51 362	73 114

Table 3.6: Bleichenbacher's Attack – TFT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	theoretical analysis 2a
512	20 115	28 801	68 813
1024	20 259	29 017	68 813
2048	20 517	29 421	68 813

Table 3.7: Bleichenbacher's Attack – TFT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	theoretical analysis 2b
512	0	6 969	0
1024	0	7 000	0
2048	0	6 911	0

Table 3.8: Bleichenbacher's Attack – TFT Oracle – 10^6 Simulations – Step 2b

keylength	median 2c	mean 2c	theoretical analysis 2c
512	1 071	3 759	1 076
1024	2 213	6 964	2 151
2048	4 497	15 030	4 301

Table 3.9: Bleichenbacher's Attack – TFT Oracle – 10^6 Simulations – Step 2c

keylength	median rounds 2b	mean rounds 2b	theoretical analysis rounds 2b
512	0	0.23	0
1024	0	0.23	0
2048	0	0.23	0

Table 3.10: Bleichenbacher's Attack – TFT Oracle – 10^6 Simulations – Rounds of Step 2b

3.4.3 FFT Oracle

For the FFT oracle, the complexity of step 2a increases significantly because it checks for a `0x00` byte, see Table 3.12. We still overestimated the complexity of steps 2a and 2b in the theoretical analysis in Section 3.3.3. However, the influence of the different parameters on the complexity was accurately predicted. For example, we see that with increasing keylength, the attack complexity decreases as it becomes easier for an attacker to randomly generate a `0x00` byte, see Table 3.11.

The probability of performing step 2b has also increased a lot compared to the TTT and TFT oracles. This can be seen in Table 3.15 as well as Figures 3.4, 3.5 and 3.6. In the table, we observe that for keylengths of 512 and 1024 bits in the median, we have to do one round of step 2b, compared to zero rounds for the more permissive oracle types, see Tables 3.5 and 3.10. The figures show many more orange bars, corresponding to step 2b, than Figures 3.1, 3.2 and 3.3, indicating that this step is done more frequently compared to the TTT and TFT oracles. Furthermore, it is evident that steps 2a and 2b determine the complexity of the attack for this oracle, see Tables 3.12 and 3.13 as well as Figures 3.4, 3.5, 3.6.

The other observations made for the TTT oracle still apply here. There is a region into which the number of oracle calls for step 2a does not fall and the number of calls for step 2b is constantly above this region, see Figures 3.4, 3.5 and 3.6. The mean values are notably higher than the corresponding median values as all distributions are still long-tailed, as visible in the figures and Tables 3.11, 3.12, 3.13 and 3.14. Finally, the theoretical analysis in Section 3.3.3 approximates the median of step 2c quite well, see Table 3.14.

keylength	median total	mean total	theoretical analysis total
512	359 198	434 687	718 603
1024	153 859	200 970	372 736
2048	50 191	109 099	225 731

Table 3.11: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	theoretical analysis 2a
512	144 030	213 858	356 516
1024	68 647	103 614	183 501
2048	37 019	56 882	109 446

Table 3.12: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	theoretical analysis 2b
512	121 714	212 733	356 516
1024	32 483	84 762	183 501
2048	0	33 371	109 446

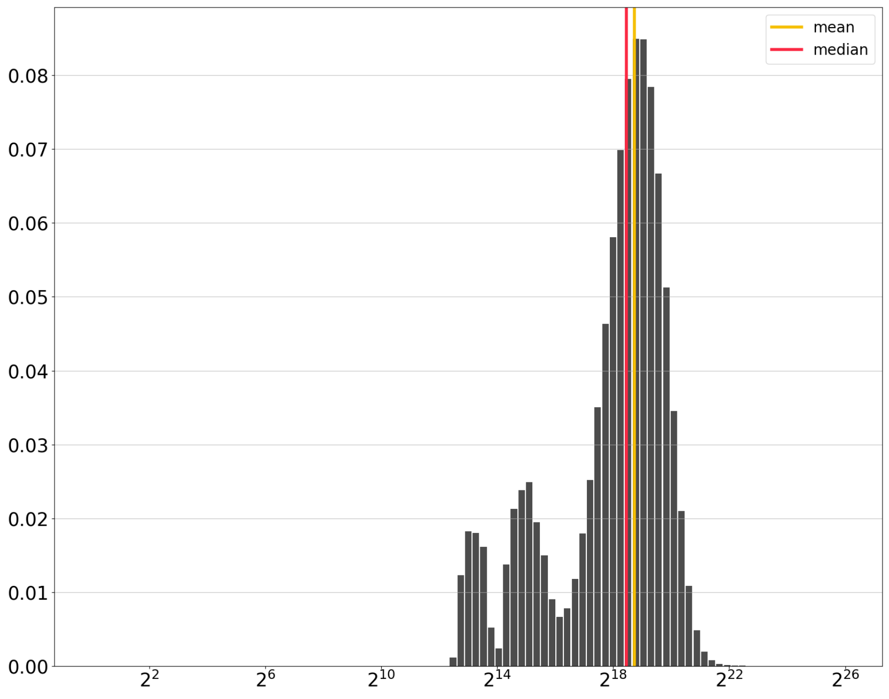
Table 3.13: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – Step 2b

keylength	median 2c	mean 2c	theoretical analysis 2c
512	6 134	8 097	5 571
1024	6 394	12 593	5 735
2048	7 545	18 846	6 841

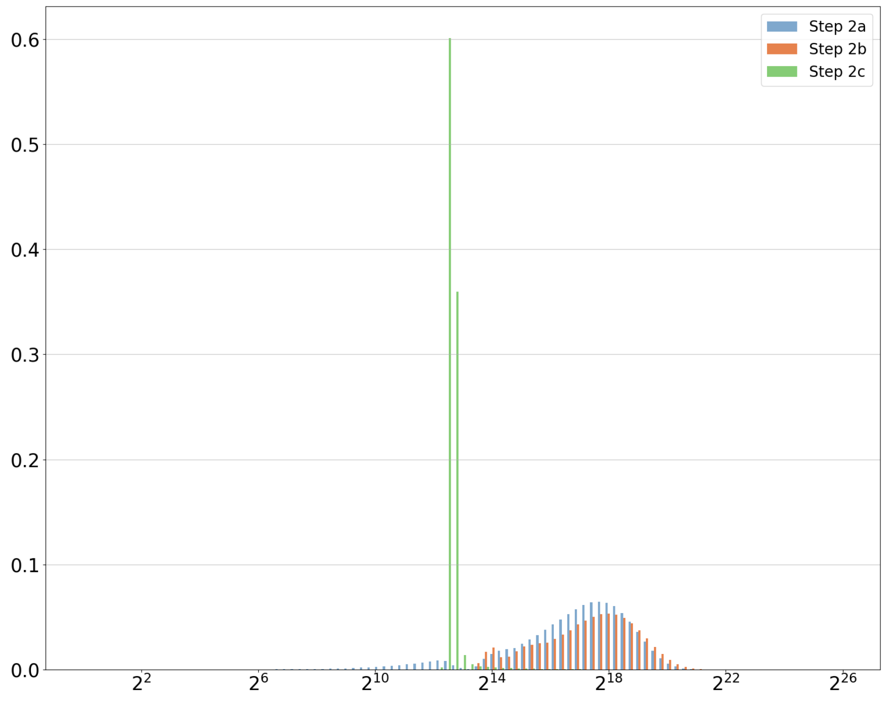
Table 3.14: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – Step 2c

keylength	median rounds 2b	mean rounds 2b	theoretical analysis rounds 2b
512	1	0.95	1
1024	1	0.75	1
2048	0	0.51	1

Table 3.15: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – Rounds of Step 2b

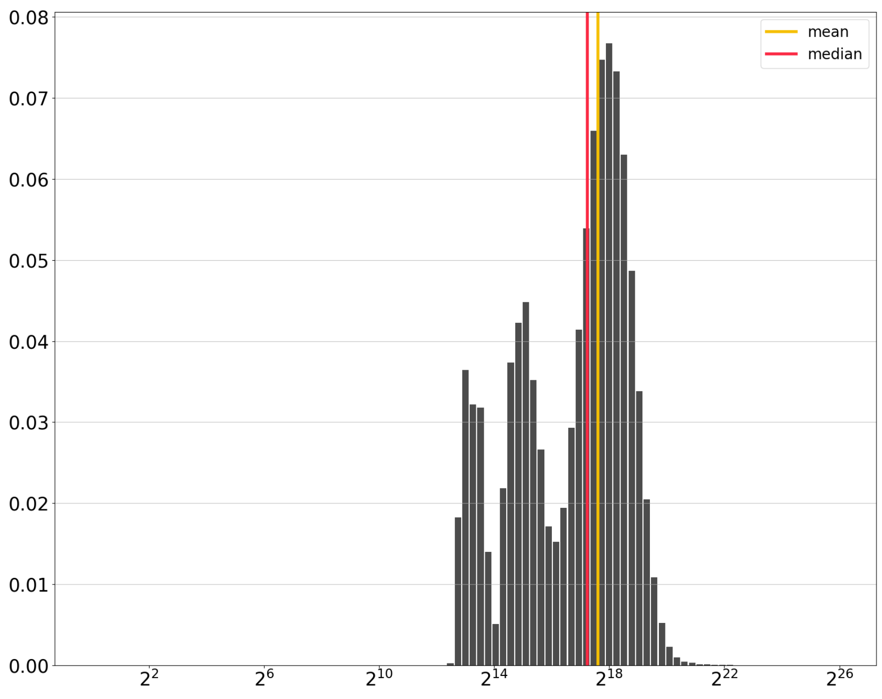


(a) Total Distribution

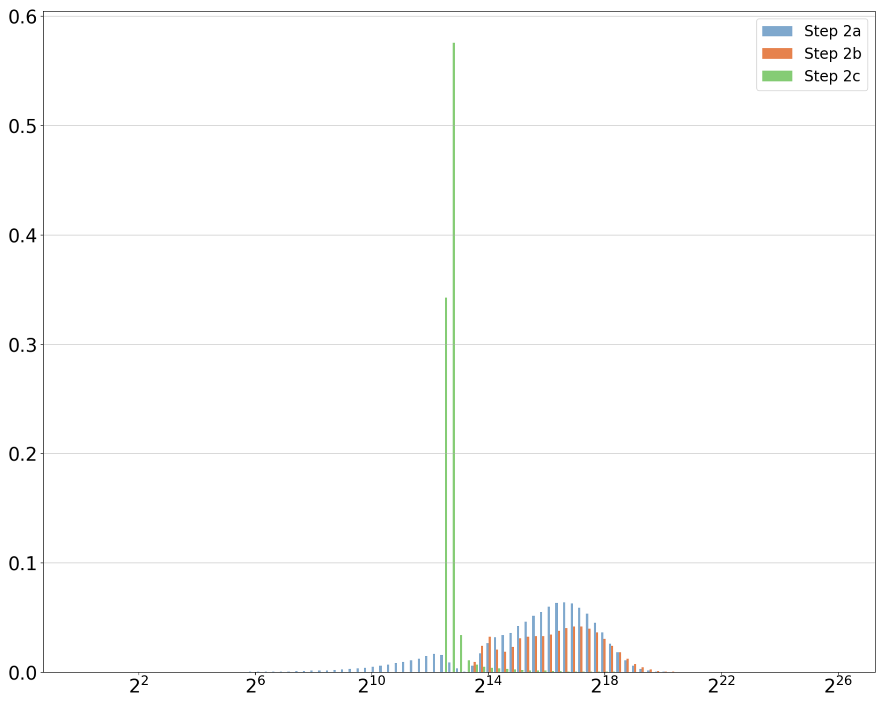


(b) Distribution of Individual Steps

Figure 3.4: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 3 from Total, 3 from Step 2c

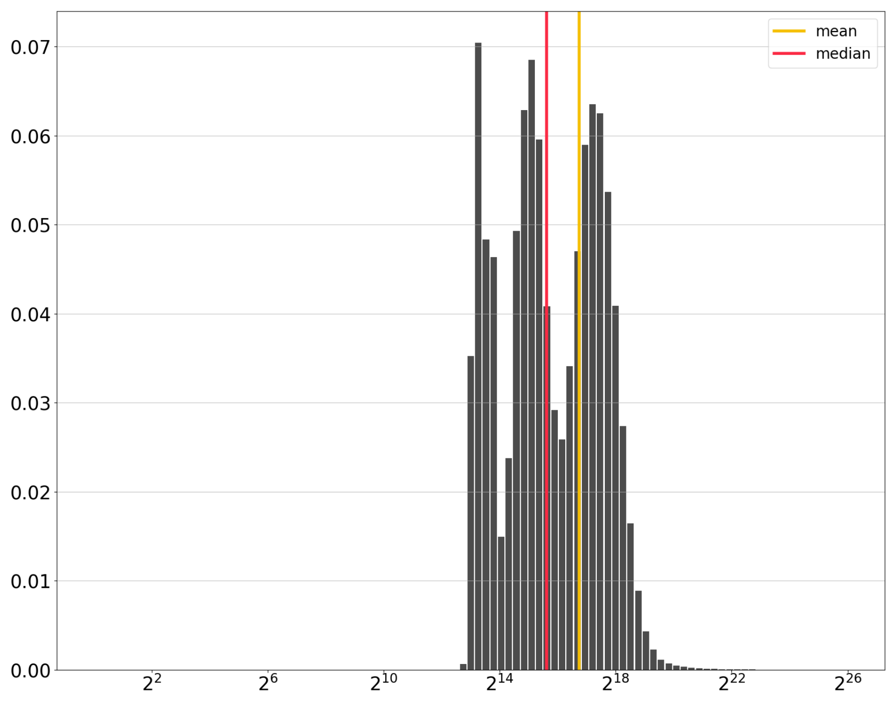


(a) Total Distribution

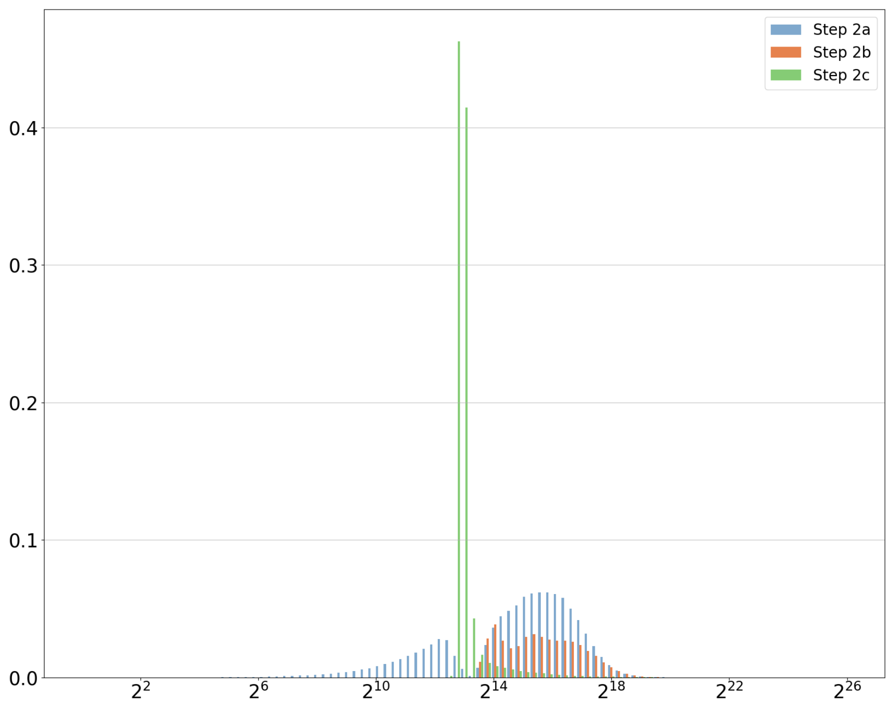


(b) Distribution of Individual Steps

Figure 3.5: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – keylength 1024 – outliers removed: 9 from Total, 1 from Step 2b, 8 from Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 3.6: Bleichenbacher's Attack – FFT Oracle – 10^6 Simulations – keylength 2048 – outliers removed: 12 from Total, 12 from Step 2c

3.4.4 Bad Version Oracle

As predicted in the theoretical analysis in Section 3.3.4, the complexity of the attack increases by roughly a factor of 2^8 compared to the FFT oracle because the BVO checks for a zero byte at a particular position. The intuition that for a longer keylength, it is harder to generate the nonzero padding has also been confirmed, see Table 3.16. For this oracle, the complexity is also determined by steps 2a and 2b, as visible in Tables 3.17 and 3.18. Additionally, the number of rounds required for step 2b to end up with a single interval has grown compared to the FFT oracle. The median of rounds required for step 2b for a modulus of 2048 bits is now also 1, see Table 3.20. This is also visible in Figures 3.7, 3.8 and 3.9 because there are now as many orange bars as blue ones, suggesting that step 2b has to be performed for every experiment.

However, one result from the experiments does not match our expectations. The mean number of rounds required for step 2b decreases with increasing keylength in Table 3.20. This results from the fact that the mean is strongly affected by outliers. For the BVO, we only performed 10 000 simulations per keylength. Therefore, a very bad run influences the mean value a lot, which happened in these cases. With this observation, it is again important to note that all the distributions have a long tail, see Figures 3.7, 3.8 and 3.9. There exist some cases where at least one of the steps 2a, 2b and 2c takes an enormous number of oracle calls.

In the next chapter, we present different improvements devised to overcome this problem and improve steps 2a and 2b. Afterwards, we introduce a heuristic based on our observations during the experimental runs to improve or eliminate the unfortunate cases of step 2c.

keylength	median total	mean total	theoretical analysis total
512	24 227 787	50 146 387	35 845 571
1024	31 076 365	62 343 618	46 687 847
2048	51 200 904	75 265 298	78 894 859

Table 3.16: Bleichenbacher's Attack – BVO – 10^4 Simulations – Total

keylength	median 2a	mean 2a	theoretical analysis 2a
512	8 019 905	14 499 301	17 783 849
1024	10 548 557	18 231 944	22 984 786
2048	17 708 014	27 544 262	38 252 053

Table 3.17: Bleichenbacher's Attack – BVO – 10^4 Simulations – Step 2a

keylength	median 2b	mean 2b	theoretical analysis 2b
512	11 113 772	34 805 695	17 783 849
1024	13 778 521	42 711 607	22 984 786
2048	22 443 601	44 014 214	38 252 053

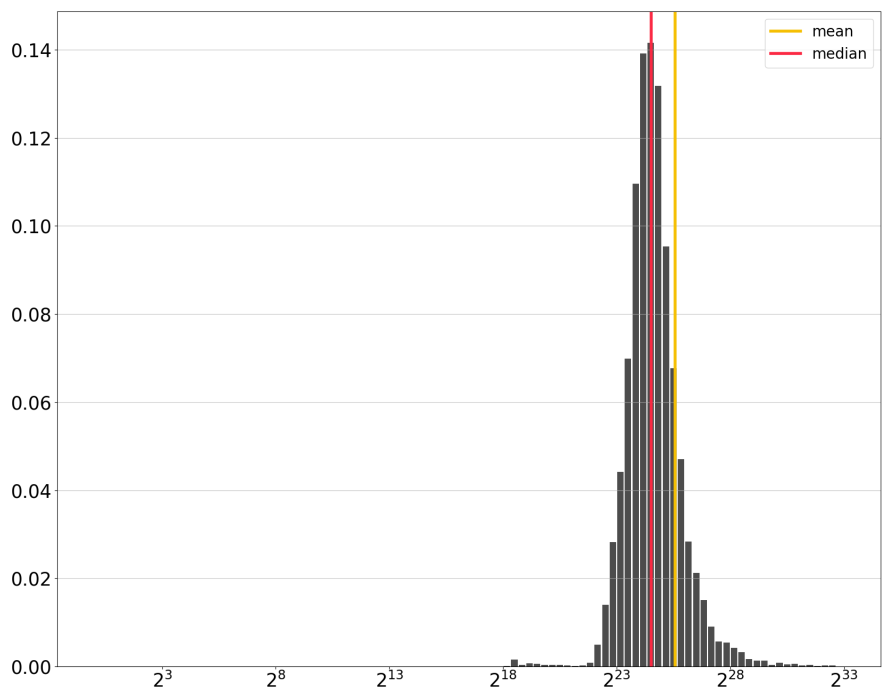
Table 3.18: Bleichenbacher's Attack – BVO – 10^4 Simulations – Step 2b

keylength	median 2c	mean 2c	theoretical analysis 2c
512	274 367	841 390	227 873
1024	742 132	1 400 068	718 275
2048	2 583 223	3 706 822	2 390 753

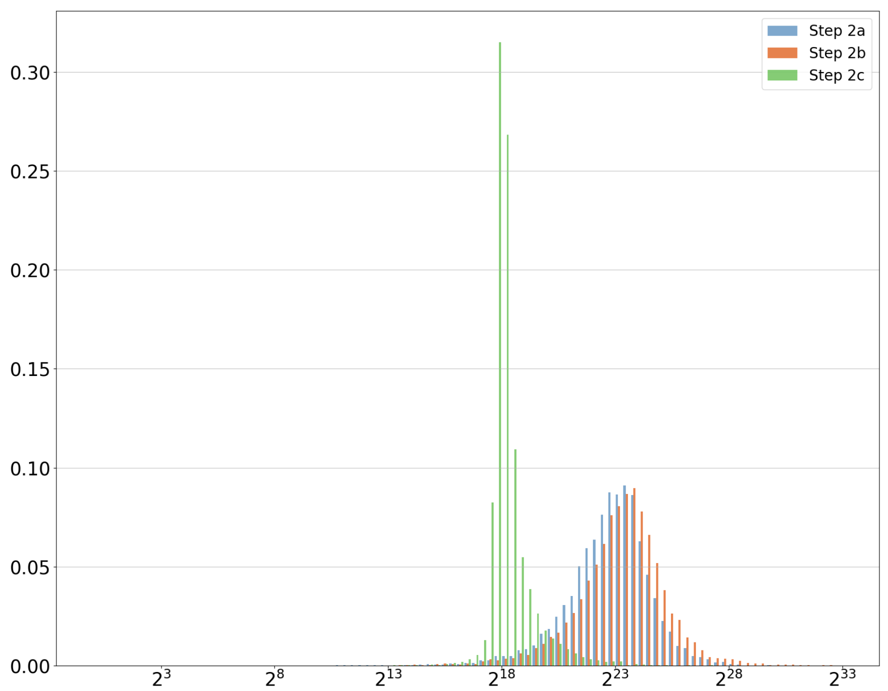
Table 3.19: Bleichenbacher's Attack – BVO – 10^4 Simulations – Step 2c

keylength	median rounds 2b	mean rounds 2b	theoretical analysis rounds 2b
512	1	2.68	1
1024	1	2.48	1
2048	1	1.78	1

Table 3.20: Bleichenbacher's Attack – BVO – 10^4 Simulations – Rounds of Step 2b

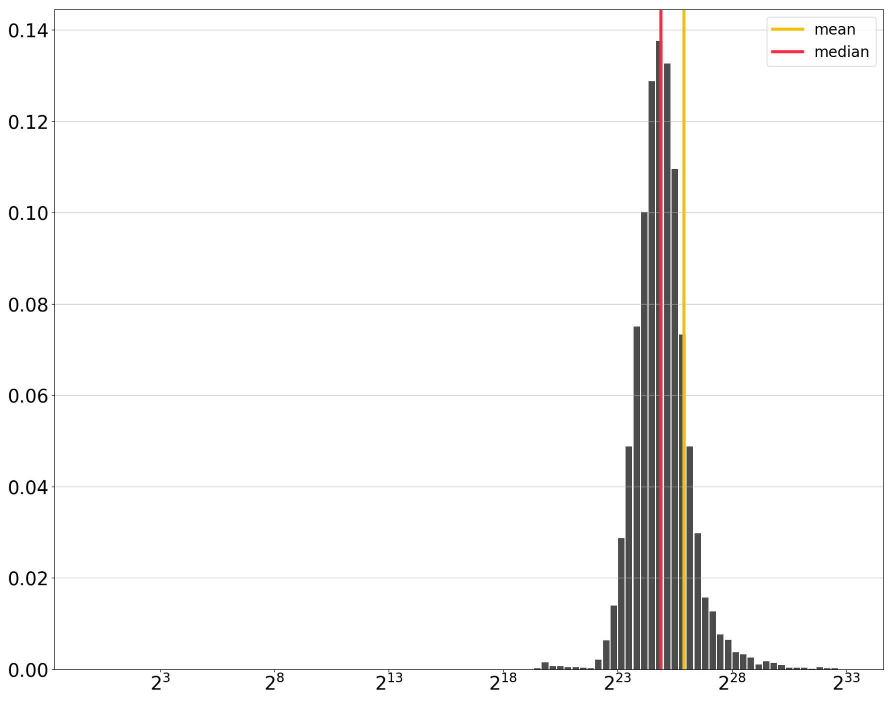


(a) Total Distribution

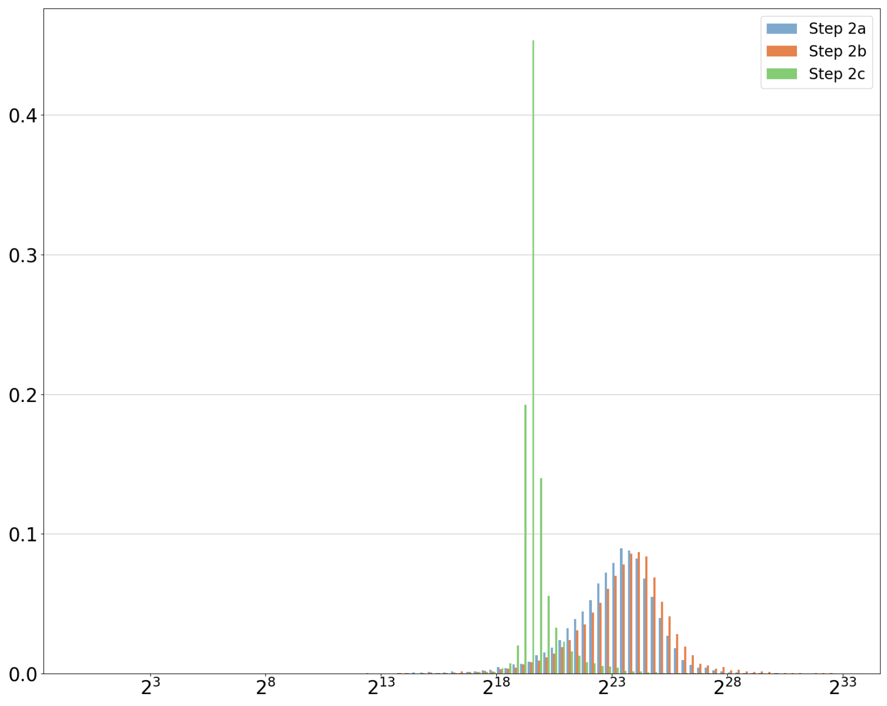


(b) Distribution of Individual Steps

Figure 3.7: Bleichenbacher's Attack – BVO – 10^4 Simulations – keylength 512 – outliers removed: 4 from Total, 3 from Step 2b

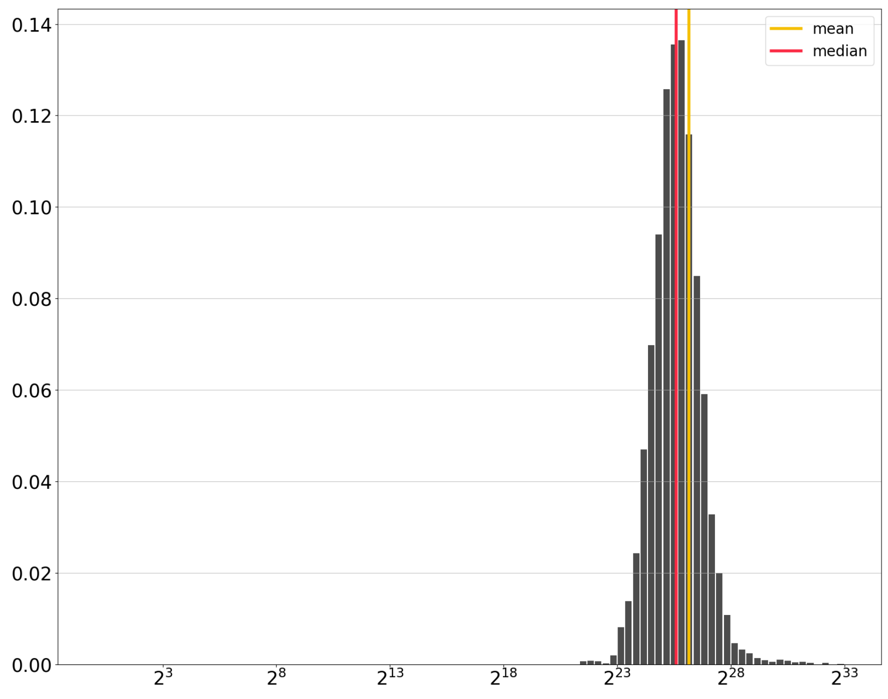


(a) Total Distribution

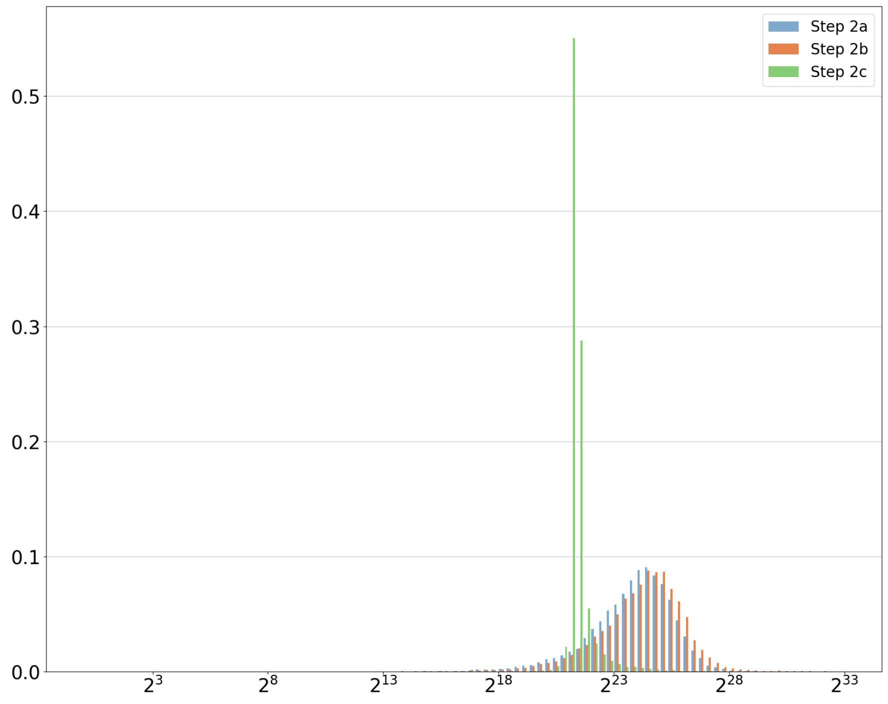


(b) Distribution of Individual Steps

Figure 3.8: Bleichenbacher's Attack – BVO – 10^4 Simulations – keylength 1024 – outliers removed: 6 from Total, 6 from Step 2b



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 3.9: Bleichenbacher's Attack – BVO – 10^4 Simulations – keylength 2048 – outliers removed: 1 from Total, 1 from Step 2b

Chapter 4

Improvements of the Attack

Over the last twenty years, following Bleichenbacher's publication [3], the attack has been extensively studied to devise methods that improve its complexity. The experimental results in Section 3.4 show that the attack complexity is usually dominated by steps 2a and 2b. As step 2a is always required, it seems that the attack takes many queries whenever step 2b has to be performed. Hence, Klíma, Pokorný and Rosa introduced two different methods for improving step 2b in 2003 [8]. These techniques are called the Beta Method and the Parallel Threads Method. As we will see from the experiments, using the Parallel Threads Method reduces the complexity of step 2b significantly, leaving step 2a as the bottleneck. Afterwards, there was a great interest in finding techniques to cope with the complexity of step 2a. In 2012, two novel techniques were introduced with this objective [2]. These methods are called Skipping Holes and Trimmers. In combination, they provide a considerable improvement. We first provide a description of the different improvements and then show the experimental results obtained by applying these optimizations.

4.1 Tighter Bounds

Bleichenbacher's attack uses the fact that we want to decrypt a previously encrypted message m. The message conforms to the PKCS #1 v1.5 standard, i.e., $m \in [2B, 3B-1]$. This observation provides the bounds for the initial interval M_0. However, the format of the message m is defined more precisely. Hence, one can specify more constrained bounds on the unknown message m, depending on the oracle type. This improvement is for free. We now provide these bounds for the different oracle types but note that they do not significantly improve the attack. Thus, we have not considered them in our implementation.

We start by looking at the bounds for the TTT, TFT and FFT oracles. As each message is padded using the PKCS #1 v 1.5 encryption standard, the first two bytes are `0x00 || 0x02`. After that, there are at least eight nonzero bytes followed by some zero byte in the remaining $k - 10$ bytes. Because of this format, we can improve the bounds on m to $[a, b]$, where:

$a = 2B + 2^{8(k-3)} + ... + 2^{8(k-10)}$ and

$b = 2B + 255 \cdot (2^{8(k-3)} + ... + 2^8) + 0 = 3B - 2^8$.

For the BVO, the message additionally has a length of 48 bytes, where the first two bytes are the `major` and `minor` version numbers. Hence, we can construct a more restrictive starting interval $[a, b]$, where:

$a = 2B + 2^{8(k-3)} + ... + 2^{8(49)} = 2B + 2^{8(49)}(2^{8(k-51)} - 1)/255$ and

$b = 2B + 255 \cdot (2^{8(k-3)} + ... + 2^{8(49)}) + 0 + 255 \cdot (2^{8(47)} + ... + 2^{8(0)}) = 3B - 255 \cdot 2^{8(48)} - 1$.

If the attacker knows the version numbers, he can make these intervals even smaller.

4.2 Beta Method

The Beta Method was presented by Klíma, Pokorný and Rosa [8]. This method is intended to improve the complexity of step 2b. However, it was introduced in the context of the Bad Version Oracle and is only applicable to FFF oracles as it uses knowledge about the message length. The technique is based on a generalization of a remark from Bleichenbacher's original paper [3]. We present the method and argue why we did not employ it in our implementation. According to Bardou et al. [2], this method does not significantly improve the attack complexity. Additionally, the Parallel Threads Method introduced in the next section performs well in practice and can be applied to all oracle types. Hence, we did not consider the Beta Method in our implementation and experimental runs.

To explain how this method works, we start by mentioning the remark in Bleichenbacher's paper on which this approach is based. For the execution of step 2b, we have assumed that the different s_i values are independent of each other. This assumption, however, may be wrong in some cases. Consider two messages m_0 and $m_0 s_i \pmod n$, which are both PKCS #1 v1.5 conforming with padding strings of similar lengths. Then, with a high probability $(2s_i - 1)m_0 \pmod n$ is also PKCS #1 v1.5 conforming. We show this by assuming that there exists some integer j such that:

$m_0 = 2 \cdot 2^{8(k-2)} + 2^{8j} PS + D$ and

$m_0 s_i \pmod n = 2 \cdot 2^{8(k-2)} + 2^{8j} PS' + D'$.

It follows that:

$(2s_i - 1)m_0 \text{ (mod } n) = 2 \cdot 2^{8(k-2)} + 2^{8j}(2PS' - PS) + 2D' - D.$

This message is often PKCS #1 v1.5 conforming too. We can see that this remark applies whenever the message length, and hence the padding size, is fixed. If this is the case, we can possibly find a new suitable multiplier s_{i+1} more efficiently.

The Beta Method generalizes this observation:

Lemma 4.1 *Suppose we have two ciphertexts c_i and c_j, such that $c_i = (s_i)^e c_0$ (mod n), $c_j = (s_j)^e c_0$ (mod n), and both c_i and c_j are S-PKCS conforming. That is, we can write $m_i = c_i^d$ (mod n) = $2B + 2^{8 \cdot 49} PS_i + D_i$ and $m_j = c_j^d$ (mod n) = $2B + 2^{8 \cdot 49} PS_j + D_j$, where $0 < PS_{i,j} < 2^{8(k-51)}$ and $0 \le D_{i,j} < 2^{8 \cdot 48}$. Then for $c = s^e c_0$ (mod n) and $\beta \in Z$, where $s = [(1-\beta)s_i + \beta s_j]$ (mod n), it holds that c^d (mod n) $= m$, such that $m = [2B + 2^{8 \cdot 49}((1-\beta)PS_i + \beta PS_j) + (1-\beta)D_i + \beta D_j]$ (mod n).*

Proof We use the observation that c^d (mod n) $= (s^e c_0)^d$ (mod n) $= sm_0$ (mod n) = $[(1-\beta)s_i + \beta s_j]m_0$ (mod n) $= [(1-\beta)s_i m_0 + \beta s_j m_0]$ (mod n) $=$ $[(1-\beta)m_i + \beta m_j]$ (mod n) $= m$, where $m_0 = c_0^d$ (mod n). □

It follows that once we have found suitable multipliers s_i, s_j for a ciphertext c_0, we can try linear combinations of s_i and s_j to find another suitable multiplier s. This method is intended to improve the efficiency of step 2b because when searching for s_i with $i > 1$, we have already found at least two suitable multipliers s_0 and s_1. We now want to show that there exists a value of β for each triplet of suitable multipliers (s_i, s_j, s). Since the modulus n is the product of two primes, we can assume that gcd($s_j - s_i$, n) $= 1$. Using Bézout's Lemma [6], there exist some values u and $v \in Z$ such that $u(s_j - s_i) + vn = 1$ and hence $u(s_j - s_i) + vn$ (mod n) $= u(s_j - s_i)$ (mod n) $= 1$. We can derive $u(s_j - s_i) \cdot (s - s_i)$ (mod n) $= (s - s_i)$ (mod n). By defining $\beta = u(s - s_i)$ (mod n), we have $\beta(s_j - s_i)$ (mod n) $= (s - s_i)$ (mod n). Finally, $\beta s_j - (\beta - 1)s_i$ (mod n) $= s$ (mod n). Therefore, we can conclude that we can find any suitable multiplier s with this method when trying the corresponding value for β.

In [8], Klíma, Pokorný and Rosa specify that in practice, one can try small positive and negative values of β and test whether the resulting linear combination s produces a valid ciphertext. However, their experiments showed differences in how much information can be obtained from the new s depending on the size of β. Values of β close to $n/2$ were able to reduce the size of M_i faster than small values of β. They speculate that this could be due to a linear dependency on Z, which is stronger for small β. This introduces another challenge, namely finding a "good" β. Finding a β close to $n/2$ is not easy as a brute force search would require too many oracle calls. Even if we could find a β of this size, the resulting s value may be very large. This value might not work with our algorithm as it induces a large interval for r in step 3. Hence, it is desired to extract as much information as possible by trying small values of β.

Another important point is that when using this method with negative values of β, assuming that $s_i < s_j$, we may get a value for s close to n (a small negative value modulo n). This s again does not work with our algorithm. However, because of symmetry, we are still able to process these values by using the following observation:

Lemma 4.2 *Suppose we have integers s, m_0, and n satisfying $a \leq m_0 s \pmod n \leq b$, where $a, b \in Z$. Then, the integer $v = n - s$ satisfies $a' \leq m_0 v \pmod n \leq b'$, where $a' = n - b$ and $b' = n - a$.*

Proof We can see that $m_0 v \pmod n = m_0(n - s) \pmod n = -m_0 s \pmod n = n - m_0 s \pmod n$. As the upper bound on $m_0 s \pmod n$ is b, the lower bound on $m_0 v \pmod n$ is $a' = n - b$. We can determine the upper bound b' of $m_0 v \pmod n$ in the same way, obtaining $b' = n - a$. $\square$

This symmetry can be used in practice to allow testing negative values for β. If the resulting s value is close to n, we transform it to $v = n - s$ and update the bounds to a' and b'. Otherwise, the algorithm stays the same.

4.3 Parallel Threads Method

4.3.1 Description

The Parallel Threads Method was the second improvement presented by Klíma, Pokorný and Rosa [8] to cope with the complexity of step 2b. Recall that after step 2a the attacker either proceeds with step 2b or 2c depending on the number of intervals in M_1. In the experimental results in Section 3.4, we have seen that performing step 2b for finding s_2 is expensive. On the other hand, if we can directly continue with step 2c, we are able to use an efficient method for finding the next conforming ciphertext. Additionally, the experiments showed that even if $|M_1| > 1$, there is usually only a small number of intervals in this set. This was also observed by Klíma, Pokorný and Rosa during their experiments. These observations intuitively suggest starting an individual thread for each interval in M_1 and performing step 2c on it. Because one of these intervals has to contain m_0, we will always be able to find a value for s_2. However, when searching in a wrong interval, we may never find a conforming multiplier s_2. In practice, it is usually not possible to parallelize these threads. Therefore, they are arranged in a cycle iterating through them by doing one oracle call a time for each interval. To describe this more precisely: If M_1 contains t intervals, we start a thread T_i executing 2c for each interval $[a_i, b_i]$ for $1 \leq i \leq t$. We begin by performing a single oracle call with T_1. If we do not succeed, we continue doing one step with thread T_2 and so on. As soon as we find a value s_2 that results in an accepted message, we update all intervals in the same way as for the initial algorithm. This could result in some empty intervals. In that case, the corresponding thread gets eliminated and the intervals renumbered. For the new set of intervals, we again start with T_1 and do the same procedure as before, now searching for s_3. The Parallel Threads Method gets continued until we end up with a single interval. If there is only one interval left, we continue by doing step 2c on this interval.

4.3.2 Heuristic for Parallel Threads Method

Klíma, Pokorný and Rosa [8] presented a heuristic to decide whether to start the Parallel Threads Method. They set a bound on the number of intervals in M_{i-1} such that the method provides an improvement to the attack complexity. Let ϵ be the number of rounds of step 2b we have to perform with the Parallel Threads Method to end up with a single interval. Hereby, one round of step 2b corresponds to finding an s_i value that works and updating all intervals. The Parallel Threads Method is started in step i if the following inequality holds:

$$|M_{i-1}| < (2\epsilon \Pr(A))^{-1} + 1.$$

This bound can be explained in the following way: Let us approximate the number of oracle calls required for step 2b, if it has to be performed, by $1/\Pr(P)$ (see theoretical analysis in Section 3.3). Additionally, we expect to find the next s_i value in step 2c after $2/\Pr(P \mid A)$ oracle calls. For $|M_{i-1}|$ intervals, we hence expect to find s_i after $\frac{2|M_{i-1}|}{\Pr(P|A)}$ oracle calls. This follows as we perform one oracle call per interval in a round-robin fashion and only expect to find a conforming multiplier s_i when performing step 2c on the interval containing m_0. Because we want to improve the complexity of the attack and considering that we need to find ϵ conforming multipliers s_i to finish the Parallel Threads Method, we get the constraint: $\epsilon \cdot \frac{2|M_{i-1}|}{\Pr(P|A)} \leq \frac{1}{\Pr(P)} = \frac{1}{\Pr(P|A) \cdot \Pr(A)}$. This results in: $|M_{i-1}| < (2\epsilon \Pr(A))^{-1} + 1$.

In [8], the authors tested this method only with the BVO. For their experiments, they used $\epsilon = 2$ as it was the ceiling of the mean value observed for ϵ. Using that $1/\Pr(A) < 2^{16}$ gives us a bound of roughly 16 000. We note that in our experiments, the ceiling of ϵ for the BVO was also 2. For the other oracle types, the value of ϵ is lower in practice, leading to an even higher bound on $|M_{i-1}|$ for which the Parallel Threads Method is expected to improve the attack performance.

During the experiments for the unmodified Bleichenbacher attack, we observed that especially for the TTT, TFT and FFT oracles, only a small number of intervals were present after step 2a. The reason is that s_1 never gets too large for these oracle types and the number of intervals depends on the size of s_1. Hence, for these oracles, we always want to use the Parallel Threads Method. This behavior is captured by the heuristic as it gives a large bound on $|M_{i-1}|$. For the BVO, it is harder to find s_1 and therefore the number of intervals in M_1 is typically a lot higher. However, there are only in rare extreme cases over 10 000 intervals in M_1. This implies that it always makes sense to employ the Parallel Threads Method, except for rare cases for the BVO where the unmodified step 2b is more efficient. This is exactly the aim of the heuristic.

As we can see in the next section, the Parallel Threads Method decreases the complexity of step 2b significantly.

4.3.3 Experimental Results

We carried out experiments with different bounds on the number of intervals in M_1 such that the Parallel Threads Method is started. Bardou et al. [2] used the Parallel Threads Method for $|M_{i-1}| \leq 40$. Considering this and the value provided by the heuristic in Section 4.3.2, we tested bounds of 40, 1 000 and 16 000. For the TTT, TFT and FFT oracles, there was no notable difference between a cutoff value of 40 and 1 000. This follows as the number of intervals is usually very low for these oracle types. However, a higher bound is better because we still want to perform the Parallel Threads Method in a rare case where $|M_1| = 100$. On the other hand, for the BVO, a cutoff value of 1 000 is significantly better than 40. The difference between a bound of 1 000 and 16 000 could only be observed for a keylength of 2048 bits. There, as expected, the bound of 16 000 performed better. Overall, the results of these experiments confirmed the heuristic.

Following these observations, in our implementation, we start the Parallel Threads Method if $M_1 \leq 16\,000$. Note that we start the method after step 2a and do not check whether to start the method after each step i as described in [8]. This is reasonable because our experiments showed that the number of intervals in M_i only decreases as i increases. Since the bound of 16 000 is based on a heuristic, one could possibly find a better bound for the BVO experimentally or by considering the complexity results of the original algorithm in practice.

Now, we present the results of the experiments for the Parallel Threads Method. Firstly, the complexity of step 2a is equal to the original attack as it has not been modified.

Step 2b is now defined as the number of oracle calls required for the Parallel Threads Method until we end up with a single interval. Furthermore, the number of rounds required of step 2b with the Parallel Threads Method is still the number of s_i we need to find to get down to a single interval. As soon as only one interval is left, the usual step 2c is performed on this interval. Overall, this method improves step 2b significantly, shifting the main complexity of the attack to step 2a. This is the case as the Parallel Threads Method uses step 2c in each thread and step 2c finds an s_i quickly when operating on the correct interval.

How much the median and mean of the total complexity are reduced depends on the oracle type. In particular, this depends on how often we have to perform step 2b in expectation for the oracle and keylength combination. Hence, we can see large improvements in the total mean and median values for the BVO and the FFT oracle, whereas the median values stay the same for the TTT and TFT oracles. However, the total mean values also decrease for the TTT and TFT values as extreme values influence the mean and having to perform step 2b is a bad case for these oracle types.

Interestingly, the Parallel Threads Method also reduces the mean of step 2c. We suppose that this is the case as we find larger values for s_i with this method compared to the original step 2b, where we search incrementally. Thus, we are able to trim the interval slightly more in step 3 of the algorithm, improving some of the cases where step 2c takes unexpectedly long. These larger values for s_i also explain why we require fewer rounds of step 2b with the Parallel Threads Method in the mean to end up with a single interval as we produce smaller intervals.

In the tables we provide, we compare the complexity of the attack with the Parallel Threads Method to the complexity of the “original” Bleichenbacher attack.

TTT Oracle

For the TTT oracle, the experiments have shown that if we have to perform step 2b, it is now essentially for free, see Table 4.2. This is also visible in Figure 4.1, as the number of oracle calls required for step 2b is now always very low. However, the median of the total complexity has not improved because step 2b does not have to be performed most of the time, see Table 4.1. The overall complexity is still determined by step 2a. But, because step 2b is a rare but expensive event for the TTT oracle, this method improves the mean of the total complexity, see Table 4.1. We mention that the mean number of rounds required for step 2b went down from 0.21 to 0.18 with this optimization. Also, the mean complexity of step 2c could be slightly reduced, see Table 4.3.

We only provide the plot for a modulus size of 512 bits, see Figure 4.1, as the improvement on the complexity of step 2b looks identical for the other keylengths.

keylength	median total	mean total	original median total	original mean total
512	21 142	30 294	21 114	36 272
1024	23 048	33 626	23 000	40 815
2048	26 474	39 417	26 439	49 767

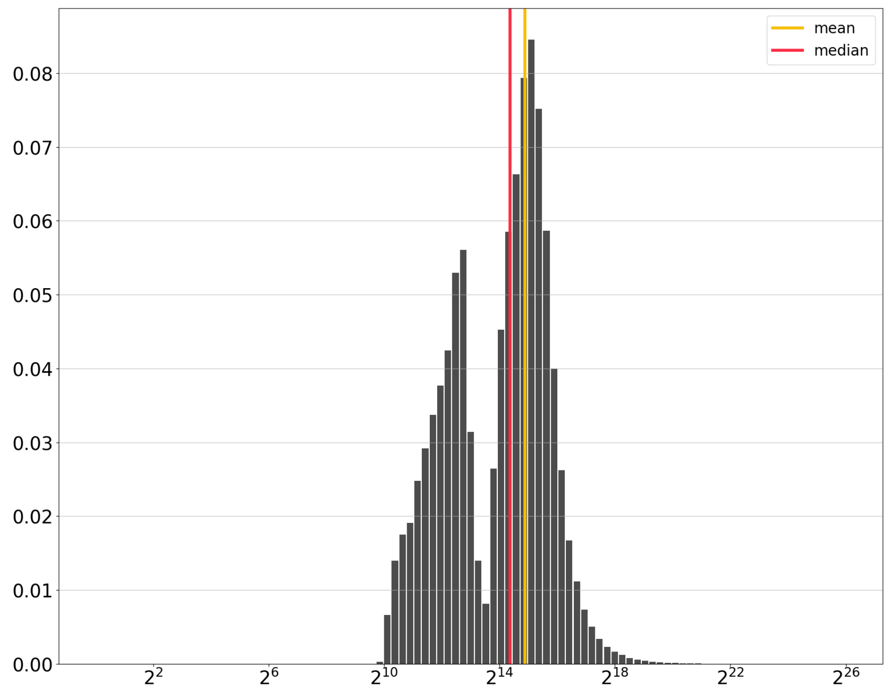
Table 4.1: Bleichenbacher's Attack with Parallel Threads Method – TTT Oracle – 10^6 Simulations – Total

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	2.63	0	5 634
1024	0	4.02	0	5 675
2048	0	2.89	0	5 829

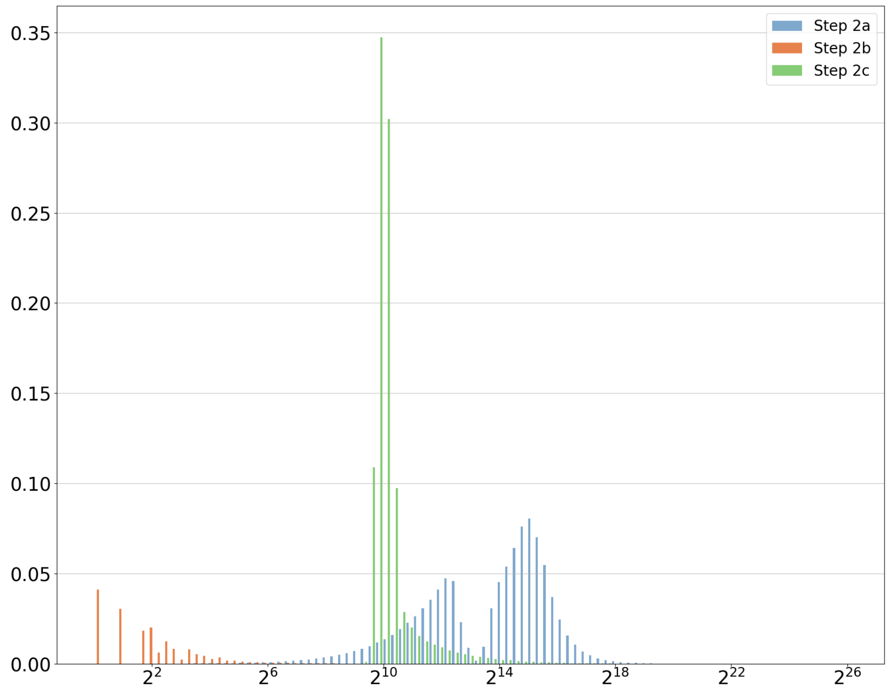
Table 4.2: Bleichenbacher's Attack with Parallel Threads Method – TTT Oracle – 10^6 Simulations – Step 2b

keylength	median 2c	mean 2c	original median 2c	original mean 2c
512	1 030	3 038	1 030	3 458
1024	2 126	6 174	2 126	7 779
2048	4 318	11 444	4 320	16 097

Table 4.3: Bleichenbacher's Attack with Parallel Threads Method – TTT Oracle – 10^6 Simulations – Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.1: Bleichenbacher's Attack with Parallel Threads Method – TTT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 2 from Total, 2 from Step 2c

TFT Oracle

All observations are the same as for the TTT oracle. For this oracle, the mean number of rounds required for step 2b could be reduced from 0.23 to 0.20.

We do not provide any figures for this oracle type as the improvement of step 2b looks identical to the TTT oracle with a keylength of 512 bits, see Figure 4.1.

keylength	median total	mean total	original median total	original mean total
512	22 091	31 888	22 077	39 530
1024	23 920	35 041	24 010	42 981
2048	27 519	41 426	27 486	51 362

Table 4.4: Bleichenbacher's Attack with Parallel Threads Method – TFT Oracle – 10^6 Simulations – Total

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	6.68	0	6 969
1024	0	5.32	0	7 000
2048	0	4.27	0	6 911

Table 4.5: Bleichenbacher's Attack with Parallel Threads Method – TFT Oracle – 10^6 Simulations – Step 2b

keylength	median 2c	mean 2c	original median 2c	original mean 2c
512	1 071	2 956	1 071	3 759
1024	2 214	6 048	2 213	6 964
2048	4 496	11 839	4 497	15 030

Table 4.6: Bleichenbacher's Attack with Parallel Threads Method – TFT Oracle – 10^6 Simulations – Step 2c

FFT Oracle

For the FFT oracle, both the median and mean of the total number of required oracle calls are improved by applying the Parallel Threads Method as we have to perform step 2b most of the time, see Table 4.7. We also observe that with growing keylength, the influence of the improvement shrinks as the attack becomes easier. The other observations are identical to the TTT oracle. We note that the median number of rounds of step 2b stayed the same for all keylengths, i.e., 1 for keylengths of 512 and 1024 bits and 0 for a keylength of 2048 bits. The mean number of rounds required of step 2b was decreased by roughly 0.03 for each keylength, as for the TTT and TFT oracles.

Figures 4.2, 4.3 and 4.4 show the distributions of the total number of oracle calls and the oracle calls for the individual steps required for the Bleichenbacher attack with the Parallel Threads Method applied for the different keylengths.

keylength	median total	mean total	original median total	original mean total
512	151 822	222 568	359 198	434 687
1024	76 510	113 759	153 859	200 970
2048	46 787	71 346	50 191	109 099

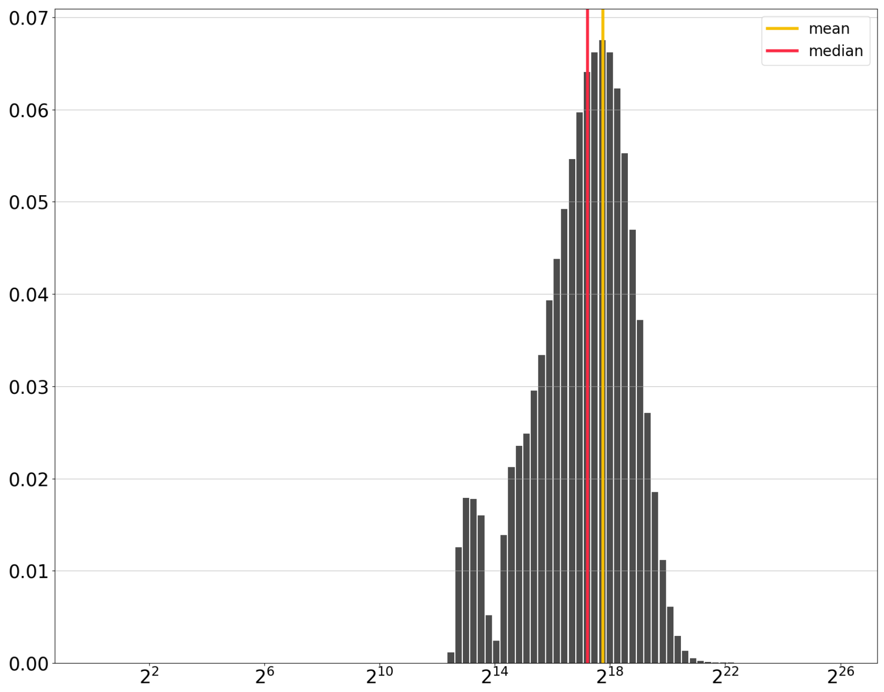
Table 4.7: Bleichenbacher's Attack with Parallel Threads Method – FFT Oracle – 10^6 Simulations – Total

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	45	132	121 714	212 733
1024	6	33	32 483	84 762
2048	0	15	0	33 371

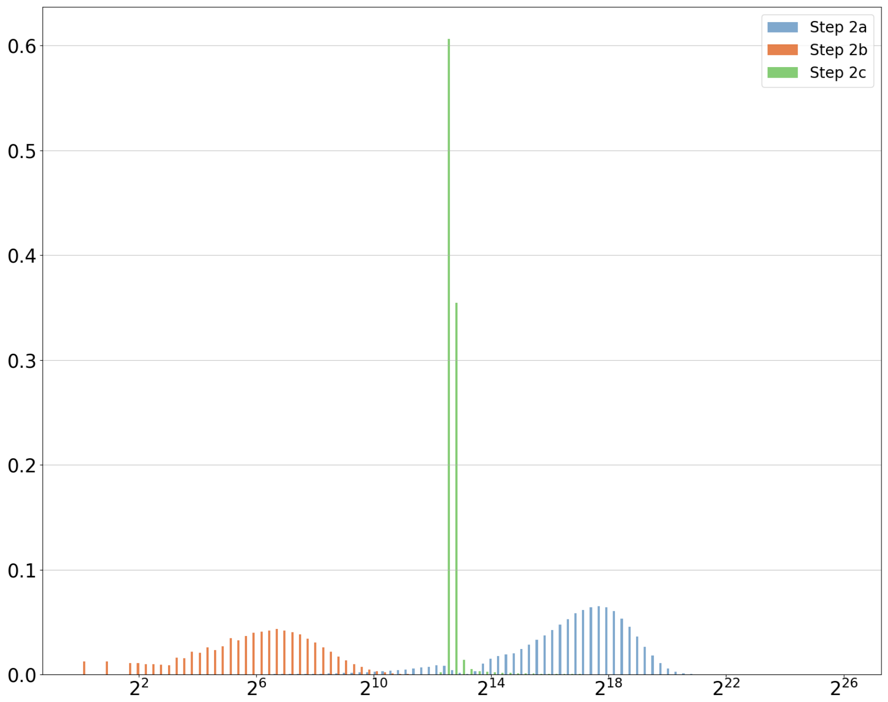
Table 4.8: Bleichenbacher's Attack with Parallel Threads Method – FFT Oracle – 10^6 Simulations – Step 2b

keylength	median 2c	mean 2c	original median 2c	original mean 2c
512	6 128	7 947	6 134	8 097
1024	6 392	10 080	6 394	12 593
2048	7 544	14 256	7 545	18 846

Table 4.9: Bleichenbacher's Attack with Parallel Threads Method – FFT Oracle – 10^6 Simulations – Step 2c

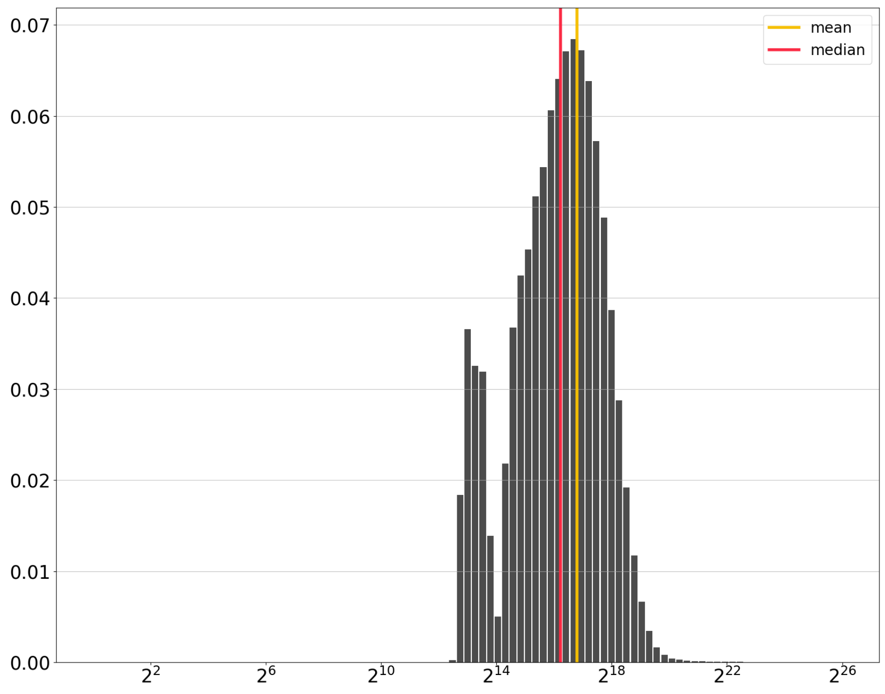


(a) Total Distribution

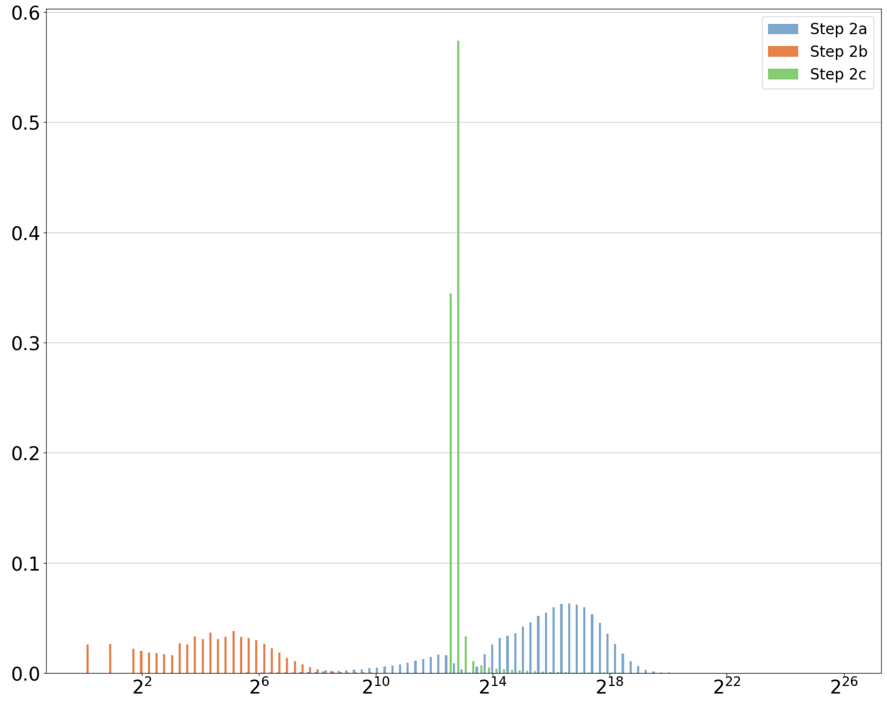


(b) Distribution of Individual Steps

Figure 4.2: Bleichenbacher's Attack with Parallel Threads Method – FFT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 7 from Total, 7 from Step 2c

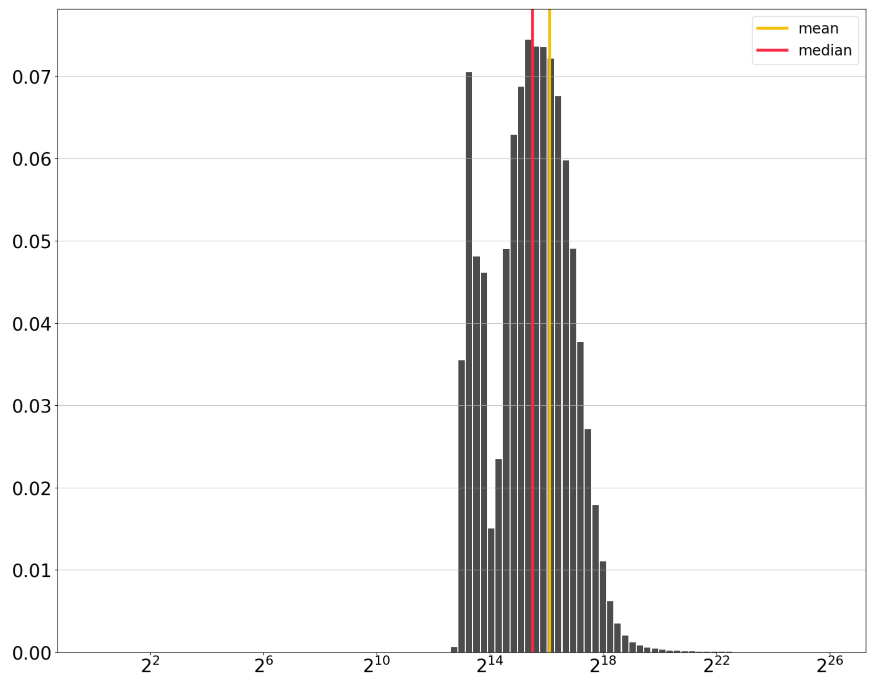


(a) Total Distribution

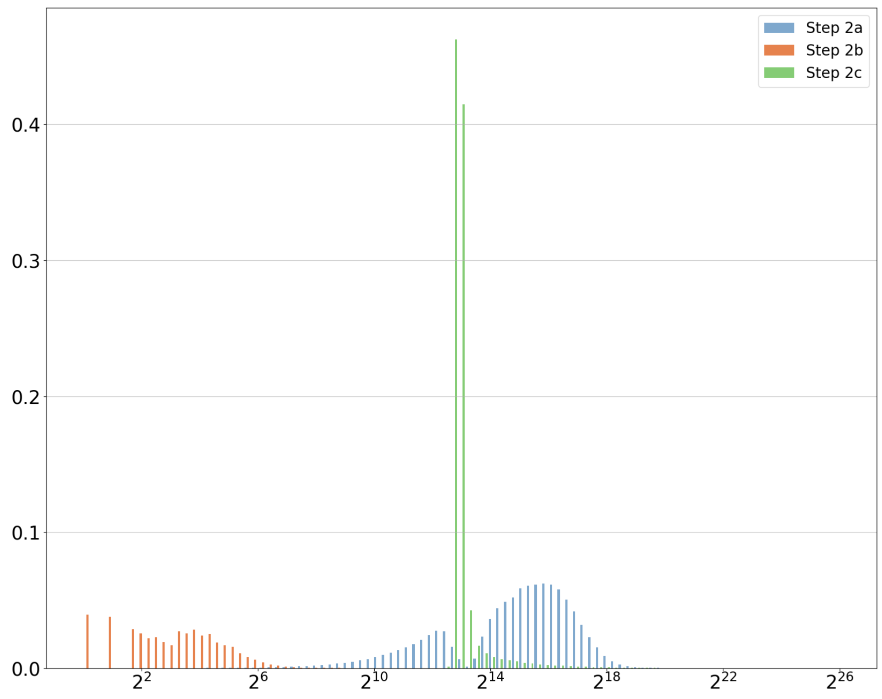


(b) Distribution of Individual Steps

Figure 4.3: Bleichenbacher's Attack with Parallel Threads Method – FFT Oracle – 10^6 Simulations – keylength 1024 – outliers removed: 6 from Total, 1 from Step 2a, 5 from Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.4: Bleichenbacher's Attack with Parallel Threads Method – FFT Oracle – 10^6 Simulations – keylength 2048 – outliers removed: 8 from Total, 8 from Step 2c

Bad Version Oracle

For the BVO, there is a notable improvement in the median and mean of the total required oracle calls, see Table 4.10. This results from the fact that step 2b almost always has to be performed for this oracle type and the Parallel Threads Method lowers the complexity of step 2b significantly, see Table 4.11. We can also see that with growing keylength, the influence of the improvement on the total median and mean values increases as the attack becomes more difficult.

For this oracle type, the median number of rounds required of step 2b stayed at 1 for all keylengths. However, the mean number of rounds of step 2b for a keylength of 512 bits was reduced from 2.68 to 1.28, for a keylength of 1024 bits from 2.48 to 1.27 and for a keylength of 2048 bits from 1.78 to 1.24. We note that the variations of the median number of oracle calls for step 2c in Table 4.12 arise from the randomness of the experiments as we only performed 10 000 simulations for the BVO. Furthermore, the Parallel Threads Method also reduces the mean of step 2c for the BVO, as visible in Table 4.12.

We only include the figure visualizing the improvement for a keylength of 512 bits as the influence of the method looks similar for the different keylengths.

keylength	median total	mean total	original median total	original mean total
512	8 928 141	17 141 061	24 227 787	50 146 387
1024	11 392 567	19 728 955	31 076 365	62 343 618
2048	22 487 282	34 072 121	51 200 904	75 265 298

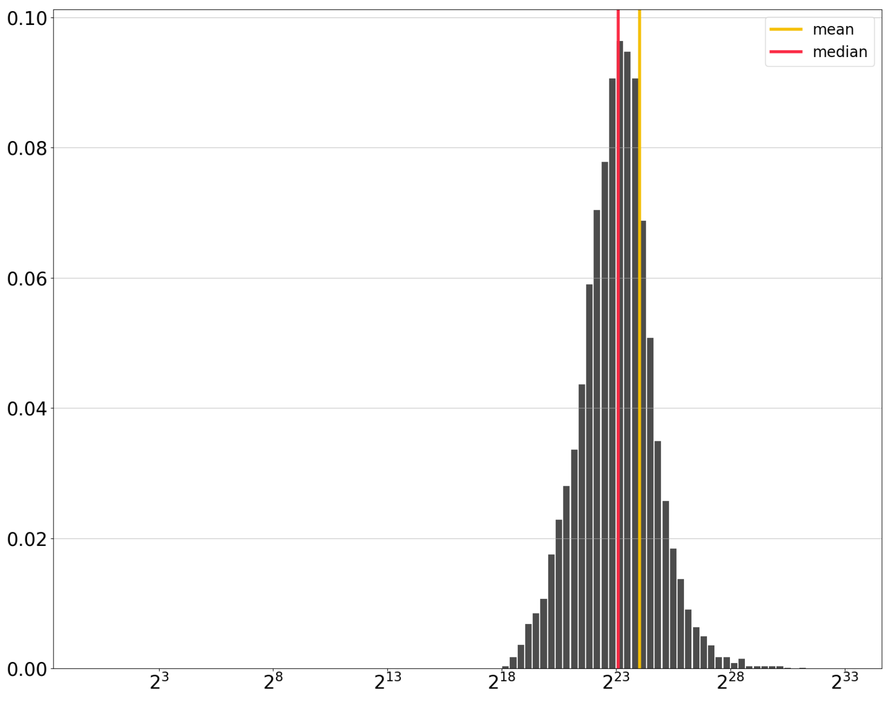
Table 4.10: Bleichenbacher's Attack with Parallel Threads Method – BVO – 10^4 Simulations – Total

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	149 303	1 606 714	11 113 772	34 805 695
1024	219 051	1 486 774	13 778 521	42 711 607
2048	610 865	1 589 241	22 443 601	44 014 214

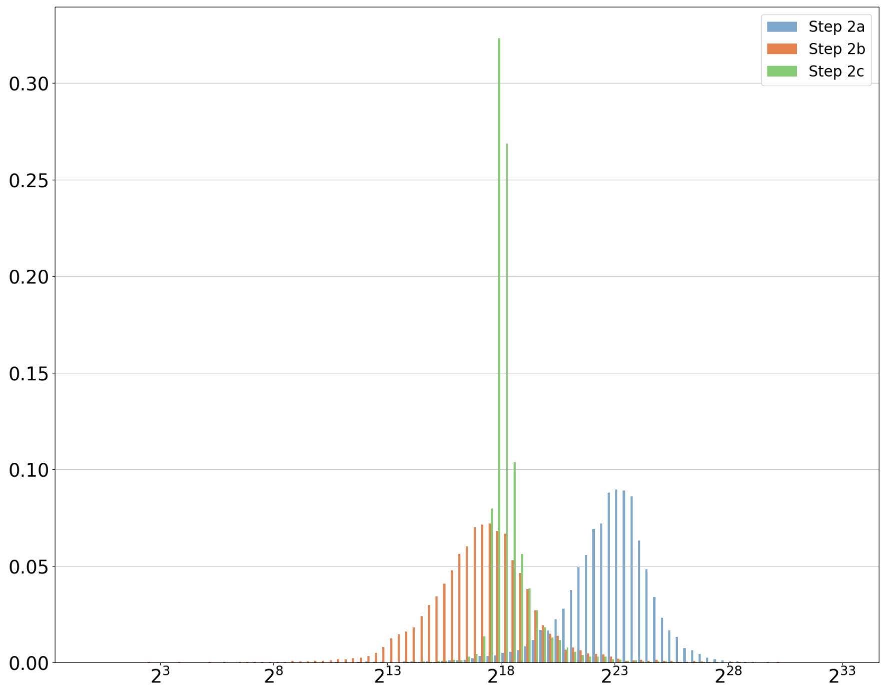
Table 4.11: Bleichenbacher's Attack with Parallel Threads Method – BVO – 10^4 Simulations – Step 2b

keylength	median 2c	mean 2c	original median 2c	original mean 2c
512	273 782	673 912	274 367	841 390
1024	753 824	1 229 558	742 132	1 400 068
2048	2 589 309	3 277 990	2 583 223	3 706 822

Table 4.12: Bleichenbacher's Attack with Parallel Threads Method – BVO – 10^4 Simulations – Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.5: Bleichenbacher's Attack with Parallel Threads Method – BVO – 10^4 Simulations – keylength 512

4.4 Skipping Holes

4.4.1 Description

In the experimental results in Section 4.3.3, we have seen that the Parallel Threads Method significantly improves step 2b. It follows that step 2a now usually determines the complexity of the attack. The Skipping Holes technique, presented by Bardou et al. [2], aims at improving this step. Like the tighter bounds, this improvement is for free.

The intuition behind the method is to make the search for s_1 faster by skipping over ranges of values for which we know $m_0 s_1 \pmod{n}$ is never accepted by the oracle. As a first step, the starting value for s_1 can be optimized by noticing that we require $n + 2B \leq m_0 s_1$ for a conforming message. In combination with $m_0 \leq 3B - 1$, we obtain that we can start with

$$s_1 = \lceil (n + 2B)/(3B - 1) \rceil.$$

This is a slightly better starting value for s_1 compared to $s_1 = \lceil n/3B \rceil$ introduced by Bleichenbacher [3]. The observation can also be used when better bounds on m_0 are known. If we can bound m_0 from above by b, we set the starting value to

$$s_1 = \lceil (n + 2B)/b \rceil.$$

This is relevant in combination with the improvement presented in the next section of trimming the interval M_0 before the algorithm is started.

Similarly to this observation, we know that for each value s_i such that $m_0 s_i \pmod{n}$ is accepted by an oracle $2B \leq m_0 s_i \pmod{n} \leq 3B - 1$ holds. This is equal to saying that $2B \leq m_0 s_i - jn \leq 3B - 1$ for some natural number j. Hence, we obtain a list of intervals in which s_i can lie:

$$\frac{2B + jn}{3B - 1} \leq s_i \leq \frac{3B - 1 + jn}{2B}$$

for $j \in N$. At the same time, this also provides us with a list of intervals where a suitable multiplier s_i can never lie, namely when:

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

for a natural number j. One can observe that with the initial bounds on m_0, i.e., $2B \leq m_0 \leq 3B - 1$, we can only identify one "hole". For $j = 1$, we can skip roughly $\frac{n}{6B}$ values. However, for $j > 1$, we cannot identify any more intervals where m_0 certainly does not lie because the inequality $\frac{2B+jn}{3B-1} \leq s_i \leq \frac{3B-1+jn}{2B}$ allows all possible s_i values. Note that this first hole could also be observed in the experimental results, especially for the TTT oracle, see Step 2a in Figures 3.1, 3.2 and 3.3.

In combination with the Trimmers method [2] presented in the next section, we can potentially find a list of such holes, depending on how successful the trimming step was. If we were able to trim the interval M_0 to $[a, b]$, we know that for a conforming s_i, it holds that

$$\frac{2B + jn}{b} \leq s_i \leq \frac{3B - 1 + jn}{a}$$

and a hole exists if

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

for a natural number j.

Having identified all possible holes, we can then skip these to make the search for s_i more efficient.

We extend the Skipping Holes technique with another fact based on an observation from the experimental results. If we have to do a round of step 2b, we have to do at least n/m_0 calls until we find the next conforming multiplier s_i, see for example Figure 3.1. Here, we assume the initial step 2b and not step 2b optimized with the Parallel Threads Method. Recall that s_i is a suitable multiplier if $2B \leq m_0 s_i - jn \leq 3B - 1$ for some natural number j. If we increment s_i by one in each step of 2b, s_{i+1} corresponds to the first value such that $2B \leq m_0 s_{i+1} - j'n \leq 3B - 1$ for some $j' \geq j + 1$. Hence, s_{i+1} is at least n/m_0 larger than s_i. Furthermore, the number of oracle calls required to find s_{i+1} is roughly $r \cdot n/m_0$ for some natural number r. Let us define a_{min} as $\min\{a \mid [a, b] \in M_i\}$ and b_{max} as $\max\{b \mid [a, b] \in M_i\}$. We bound n/m_0 by n/b_{max} and initially set

$$s_{i+1} = s_i + \lceil n/b_{max} \rceil.$$

Additionally, we can also possibly identify a new hole to skip using a_{min} and b_{max}. However, this is not relevant in practice as the Parallel Threads Method decreases the complexity of step 2b a lot more. We still applied this approach to our implementation to see how it influences the distribution of oracle calls required for step 2b.

Now, we look at how the Skipping Holes improvement performs on its own. This means we can only identify and skip the first hole for step 2a and we also skip values for step 2b whenever a round of step 2b is required.

4.4.2 Experimental Results

This method improves the complexity of steps 2a and 2b for all oracle types as it is for free and skips ranges of invalid values for s_i. Therefore, this method should always be applied. We compare the complexity of steps 2a and 2b with the Skipping Holes optimization to the original algorithm. We note that this technique does not influence step 2c as it finds the same s_i values in steps 2a and 2b as Bleichenbacher's original algorithm. The only difference is that we find these values more efficiently.

The Skipping Holes method lowers the number of required oracle calls for step 2a whenever the value for s_1 is beyond the first hole. Additionally, we identify the hole independently of the oracle type. This means we skip the same range of values for all oracle types. Hence, because the median number of oracle calls required for step 2a is beyond the first hole for all oracles, we observe that the median is always reduced by roughly the same value. However, how much we can improve the mean of step 2a depends on how many times this improvement is used. It follows that for the TTT and TFT oracles, the mean values for step 2a increase less than for the FFT oracle and BVO. The same holds for the mean value of step 2b. However, the total complexity of the FFT oracle and BVO is a lot larger than for the more permissive oracles. Therefore, this improvement is not able to decrease their total complexity significantly. Furthermore, for the BVO, the experimental results were dominated by the randomness of the simulations and there was no noticeable difference in the performance because of the immense complexity required for this oracle.

TTT Oracle

In Table 4.14, we can see that the median of step 2a can be improved significantly with the Skipping Holes technique. Because there are many messages for which the number of oracle calls required for finding s_1 is below the first identified hole, the mean is reduced less. For the TTT oracle, it is unlikely that one has to perform step 2b. Hence, the median value for step 2b stays the same, see Table 4.15. However, as step 2b is costly, whenever it is done, its mean is decreased notably. Overall, the total mean and median can be lowered significantly with this method, see Table 4.13.

In Figure 4.6, we can see that the hole for step 2a, which was previously present in Figure 3.1, is closed. Additionally, the number of steps required for 2b is not always above n/m_0 anymore. We also observe in the figure that for 5% of the experiments our improvement for step 2b finds the right s_2 value immediately.

We only show the plot for a keylength of 512 bits as the improvement looks identical for the other keylengths.

keylength	median total	mean total	original median total	original mean total
512	14 746	28 937	21 114	36 272
1024	16 741	35 314	23 000	40 815
2048	20 230	41 225	26 439	49 767

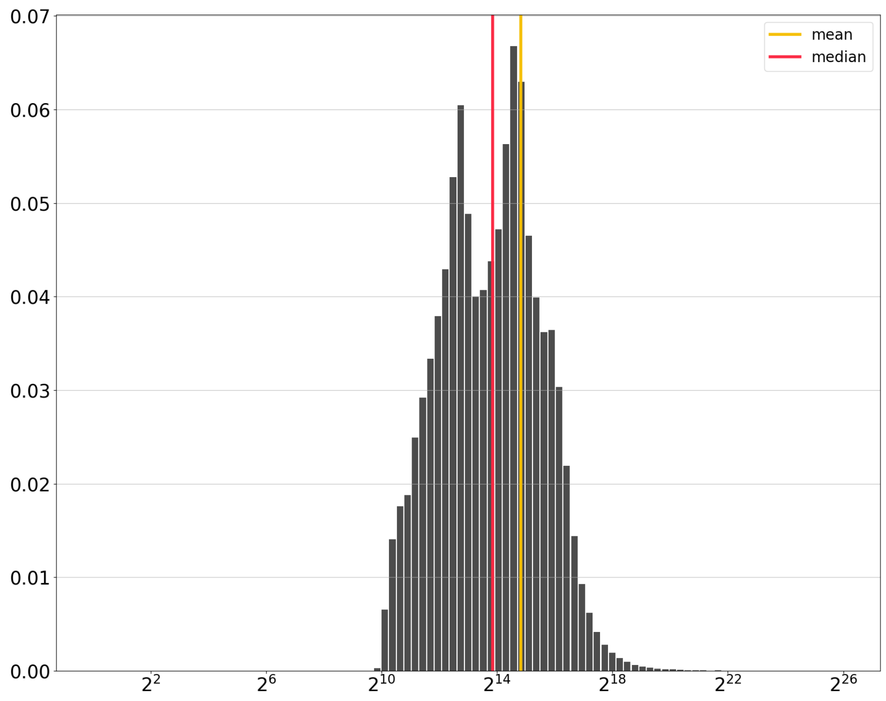
Table 4.13: Bleichenbacher's Attack with Skipping Holes – TTT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	12 417	23 121	19 178	27 180
1024	12 497	23 236	19 318	27 361
2048	12 699	23 900	19 642	27 841

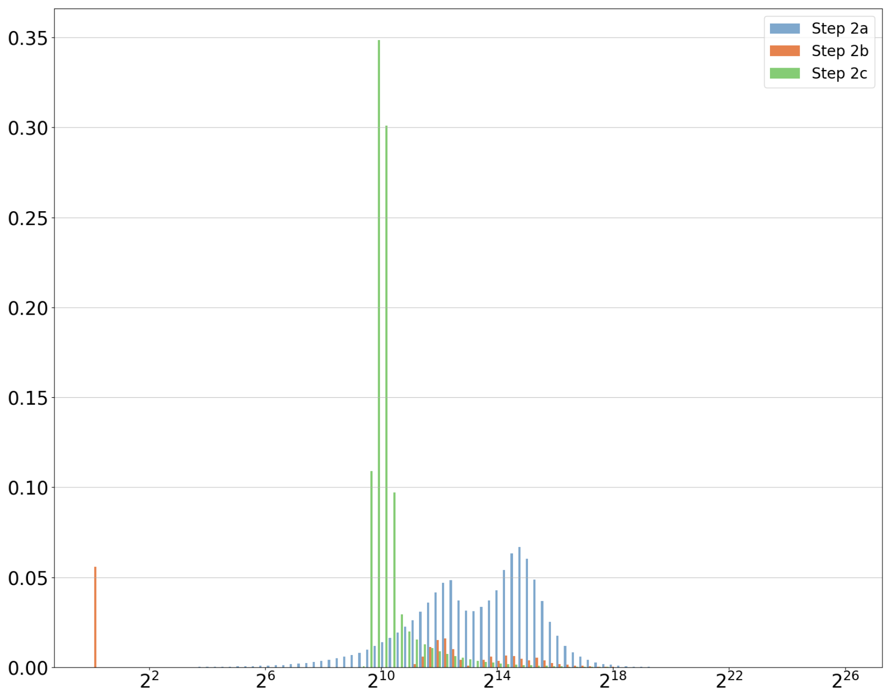
Table 4.14: Bleichenbacher's Attack with Skipping Holes – TTT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	2 642	0	5 634
1024	0	2 617	0	5 675
2048	0	2 658	0	5 829

Table 4.15: Bleichenbacher's Attack with Skipping Holes – TTT Oracle – 10^6 Simulations – Step 2b



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.6: Bleichenbacher's Attack with Skipping Holes – TTT Oracle – 10^6 Simulations – keylength 512

TFT Oracle

The observations are the same as for the TTT oracle. The effect on the distributions of the oracle calls also looks identical. Hence, the figures are omitted.

keylength	median total	mean total	original median total	original mean total
512	15 658	32 854	22 077	39 530
1024	17 637	37 243	24 010	42 981
2048	21 143	42 475	27 486	51 362

Table 4.16: Bleichenbacher's Attack with Skipping Holes – TFT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	13 373	24 656	20 115	28 801
1024	13 392	24 796	20 259	29 017
2048	13 553	25 175	20 517	29 421

Table 4.17: Bleichenbacher's Attack with Skipping Holes – TFT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	3 904	0	6 969
1024	0	3 816	0	7 000
2048	0	3 970	0	6 911

Table 4.18: Bleichenbacher's Attack with Skipping Holes – TFT Oracle – 10^6 Simulations – Step 2b

FFT Oracle

For the FFT oracle, the total improvement on the median of step 2a is equal to the improvement for the TTT and TFT oracles, see Table 4.20. We observe that the mean value for step 2a is reduced more compared to the more permissive oracles. This can be explained by the fact that for the FFT oracle, the number of oracle calls required for step 2a is more often above the identified hole skipped with this optimization. However, this is also influenced by the keylength as the improvement depends on the attack complexity. For keylengths of 512 and 1024 bits, we can reduce the median of step 2b, see Table 4.21. However, with the initial bounds, we can only identify the first hole and hence the improvement is not significant compared to the total complexity, see Table 4.19. This is because the number of oracle calls required for step 2a is too large compared to the number of values we can skip.

In Figures 4.7, 4.8 and 4.9, we can see that the hole for step 2a is closed, compared to Figures 3.4, 3.5 and 3.6. Additionally, the distribution of oracle calls required for step 2b now looks similar to the distribution of oracle calls for step 2a.

keylength	median total	mean total	original median total	original mean total
512	338 383	416 462	359 198	434 687
1024	134 492	185 089	153 859	200 970
2048	42 177	94 632	50 191	109 099

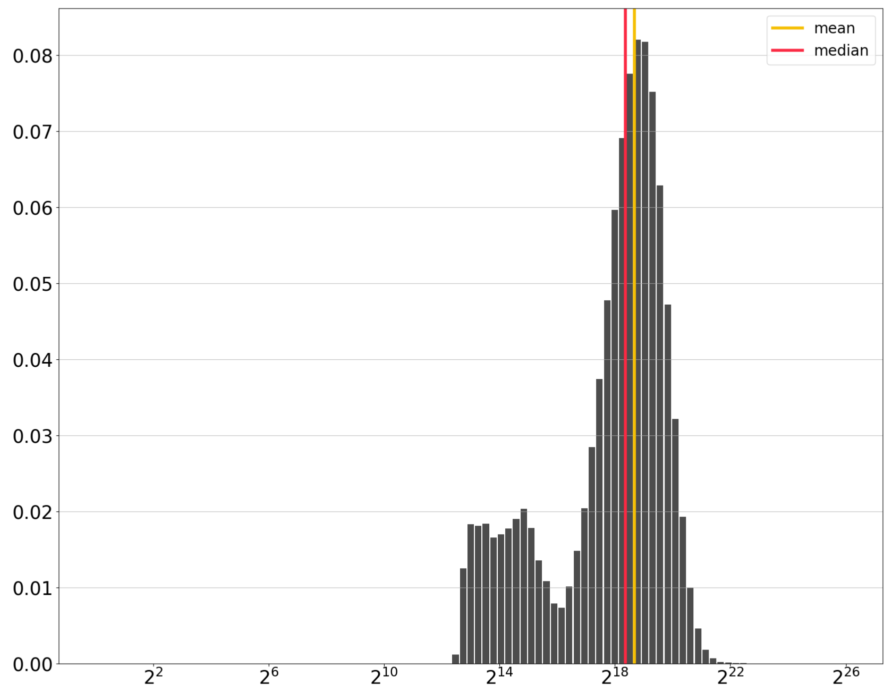
Table 4.19: Bleichenbacher's Attack with Skipping Holes – FFT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	137 592	208 176	144 030	213 858
1024	61 751	97 733	68 647	103 614
2048	30 014	51 431	37 019	56 882

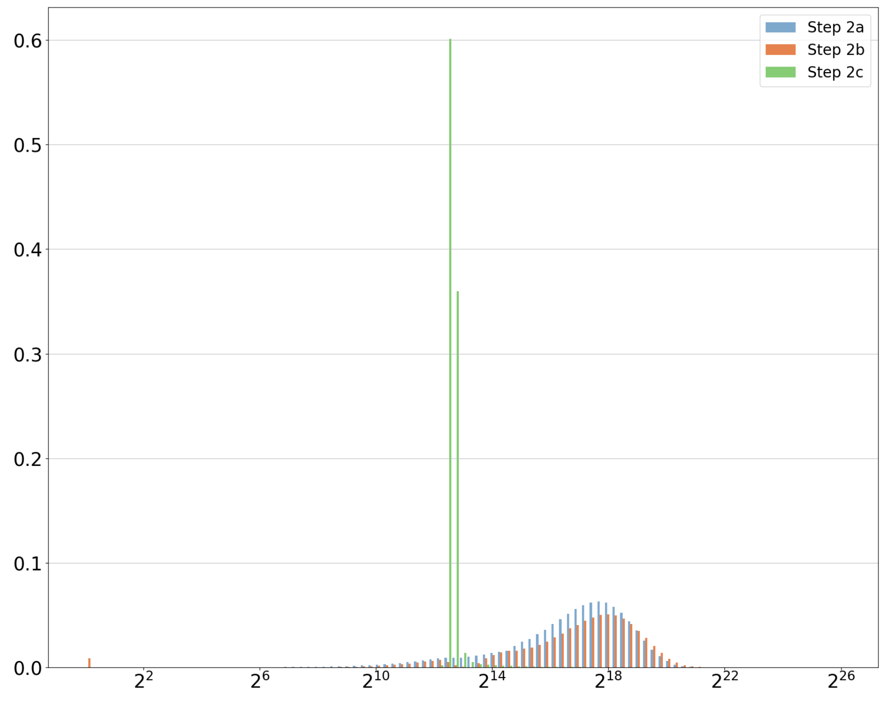
Table 4.20: Bleichenbacher's Attack with Skipping Holes – FFT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	107 318	200 315	121 714	212 733
1024	19 590	75 148	32 483	84 762
2048	0	27 117	0	33 371

Table 4.21: Bleichenbacher's Attack with Skipping Holes – FFT Oracle – 10^6 Simulations – Step 2b

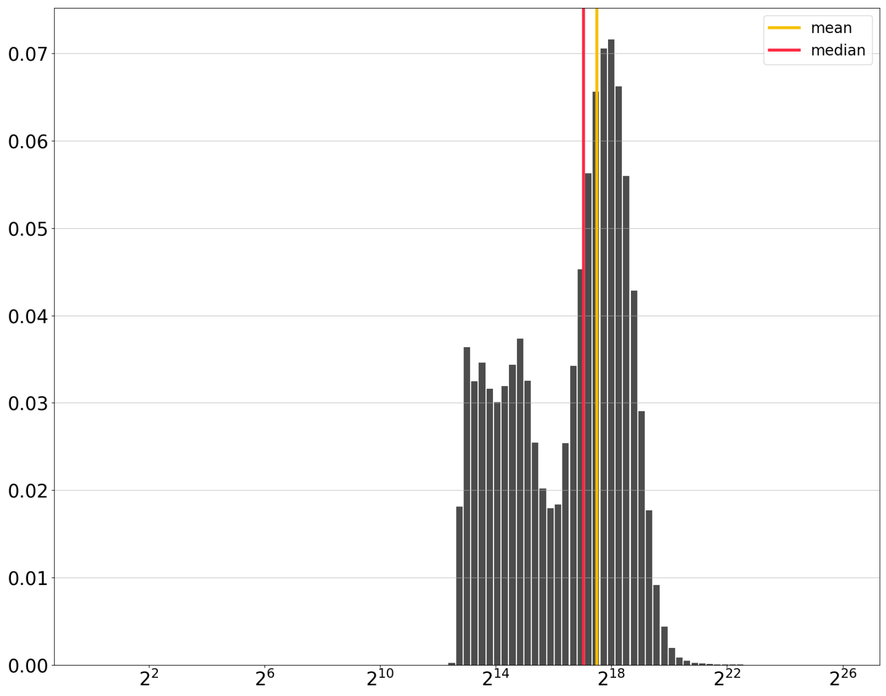


(a) Total Distribution

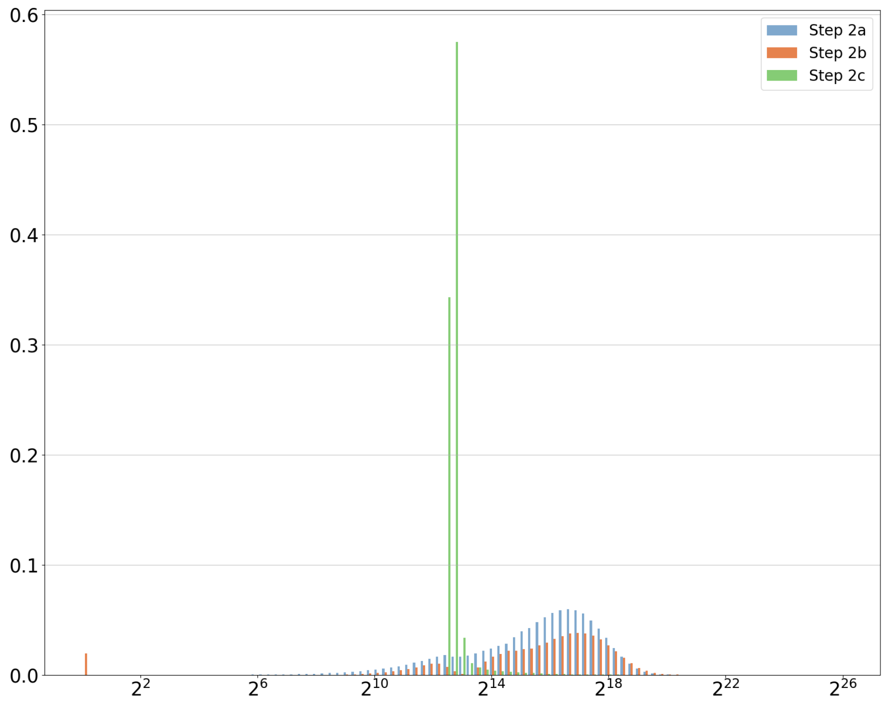


(b) Distribution of Individual Steps

Figure 4.7: Bleichenbacher's Attack with Skipping Holes – FFT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 5 from Total, 3 from Step 2b, 2 from Step 2c

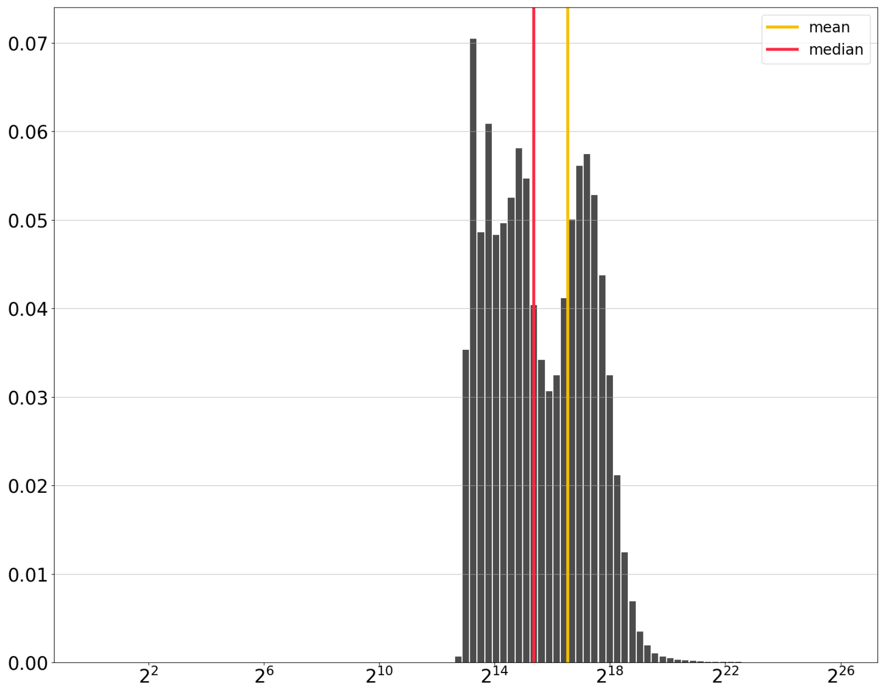


(a) Total Distribution

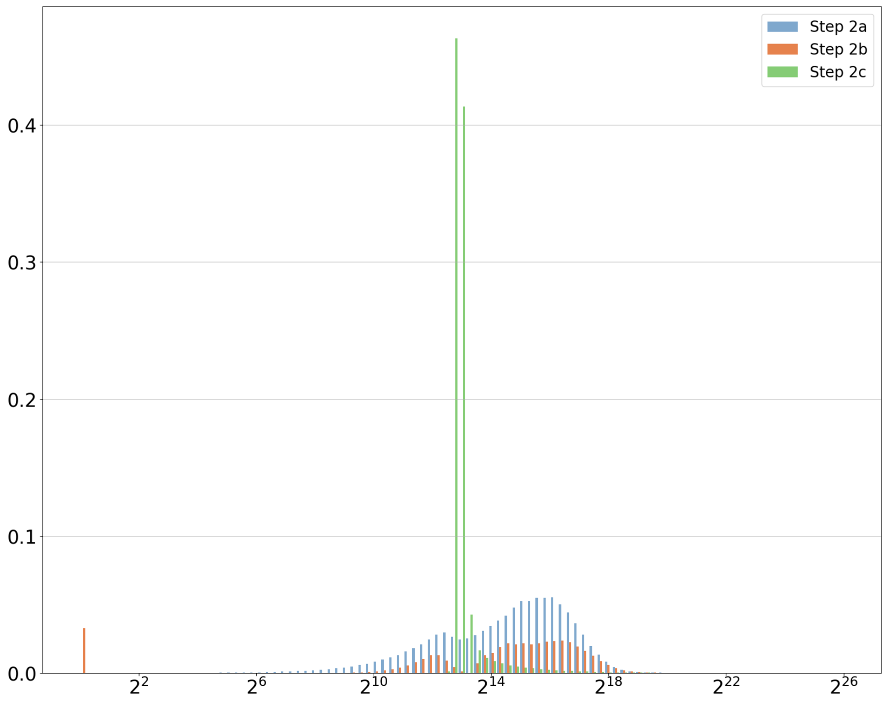


(b) Distribution of Individual Steps

Figure 4.8: Bleichenbacher's Attack with Skipping Holes – FFT Oracle – 10^6 Simulations – keylength 1024 – outliers removed: 11 from Total, 1 from Step 2b, 10 from Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.9: Bleichenbacher's Attack with Skipping Holes – FFT Oracle – 10^6 Simulations – keylength 2048 – outliers removed: 15 from Total, 15 from Step 2c

Bad Version Oracle

The experimental results for the Skipping Holes method are omitted as the technique does not provide a significant improvement for the BVO and the results are highly influenced by the randomness of the experiments. We note that as the improvement is for free, it should still be applied. However, the overall complexity is much larger than what can be gained by skipping the identifiable values. We will see that in combination with the Trimmers method presented in the following section, we can notably improve the attack complexity.

4.5 Trimmers

4.5.1 Description

The Trimmers improvement was introduced in 2013 alongside the Skipping Holes method [2]. This technique improves the complexity of step 2a by "trimming" the initial interval M_0 before starting Bleichenbacher's attack. By having tighter bounds on the desired message m_0, we can find the value s_1 more efficiently. This is achieved by combining the trimming of M_0 with the insights gathered in the previous section about skipping holes. The smaller the interval M_0 is, the more holes can be identified, and hence the more values can be skipped in step 2a. We start by introducing the theory behind this method, followed by a description of our implementation of the Trimmers method. Finally, we compare the experimental results obtained with this improvement in combination with the Skipping Holes and Parallel Threads methods to the original algorithm.

The goal of Bleichenbacher's attack is to find the decryption of some ciphertext c. To do this, we multiply the ciphertext by different values s^e (mod n), which is equal to multiplying the corresponding plaintext by s (mod n). With the help of the values s that succeed, we collect more and more information about the intervals that m_0 can lie in until we end up with only one possible value. The fundamental observation presented by Bardou et al. [2] is that instead of multiplying the message m by some integer s, we can also divide it by an integer t. This can be done by multiplying m_0 by t^{-1} (mod n), which corresponds to multiplying c by t^{-e} (mod n). We know that these inverses are unique because multiplication modulo n is a group operation on (Z_n)*. Furthermore, if the original message m_0 is divisible by t, the result $m_0 t^{-1}$ (mod n) is m_0/t. On the other hand, if m_0 is not divisible by t, this results in a random value that cannot be predicted.

Using this fact, Bardou et al. devised the following proposition:

Proposition 4.3 *Let u and t be two coprime positive integers such that $u < \frac{3}{2}t$ and $t < \frac{2n}{9B}$. If m_0 and $m_0 \cdot ut^{-1}$ (mod n) are PKCS #1 v1.5 conforming, then m_0*

is divisible by t.

Proof Because $u < \frac{3}{2}t$ we know that $m_0 u < m_0 \frac{3}{2}t < 3B\frac{3}{2}t < n$. It follows that $m_0 u \pmod n = m_0 u$. Let $x = m_0 \cdot ut^{-1} \pmod n$. Since x is PKCS #1 v1.5 conforming, we know that $x < 3B$ and hence $xt < 3Bt < n$. So, we can conclude $xt \pmod n = xt$. Combining these insights, we get $xt = xt \pmod n = m_0 u \pmod n = m_0 u$. Since u and t are coprime, this implies that t divides m_0. □

This proposition suggests that if one can find two coprime positive integers u and t with $u < \frac{3}{2}t$ and $t < \frac{2n}{9B}$ such that $m_0 \cdot ut^{-1} \pmod n$ is also PKCS #1 v1.5 conforming, we know that m_0 is divisible by t and hence $m_0 \cdot ut^{-1}$ (mod n) $= m_0 \cdot u/t$. It follows that $2B \leq m_0 \cdot u/t \leq 3B - 1$. Hence, we can now "trim" the initial interval M_0, which we know m_0 lies in. The new bounds are: $2B \cdot t/u \leq m_0 < 3B \cdot t/u$.

Next, Bardou et al. [2] describe how we find these values u and t. Because $2B \leq m_0 < 3B$ we only have to search for t and u such that $2/3 < u/t < 3/2$. For smaller or larger fractions u/t the message $m_0 \cdot u/t$ never lies in the interval $[2B, 3B - 1]$.

To analyze the probability that $m_0 \cdot u/t \in [2B, 3B - 1]$ given a fraction u/t in the specified range, assume that m_0 is uniformly distributed in $[2B, 3B - 1]$. The probability that t divides m_0 is $1/t$. For $2/3 < u/t < 1$ the probability that t divides m_0 and $m_0 \cdot u/t$ is in the interval is $1/t \cdot (3 - 2 \cdot t/u)$ and for $1 < u/t < 3/2$ it is $1/t \cdot (3 \cdot t/u - 2)$. We can directly see that smaller values t are more likely to succeed. This is why we will start with small values for t and then work our way up to trying larger denominators. However, larger values for t allow for a more efficient trimming as one is more flexible in choosing u values. Furthermore, for u close to t, the probability is higher that we succeed. This observation will also be used for finding the values for u and t efficiently.

To find the best u and t pairs, Bardou et al. construct a list of suitable fractions u/t called "trimmers". They used a few thousand trimmers in practice depending on which oracle is used and upper bound t by 2^{12}. For small t, they added all suitable fractions to the trimmers list. For t > 50, they only added $(t-1)/t$ and $(t+1)/t$. Each of these trimmers is then tested against the oracle to see whether it can be used to trim the bound of the interval $M_0 = [2B, 3B - 1]$. We have noted before that large denominators t allow for more efficient trimming. This is why in [2], they adapted the search for the best u and t pairs by computing the least common multiple t' of the denominators of all successful trimmers. The least common multiple t' also divides the message m_0. Using t', Bardou et al. then search for the lowest and highest numerators u_l and u_h, resulting in a PKCS conforming message. We search for two numerators because we want t'/u_l for trimming the lower bound to be as large as possible and t'/u_h for trimming the upper bound to be as small as possible. Using the least common multiple allows us to do a more efficient trimming as all valid u/t fractions observed during the generation of denominators still work in combination with t', but there are now new possible fractions u'/t'. Hence, we could be able to find better trimmers.

The search for these trimmers also requires oracle calls. By testing more trimmers, we might find better trimmers and thus be able to trim the interval M_0 more. The shorter the initial interval M_0, the faster we can potentially find s_1 and overall succeed with the attack. On the other hand, if we spend too many oracle calls on testing trimmers, the overall complexity could be worse than without the Trimmers method. Bardou et al. stated their experimentally found values for the best number of oracle calls allowed for testing trimmers. They also declared that the bound of 50 in the construction of the trimmers is an experimentally found bound and might not be optimal. In the following section, we present our implementation of the Trimmers method with the aim of finding good trimmers in the most efficient way to improve the performance of the attack maximally.

4.5.2 Implementation of Trimmers

When implementing the Trimmers method ourselves, we made a few observations and adapted the algorithm accordingly to do as few oracle calls as possible while finding good trimmers. If the algorithm is implemented as stated in [2], different trimmers with the same denominator are sent to the oracle when testing the generated trimmers. In the next step, one then computes the lowest common multiple t' of the denominators of all successful trimmers and discards the other information gathered during the first round of testing. So, one essentially only cares about finding successful trimmer denominators. This is why in our implementation, as soon as there was one successful trimmer for some denominator t, we skipped the remaining possible trimmers and directly tested the next denominator.

Additionally, we want to spend the oracle calls we allow for searching trimmers as carefully as possible. We want to find many different working denominators to obtain a large least common multiple, which will allow for a more effective trimming. To spend our oracle calls in the best way, we state the following:

Lemma 4.4 *Suppose we have an integer $t > 4$. Then, whenever t divides the message m_0, either $(t-1)/t$ or $(t+1)/t$ is a valid trimmer.*

Proof The message m_0 is either in the interval $L = [2B, 2.5B)$ or $R = [2.5B, 3B-1]$. If $m_0 \in L$ and $t > 4$ divides m_0, then $m_0 \cdot (t+1)/t \leq$

$m_0 \cdot 6/5 < 2.5B \cdot 6/5 = 3B$ and hence $m_0 \cdot (t+1)/t$ is also in the interval $[2B, 3B-1]$. On the other hand, if $m_0 \in R$ and $t > 4$ divides m_0, then $m_0 \cdot (t-1)/t \geq 2.5B \cdot 4/5 = 2B$. This shows that either $(t-1)/t$ or $(t+1)/t$ is a valid trimmer if $t > 4$ divides the message m_0. $\square$

Using this observation and that the only possible trimmers for $t \leq 4$ are $4/3, 3/4$ and $5/4$, we proceed in the following way: First, we test the trimmer $4/3$. Then, we test $3/4$ and only if $3/4$ does not work we test $5/4$. Otherwise, we know that $t = 4$ divides m_0 and is a valid denominator, so we do not need to check $5/4$. For $t > 4$, we first test $(t-1)/t$. Afterwards, if and only if that trimmer does not result in a conforming message, we test $(t+1)/t$. With this procedure, we should find more valid trimmer denominators than by checking all the possible u/t fractions and possibly testing one valid denominator t several times before finding the numerator u it works with. Doing this, we spend our oracle calls more carefully and hence should be able to conduct a more effective trimming afterwards.

However, there exists another problem. Considering the different oracle types, we realize that if we test a valid trimmer, i.e., $m_0 \cdot u/t \in [2B, 3B-1]$, the oracle might still reject the message if $m_0 \cdot u/t$ does not pass all its checks. For example, the message could be in the interval $[2B, 3B-1]$, but if it has a zero byte in the first eight bytes of padding, the TFT oracle rejects it. Hence, an oracle may return `False` for a working trimmer. We refer to this as a false negative result. Recall that we want to find as many working t denominators as possible such that the least common multiple t' is large. So, we do not want the probability of missing a valid denominator to be too large. This observation motivated a slight adjustment to the above procedure. For one denominator t, we do not only test the trimmers $(t-1)/t$ and $(t+1)/t$, but also $(t-2)/t$ and $(t+2)/t$, ... until $(t-s_t-1)/t$ and $(t+s_t+1)/t$ for some value s_t. As soon as one trimmer works, we stop checking and append t to the list of valid denominators. Of course, it always has to hold that $t - x \geq 2t/3$ or $t + x \leq 3t/2$ and $\gcd(t-x, t) = 1$ or $\gcd(t+x, t) = 1$, otherwise we do not test the corresponding trimmer $(t-x)/t$ or $(t+x)/t$. In this way, we reduce the probability of rejecting a valid trimmer denominator t. We aim to find a good trade-off between not rejecting too many valid denominators while also not wasting too many oracle calls on denominators that do not work. So, we have to define reasonable values for s_t for the different oracle types and keylengths, depending on how large the probability of receiving a false negative result is. To decide on this, we will analyze the different probabilities of receiving a false negative result in the next section. But first, let us present the rest of the Trimmers implementation.

In the first step, we have found a list of working denominators t and computed their least common multiple t'. From the analysis, we know that we want to find a trimmer $1 < t'/u_l < 3/2$ to trim the lower bound and another trimmer $2/3 < t'/u_h < 1$ to trim the upper bound. Bardou et al. [2] did not specify how they used this information exactly in their implementation of the Trimmers method. However, naively searching for u_l and u_h is inefficient, especially for large values t'. This is why we perform a binary search for u_l and u_h in the respective intervals. For u_l, we do a binary search in $[\lfloor 2t'/3 \rfloor + 1, t']$ and for u_h in $[t', \lceil 3t'/2 \rceil - 1]$. Let us define the median value of an interval for which we do a step of the binary search as u_m. Now we describe the binary search for u_l. We want to minimize u_l such that t'/u_l is as large as possible. Hence, if u_m is accepted by the oracle, we update the upper bound of the interval to u_m. Like this, we keep the guarantee that the upper bound is a valid trimmer nominator for t'. If u_m is rejected, we set the lower bound of the interval to $u_m + 1$. When searching for u_h, we do it the other way around because we want to maximize u_h. We update the lower bound to u_m whenever it is a valid nominator. Therefore, the lower bound always results in a working trimmer pair with t'. If u_m does not work, we update the upper bound to $u_m - 1$ and continue searching in this new interval.

For this procedure, we have the same problem with false negatives as before. Some false negatives could occur, resulting in cutting the interval incorrectly and thus losing a good trimmer. Hence, we want to reduce the probability of falsely discarding the interval in which the best trimmer lies. An important observation is that, even if our least common multiple t' is large, the binary search does not require many oracle calls. Thus, we can check more values around u_m at each step to reduce the probability of making a wrong decision. This is why we determine some value s_b for the different oracles to use in the binary search. We now describe how s_b is used to improve the binary search. When searching for u_l, we do not want to throw away the whole lower interval because u_m resulted in a false negative result, but some values below u_m would have worked as a trimmer numerator. To make the probability of this event occurring smaller, we first check u_m, then $u_m - 1$ until $u_m - s_b$. If any of these numerators work as a trimmer in combination with t', we can adapt the upper bound to this numerator and continue searching in the new interval. On the other hand, if all of the values do not work, we continue checking in the upper interval, i.e., we update the lower bound to $u_m + 1$. Different from testing for valid trimmers, we do not check around u_m in both directions. We only care about making the probability of falsely rejecting all numerators $u' \leq u_m$ low because that would hurt the performance of the trimmers. For finding the best value for u_h, we proceed in an analogous way. First, we check u_m, then $u_m + 1$ until $u_m + s_b$. As soon as we find one valid trimmer numerator, we adjust the lower bound accordingly. If none of the tested trimmers is accepted by the oracle, we continue in the interval with a new upper bound of $u_m - 1$.

We note that our implementation of the Trimmers method has the aim of finding the best trimmer pairs. However, it is not guaranteed to be optimal. One possible enhancement, for example, could be to allow s_t to shrink with growing t as it is less likely that a larger t divides the message. Furthermore, in the current implementation, we gradually increase t and test each possible denominator without using the already collected knowledge about valid trimmer denominators. A further improvement would be to only allow testing a denominator t which does not divide the least common multiple of the list of trimmer denominators collected so far. This follows because if t divides their least common multiple, we already know that t is also a valid trimmer and hence do not have to check it.

4.5.3 Probability of False Negative Result

To complete the description of our implementation of the Trimmers method, we still need to provide reasonable values for s_t and s_b. In order to do this, we analyze the false negative probability for the different oracle types and keylengths. For this, assume that some t divides m_0 and u/t is a valid trimmer, i.e., $m_0 \cdot u/t \in [2B, 3B-1]$. According to our theoretical analysis in Section 3.3, this means that for $m_0 \cdot u/t$ event A holds. The probability of a false negative result (FN) is now precisely $\Pr(FN) = 1 - \Pr(P \mid A)$, which means that although the message is in the interval $[2B, 3B-1]$, it does not pass all checks of the oracle. Hence, the oracle returns `False`. This is interpreted as u/t being an invalid trimmer. Therefore, we can reuse the computed probabilities from our theoretical analysis for approximating the likelihood of receiving a false negative result.

Based on this probability, we want to define the values s_t and s_b to bring the probability of making a wrong decision (WD) down to a reasonable value. We want a good trade-off between a low $\Pr(WD)$ and not doing too many unnecessary oracle calls. To approximate $\Pr(WD)$ by using $\Pr(FN)$, we make some simplifications.

First, assume that $m_0 \cdot u'/t \in [2B, 3B-1]$ for all the trimmers u'/t we test. This assumption is reasonable because, for large values of t, we try different trimming fractions that are close to each other. Hence, we can assume that if u/t is a valid trimmer, then all other fractions u'/t we test also keep the message in the interval $[2B, 3B-1]$. Additionally, we assume that the tested trimmers are independent of each other. With that, the probability of receiving a false negative result is independent for the different tested trimmers. This assumption is made for simplicity, but it is important to note that it does not hold. For example, a false negative result for a TFT oracle means that there is a `0x00` byte in the bytes 3 to 10. Therefore, by trying a trimmer with a fraction close to u/t, we will likely still have a zero byte in the bytes 3 to 10 and again receive a false negative result. Nevertheless, we assume this to analyze the probability of making a wrong decision.

Using these assumptions, the probability of making a wrong decision when generating and testing valid trimmer denominators is $\Pr(WD_t) = \Pr(FN)^{(2s_t+2)}$. Furthermore, when using s_b for the binary search, the probability of making a wrong decision is $\Pr(WD_b) = \Pr(FN)^{(s_b+1)}$. We now compute the respective probabilities for the different oracle types and provide precise values for s_t and s_b.

TTT Oracle

We know that $\Pr(P \mid A) = 1$ and hence $\Pr(FN) = 0$. Therefore, we set $s_t = s_b = 0$ because we cannot have any false negative results. This means that when searching for trimmer denominators for $t > 4$, we only try $(t-1)/t$ and $(t+1)/t$ as we know that if t divides the message, one of these two trimmers has to work. For $t \leq 4$, we potentially test all three pairs. When performing the binary searches in the intervals, we only test u_m.

TFT Oracle

We know that $\Pr(P \mid A) = \frac{255}{256}^8$ and hence $\Pr(FN) = 1 - \Pr(P \mid A) = 1 - \frac{255}{256}^8 < 0.031$. Therefore, the probability of receiving a false negative result is roughly 3%. For this oracle, we also set $s_t = s_b = 0$ as this is a low false negative probability. Larger values for s_t and s_b were also tested but resulted in a higher attack complexity. So, we reason that a false negative probability of 3% is low enough to provide good performance.

FFT Oracle

We know that $\Pr(P \mid A) = \frac{255}{256}^8 \cdot (1 - (\frac{255}{256})^{k-10})$ and hence $\Pr(FN) = 1 - \frac{255}{256}^8 \cdot (1 - (\frac{255}{256})^{k-10})$. This probability decreases with growing keylength. Hence, we define different values for s_t and s_b depending on the keylength. We experimentally saw that bringing $\Pr(WD)$ to approximately 3% provides the best improvement to the attack. So, we set s_t and s_b to achieve this.

For a keylength of 512 bits, we have $\Pr(FN) \approx 0.815$. It follows that $s_t = 8$ and $s_b = 17$.

For a keylength of 1024 bits, we have $\Pr(FN) \approx 0.641$. It follows that $s_t = 3$ and $s_b = 7$.

For a keylength of 1024 bits, we have $\Pr(FN) \approx 0.4$. It follows that $s_t = 1$ and $s_b = 3$.

Bad Version Oracle

We know that $\Pr(P \mid A) = (1 - 2^{-16})(\frac{255}{256}^{k-51} \cdot 2^{-8})$ and hence $\Pr(FN) = 1 - (1 - 2^{-16})(\frac{255}{256}^{k-51} \cdot 2^{-8})$. This probability grows with growing keylength.

Overall, the probability of receiving a false negative result is very large.

For a keylength of 512 bits, we have $\Pr(FN) \approx 0.996$. It follows that $s_t = 450$ and $s_b = 900$.

For a keylength of 1024 bits, we have $\Pr(FN) \approx 0.997$. It follows that $s_t = 600$ and $s_b = 1\,200$.

For a keylength of 2048 bits, we have $\Pr(FN) \approx 0.998$. It follows that $s_t = 900$ and $s_b = 1\,800$.

For the BVO, we observe that this theoretical derivation results in very large values for s_t and s_b. Of course, in the actual code, we usually cannot use such large values of s_t and s_b because of the constraints that $2t/3 \leq u \leq 3t/2$.

We note that we started off using these values for s_t and s_b in our experiments and experimentally refined them later.

The following section shows the experimental results for our implementation of the Trimmers method combined with the Skipping Holes and Parallel Threads improvements. We also present the best parameters for the Trimmers technique found in practice, depending on the oracle type and keylength.

4.5.4 Experimental Results

For all experiments, we used the Trimmers technique in combination with the Parallel Threads and Skipping Holes methods. This was done as they provide a significant improvement, especially in combination with Trimmers. Since we want to find the best parameters for improving the attack while using all possible optimizations, it is reasonable to search for the trimmer parameters with the other optimizations.

As a first step, we tested different numbers of allowed oracle calls to generate and test trimmer denominators. The aim of this was to figure out the best value for each oracle and keylength combination. For this, we started by using the number of oracle calls presented by Bardou et al. [2] for testing trimmers as well as one smaller and one larger value for each oracle type and keylength. We performed 100 000 simulations for the TTT, TFT and FFT oracles and 1 000 for the BVO to determine which of the three number of allowed oracle calls works best with our implementation of the Trimmers method. Hereby, we kept s_t and s_b fixed on the values derived from the theoretical analysis. Then, we continued searching until we found a number of allowed oracle calls such that when allowing more or fewer calls, the attack complexity is larger. After having found these values, we fixed them and refined s_t and s_b in the same way, starting with our theoretically derived values.

The experimental results show that the Trimmers method in combination with Skipping Holes significantly lowers the complexity of step 2a. Furthermore, trimming M_0 also affects the performance of step 2b. Because the initial interval $M_0 = [a, b]$ is smaller if we find a trimmer, there are fewer possible values for r in step 3 of the attack. This means that it is less likely that one has to perform step 2b and even if we have to, we expect M_1 to contain fewer intervals. The influence of this can be seen in the mean and median values for step 2b and the mean number of rounds required of step 2b. However, this improvement is not significant compared to applying the Parallel Threads Method on its own, as the Parallel Threads Method already makes step 2b highly efficient. Lastly, the Trimmers technique does not further improve step 2c compared to the Parallel Threads Method.

Now, we present the experimentally best parameters found for the different oracles and keylengths and compare the results obtained with these parameters to the original Bleichenbacher algorithm. We note that step 2a is defined as the number of oracle calls required for finding s_1 after trimming the interval M_0. In total, to obtain the value s_1, we performed some number of oracle calls for the trimmers and step 2a. In the provided plots, we show the number of oracle calls allowed for generating and testing trimmer denominators. After finding the valid denominator and computing their least common multiple, we also need a certain number of oracle calls for the binary search, i.e., for finding u_l and u_h. This value, however, is not significant as the binary search is very efficient, even for a large least common multiple t'. Hence, we do not consider it in the figures.

TTT Oracle

For the TTT oracle, we observed that allowing 400 oracle calls for generating and testing trimmers gives the best attack complexity. As analyzed in the previous section, for this oracle, we set $s_t = s_b = 0$ because we cannot have any false negative results. Both the median and mean values observed for the total number of oracle calls required for the Trimmers technique were roughly 410.

From Table 4.23, we can see that step 2a is improved a lot by applying the Trimmers method. Figure 4.10 visualizes this. Step 2a is now often very cheap after spending approximately 400 oracle calls to find trimmers. Additionally, in the experimental results, we could see that step 2a usually is more efficient the more we were able to trim the interval M_0. The cases where step 2a still requires many oracle calls arise when we cannot find a good trimmer for the desired message m_0. From Table 4.24, it is visible that the complexity of step 2b is also reduced. As we can see when comparing these values to Table 4.2, step 2b is more efficient when combining the Parallel Threads Method with the Trimmers technique. However, this improvement is not significant as the Parallel Threads Method makes step 2b already essentially free. Finally, we could further reduce the mean number of rounds required for step 2b from 0.18 to 0.03.

As there are no false negative results for this oracle, we can often find good trimmers. This is visible in Table 4.25. In the median, we expect to be able to shrink M_0 by a factor of 30. However, the mean value is a lot larger because there exist cases where one can find highly efficient trimmers.

We only provide the plot visualizing the distribution of the required oracle calls for a keylength of 512 bits, see Figure 4.10, as the improvement of the Trimmers method looks similar for the longer keylengths.

keylength	median total	mean total	original median total	original mean total
512	2 114	10 007	21 114	36 272
1024	3 384	12 641	23 000	40 815
2048	5 860	19 012	26 439	49 767

Table 4.22: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TTT Oracle – 10^6 Simulations –Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	316	6 474	19 178	27 180
1024	329	6 518	19 318	27 361
2048	338	6 651	19 642	27 841

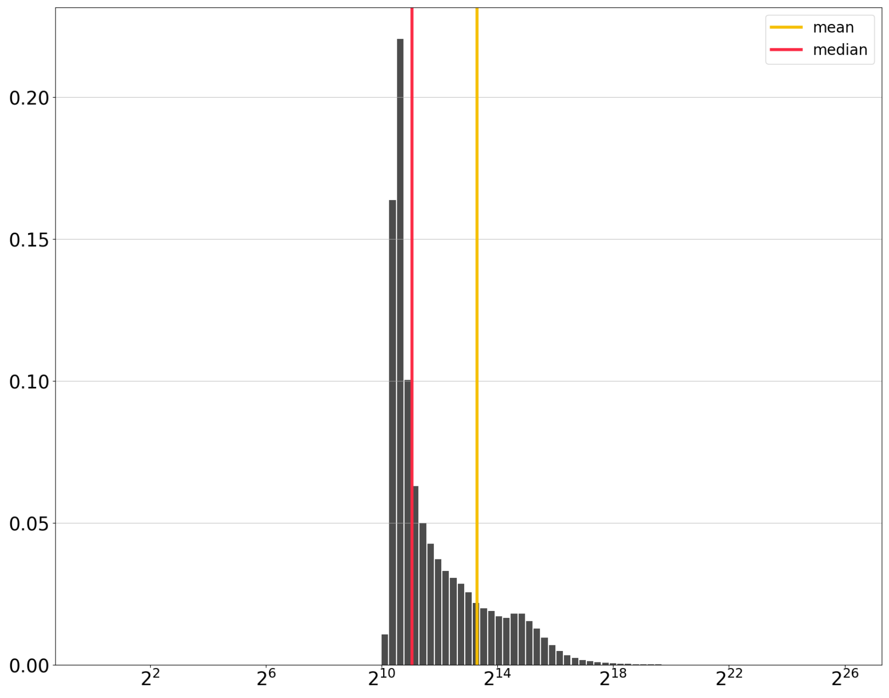
Table 4.23: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TTT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	0.82	0	5 634
1024	0	0.67	0	5 675
2048	0	0.61	0	5 829

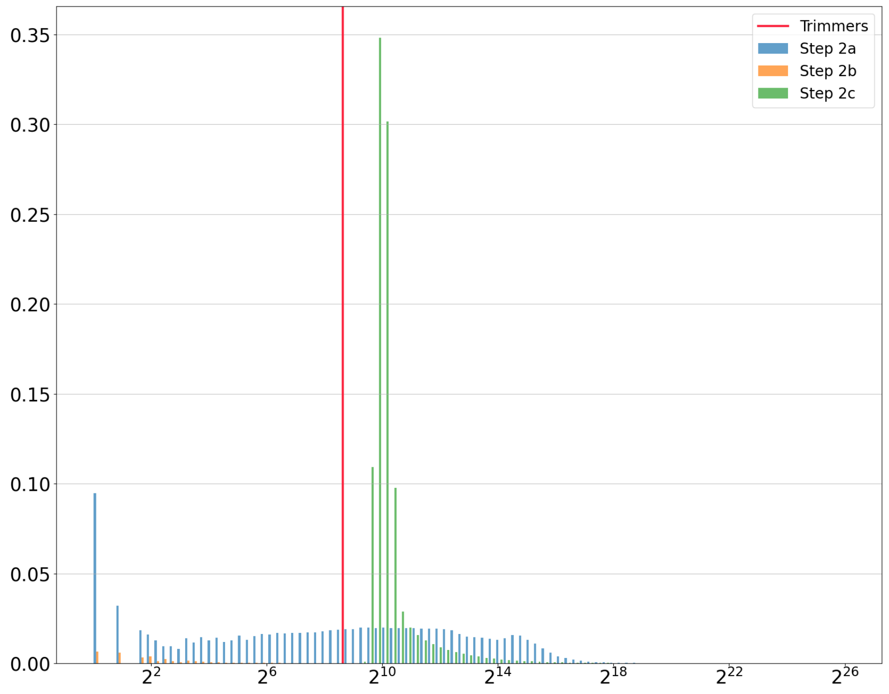
Table 4.24: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TTT Oracle – 10^6 Simulations – Step 2b

keylength	median trimmer improvement	mean trimmer improvement
512	30	1 248 449 116
1024	29	4 126 404 091
2048	29	412 411 811

Table 4.25: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TTT Oracle – 10^6 Simulations – Trimmer Improvement



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.10: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TTT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 4 from Total, 4 from Step 2c

TFT Oracle

For the TFT oracle, we experimentally found that allowing 500 oracle calls for generating and testing trimmers works the best. We again set $s_t = s_b = 0$ because the probability of receiving a false negative result is low. In practice, these parameters gave the best complexity improvement. Both the median and mean values observed for the total oracle calls required for the Trimming technique were roughly 510.

All observations are identical to the TTT oracle except for the trimmer improvement. We can observe in Table 4.29 that the median value has gone down compared to Table 4.25. This means that the Trimmers method is less effective for this oracle type because of the false negatives that can occur. What stands out more is that the mean value is drastically lower. This is unexpected as the probability of receiving a false negative result is very low for the TFT oracle. Hence, we expect the Trimmers method to achieve roughly the same improvement. Interestingly, during the experiments we observed that for the TFT oracle and also the more restrictive oracles, we could not provide an improvement of more than a factor of 2^8. On the other hand, for the TTT oracle, M_0 could be trimmed by up to a factor of 2^{50}.

We also note that for the TFT oracle, the mean number of rounds required of step 2b with all the applied improvements is 0.04 compared to 0.20 when only applying the Parallel Threads Method.

The distribution of the required oracle calls for the different steps after applying the Trimmers method with the described parameters looks identical as for the TTT oracle, see Figure 4.10.

keylength	median total	mean total	original median total	original mean total
512	2 479	10 980	22 077	39 530
1024	3 797	14 278	24 010	42 981
2048	6 355	19 605	27 486	51 362

Table 4.26: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TFT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	520	7 258	20 115	28 801
1024	540	7 416	20 259	29 017
2048	550	7 549	20 517	29 421

Table 4.27: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TFT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	0.76	0	6 969
1024	0	2.14	0	7 000
2048	0	0.80	0	6 911

Table 4.28: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TFT Oracle – 10^6 Simulations – Step 2b

keylength	median trimmer improvement	mean trimmer improvement
512	21	64
1024	20	63
2048	20	63

Table 4.29: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – TFT Oracle – 10^6 Simulations – Trimmer Improvement

FFT Oracle

For the FFT oracle, the experimentally best parameters are the following:

For a keylength of 512 bits, we allow 10 000 oracle calls for generating and testing trimmers. We set $s_t = 10$ and $s_b = 21$. Both the median and mean values observed for the total oracle calls required for the Trimming technique were roughly 10 100.

For a keylength of 1024 bits, we allow 4 000 oracle calls for generating and testing trimmers. We set $s_t = 4$ and $s_b = 9$. Both the median and mean values observed for the total oracle calls required for the Trimming technique were roughly 4 050.

For a keylength of 2048 bits, we allow 1 000 oracle calls for generating and testing trimmers. We set $s_t = 1$ and $s_b = 3$. Both the median and mean values observed for the total oracle calls required for the Trimming technique were roughly 1 025.

The main observations are the same as for the more permissive oracles. For the FFT oracle, the mean number of rounds required for step 2b now have the following values: 0.34 for a keylength of 512 bits, 0.21 for a keylength of 1024 bits and 0.12 for a keylength of 2048 bits. Furthermore, we can observe that the median and mean of the possible trimmer improvement have gone down compared to the more permissive oracles as the probability of receiving a false negative is higher for this oracle. However, with growing keylength, it becomes again easier to find good trimmers as the false negative probability shrinks, see Table 4.33.

In Figures 4.11, 4.12 and 4.13, the significant improvement on step 2a is visible, as well as how the different keylengths influence the attack complexity.

keylength	median total	mean total	original median total	original mean total
512	38 464	123 821	359 198	434 687
1024	16 137	54 058	153 859	200 970
2048	11 393	34 118	50 191	109 099

Table 4.30: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	21 153	105 768	144 030	213 858
1024	4 677	40 203	68 647	103 614
2048	1 804	18 849	37 019	56 882

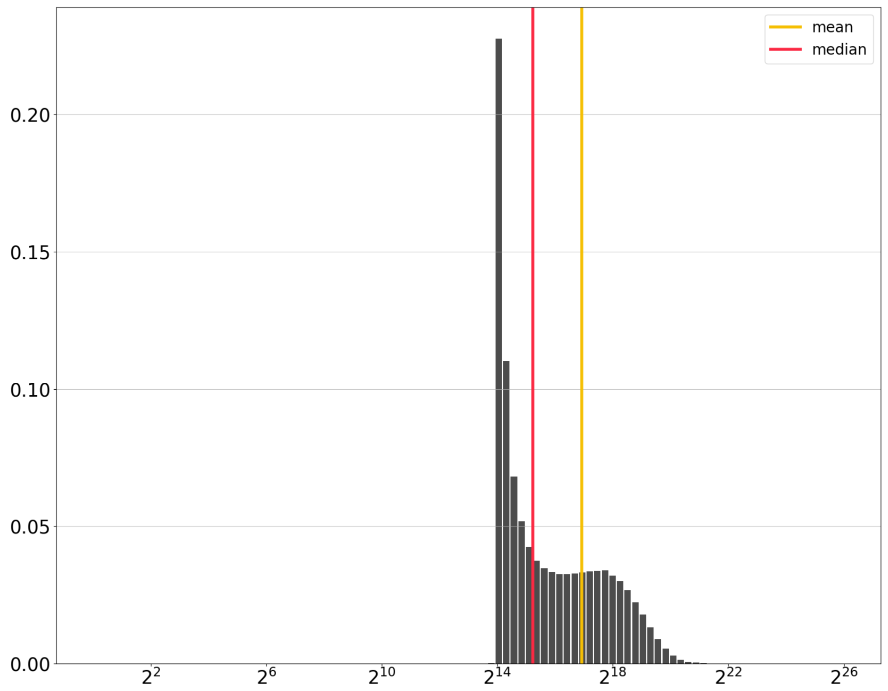
Table 4.31: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	0	43	121 714	212 733
1024	0	12	32 483	84 762
2048	0	9	0	33 371

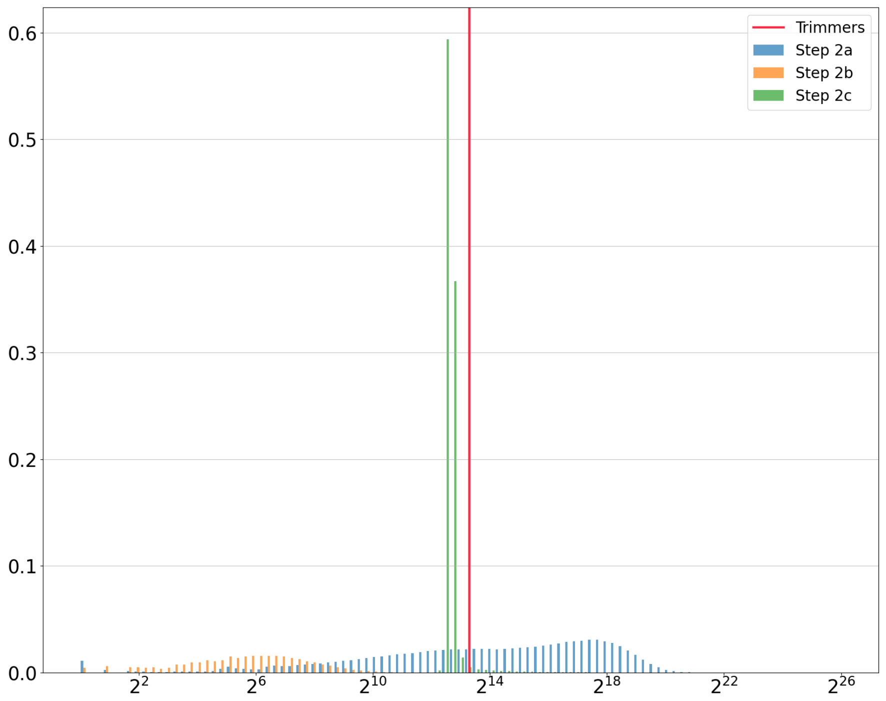
Table 4.32: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – Step 2b

keylength	median trimmer improvement	mean trimmer improvement
512	10	57
1024	14	61
2048	15	58

Table 4.33: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – Trimmer Improvement

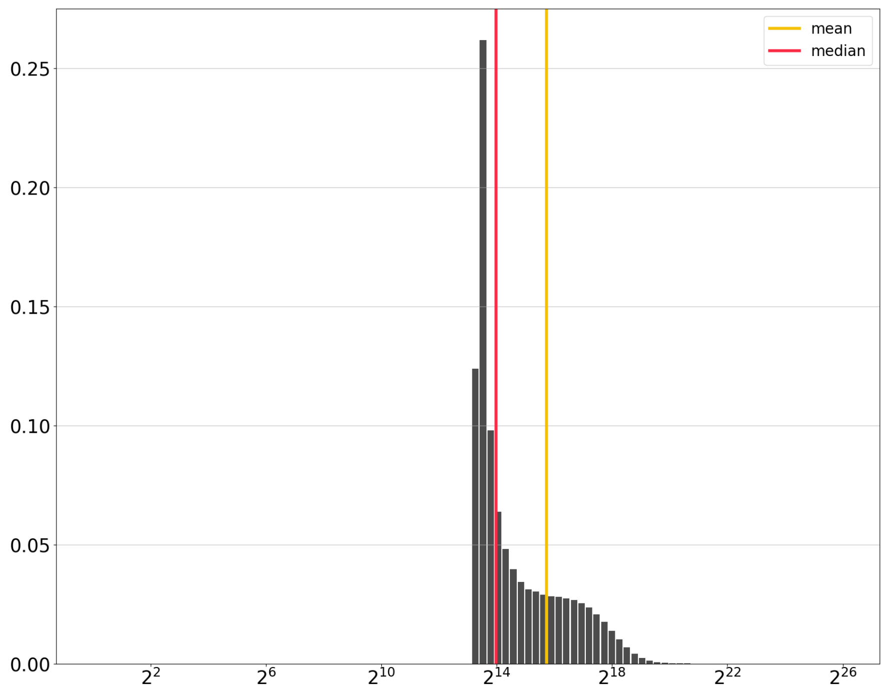


(a) Total Distribution

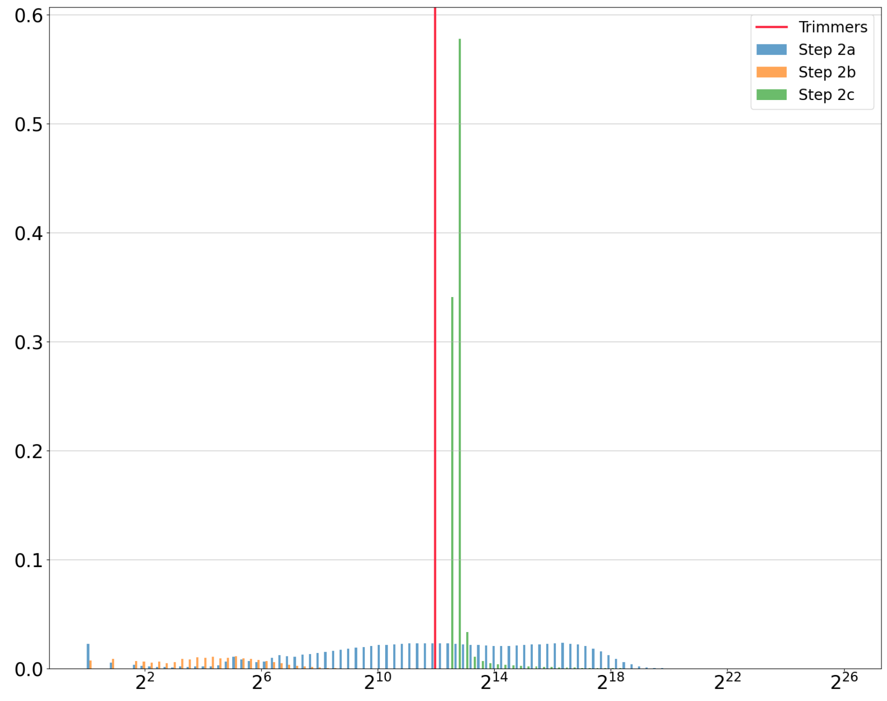


(b) Distribution of Individual Steps

Figure 4.11: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – keylength 512 – outliers removed: 6 from Total, 6 from Step 2c

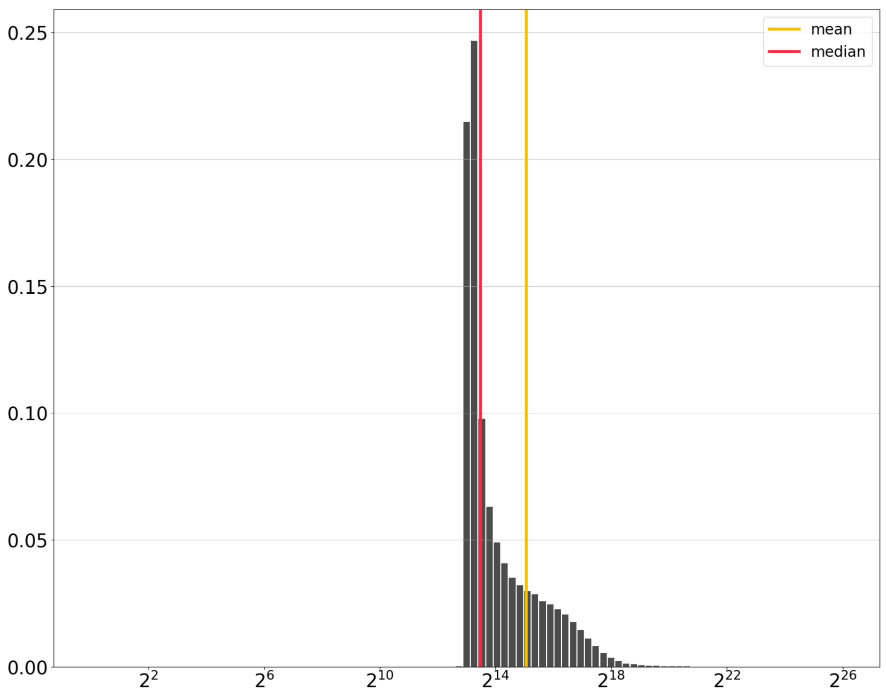


(a) Total Distribution

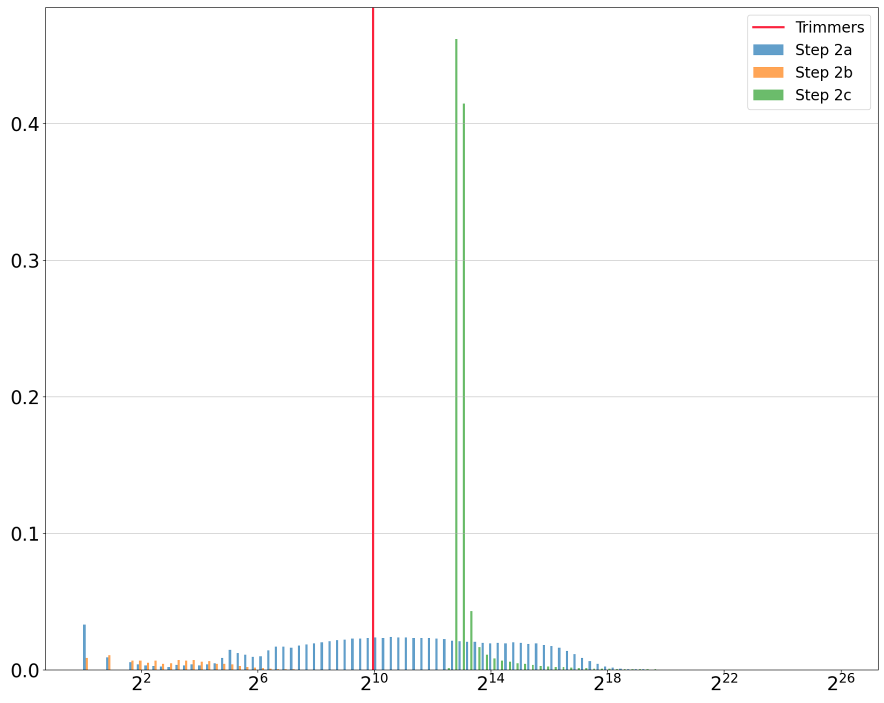


(b) Distribution of Individual Steps

Figure 4.12: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – keylength 1024 – outliers removed: 7 from Total, 7 from Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.13: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – FFT Oracle – 10^6 Simulations – keylength 2048 – outliers removed: 15 from Total, 2 from Step 2a, 13 from Step 2c

Bad Version Oracle

For the BVO, the experimentally best parameters are the following:

For a keylength of 512 bits, we allow 50 000 oracle calls for generating and testing trimmers. We set $s_t = 450$ and $s_b = 900$. The median values observed for the total number of oracle calls required for the Trimming technique was 50 000. Hence, usually we were not able to find a trimmer. The mean value was roughly 50 200.

For a keylength of 1024 bits, we allow 50 000 oracle calls for generating and testing trimmers. We set $s_t = 600$ and $s_b = 1\,200$. The median values observed for the total number of oracle calls required for the Trimming technique was 50 000. Hence, usually we were not able to find a trimmer. The mean value was roughly 50 200.

For a keylength of 2048 bits, we allow 80 000 oracle calls for generating and testing trimmers. We set $s_t = 900$ and $s_b = 1\,800$. The median values observed for the total number of oracle calls required for the Trimming technique was 80 000. Hence, usually we were not able to find a trimmer. The mean value was roughly 80 150.

We can see that for the BVO, it is challenging to find trimmers because of the high false negative rate. Hence, we cannot significantly improve step 2a, see Table 4.35. Because it becomes harder to find a trimmer with growing keylength, the mean trimmer improvement sinks with increasing modulus size, see Table 4.37. The main improvement on the total complexity still comes from the Parallel Threads Method for step 2b, see Table 4.36. However, in combination with the Trimmers technique, we can lower the complexity of step 2b notably compared to only applying the Parallel Threads Method, as visible when comparing the results to Table 4.11. Additionally, for the BVO, the mean number of rounds required for step 2b now have the following values: 1.23 for a key length of 512 bits, 1.25 for a keylength of 1024 bits and 1.23 for a keylength of 2048 bits. These are slightly lower compared to only applying the Parallel Threads Method, see Section 4.3.3.

We only visualize the improved distributions for a keylength of 512 bits, see Figure 4.14, as they look similar for the longer keylengths.

keylength	median total	mean total	original median total	original mean total
512	8 889 976	17 057 401	24 227 787	50 146 387
1024	12 033 457	21 122 774	31 076 365	62 343 618
2048	20 563 963	33 706 806	51 200 904	75 265 298

Table 4.34: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – BVO – 10^4 Simulations – Total

keylength	median 2a	mean 2a	original median 2a	original mean 2a
512	8 097 901	14 317 720	8 019 905	14 499 301
1024	10 511 671	17 590 261	10 548 557	18 231 944
2048	16 249 408	28 613 532	17 708 014	27 544 262

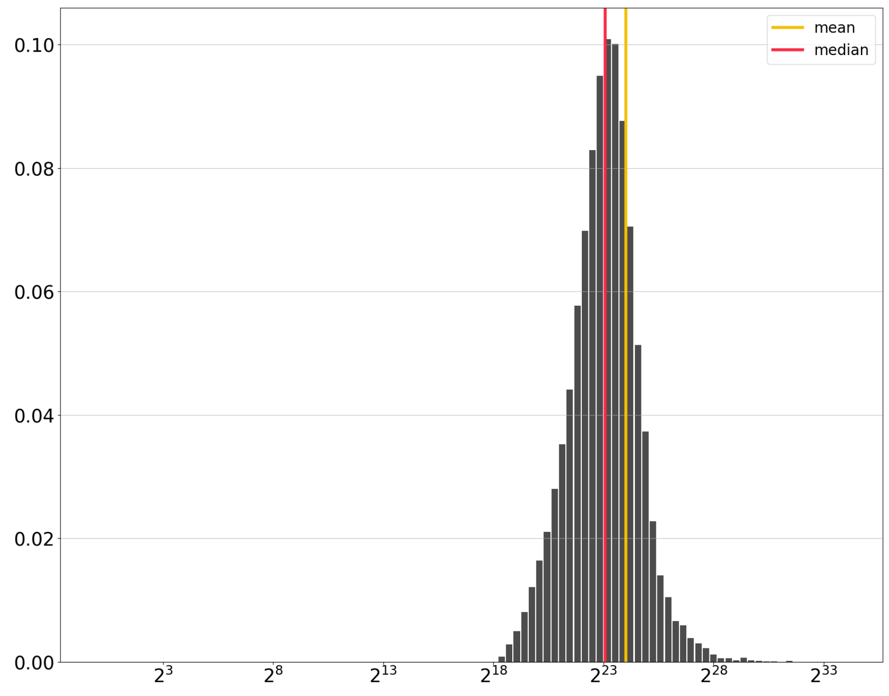
Table 4.35: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – BVO – 10^4 Simulations – Step 2a

keylength	median 2b	mean 2b	original median 2b	original mean 2b
512	112 155	1 661 303	11 113 772	34 805 695
1024	190 402	1 856 419	13 778 521	42 711 607
2048	492 769	1 721 121	22 443 601	44 014 214

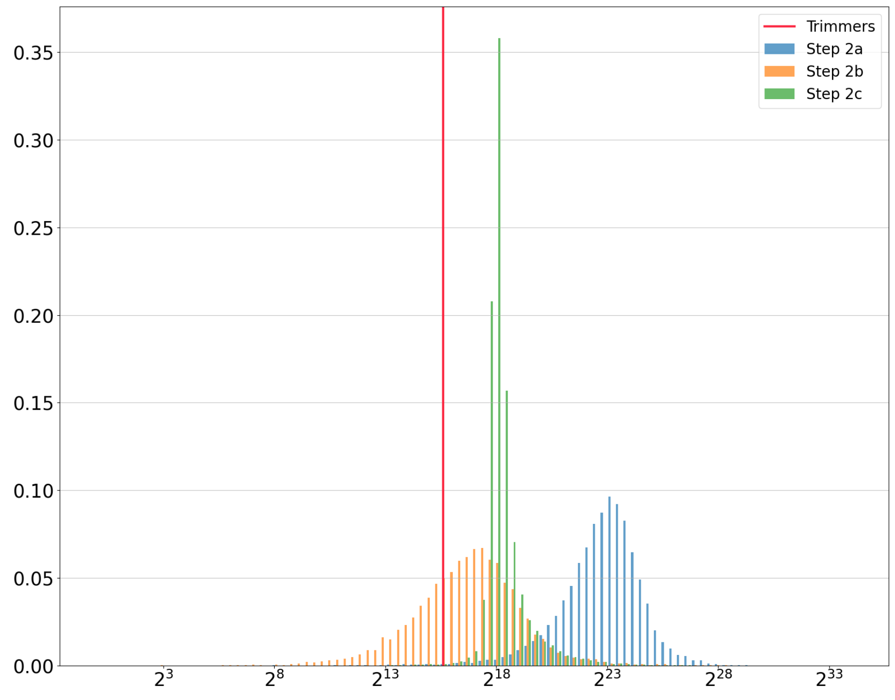
Table 4.36: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – BVO – 10^4 Simulations – Step 2b

keylength	median trimmer improvement	mean trimmer improvement
512	1	3.19
1024	1	2.52
2048	1	1.71

Table 4.37: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – BVO – 10^4 Simulations – Trimmer Improvement



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 4.14: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads – BVO – 10^4 Simulations – keylength 512

Chapter 5

Heuristic for Step 2c

In the previous chapter, we have presented different methods for improving the complexity of Bleichenbacher's attack. These techniques focus on making steps 2a or 2b less expensive as the number of oracle calls required for decrypting a message seems to rely on their performance. On the other hand, it is assumed that step 2c uses an efficient method for finding valid ciphertexts. Hence, this step is considered optimal and the improvements do not aim at improving its performance. However, this is not always the case. In the experimental results of the unmodified Bleichenbacher attack in Section 3.4, we could see that the distribution of oracle calls required for step 2c has a long tail. So, there exist some cases where Bleichenbacher's heuristic analysis for step 2c does not hold, see for example Figure 3.1. This is the case for all oracle types.

Following the experiments for the different improvements, we know that combining the optimizations can significantly decrease the attack complexity. The Trimmers and Parallel Threads Method successfully eliminate the immense complexity of step 2b. Additionally, the Trimmers and Skipping Holes techniques help in making step 2a a lot more efficient in most cases. After applying all improvements, it still seems that step 2a determines the complexity of Bleichenbacher's attack while the performance of step 2c is roughly fixed. However, the distribution of the number of oracle calls for step 2c still has a long tail, see for example Figure 4.10. Additionally, we can see that the outliers we removed for the different figures mostly came from step 2c. This indicates that the worst cases for Bleichenbacher's attack arise when step 2c requires an extremely large number of queries. In these cases, we cannot find the next s_i value quickly using the method of step 2c. These extreme cases also influence the mean value of step 2c significantly, which is visible in the different tables, for example Table 4.3. The mean number of oracle calls required for step 2c is always considerably larger than its median.

We conclude that the overall worst cases appear when the "efficient" method of finding the next s_i value in step 2c turns out not to be efficient at all. This motivated us to create a heuristic for improving the "bad" cases and aborting for "very bad" cases that cannot be saved. Before presenting the heuristic, we describe the observations made for the messages which require notably more oracle calls in step 2c than expected. These insights were then used to derive the heuristic.

5.1 Observation

First, we start by explaining the high-level pattern observed and then show a particular example of a case where the number of calls for step 2c was extremely large. In the following two sections, we assume that we have a TTT oracle with a keylength of 1024 bits. The examples we provide were found in this setting. For this oracle and keylength combination, we expect step 2c to require roughly 2 000 oracle calls with all applied improvements. However, the mean value is approximately 6 000, see Table 4.3.

In practice, the experiments which take unexpectedly long for step 2c start this step as expected. This means when starting step 2c, they only need one or two oracle calls to find the next s_i value, considering the TTT oracle. Then, after some number of rounds, they suddenly cannot find the next s_i value for a long time. After having finally found the next conforming multiplier s_i, they continue to follow a pattern. From there on, the experiments always require roughly the same number of oracle calls. But, this number is larger than the expected number of calls for one step of 2c. Essentially, after the period where they cannot find a value s_i for a long time, Bleichenbacher's method works again, just with some fixed offset.

Interestingly, this behavior happens at different scales. To demonstrate this, we describe three different examples:

- For a message m_1 and modulus n_1, we start by having to do two oracle calls to find s_2, followed by one oracle call to find s_3. Then, we require 129 oracle calls and after that always between 31 and 33. This results in a complexity of 31 811 for step 2c.
- For a message m_2 and modulus n_2, we start by having to do one oracle call to find the next s_i for seven rounds of step 2c. Then, it takes 11 379 oracle calls and after that always between 1 400 and 1 402. This results in a complexity of 1 387 189 for step 2c.
- For a message m_3 and modulus n_3, we start by having to do one or two oracle calls to find the next s_i. Suddenly, it takes 28 705 563 calls and after that once 3 633 812, once 5 776 136, once 5 897 519 and then

always between 5 776 136 and 5 776 138. This results in a complexity of 5 629 537 199 for step 2c.

We note that these experiments arose in experimental runs where all improvements were applied. However, the same behavior was observed for these experiments without any optimizations. Additionally, we expect the bad cases to arise from an unfortunate modulus and message combination. There seems to be a large region of r_i, s_i values that do not work and where Bleichenbacher's heuristic analysis for step 2c is wrong. One could possibly identify which ranges can be skipped, similarly to the Hole Skipping technique. However, we note that this is an open problem.

Our experimental observations suggest that step 2c can be improved for some of the bad cases by exploiting the structure of required oracle calls. In the following sections, we present a heuristic devised to do precisely that and analyze its performance in practice. But, we first want to look at the last example in more detail to provide the values for a case where step 2c is extremely unfortunate.

5.2 Example of an Extreme Case

Recall that we consider the TTT oracle and a keylength of 1024 bits. Additionally, all improvements are applied. Hence, we allow 400 oracle calls to find trimmer denominators and set $s_t = s_b = 0$. Furthermore, the Parallel Threads Method is performed with a bound of 16 000 intervals on M_1.

We first display the relevant values of the RSA keys, i.e., (n, e, d), and the message m. Then, we explain the run of the algorithm on this message and modulus combination in detail.

n = 1199725689746578919119599477279587385317778413719551540111539691

8951503336647323055184106739216501477266202281805159423463409824369

1718033865314593335094058708319328748614037708004150691268285952132

6622601051124580525403867641039868107446834416174599001233786662233

15861155094559095945256048112008434445593989

e = 65537

d = 5223640471324386141733337059092272462890093692950150784607593437

0245990465116096766662875292360478760668184706549006538066801858244

0937805567351378407390697270101749981791166785469464796957634582809

4508545746943486365357619447367695509074926685022639415954451485663

98115427892252756517640086051847309009817 93

m = 5757117374837080106362895065131625459948524262525083733552337861

3219825205534219401386354182928560932075070804445668252304823541003

3923054014812850146599607948717587940118591706053064762058995408131

1423791339709802121481039191106630412177084083622299772401409304241

930817987986350425955372160585228338432

We start by generating and testing different trimmer denominators for 400 oracle calls and end up with the following list of working denominators t_d:

t_d = [4, 7, 8, 13, 14, 16, 26, 28, 32, 52, 56, 64, 91, 104, 112, 128, 182, 208]

The least common multiple t' of this list is 11 648.

Then, we search for the best trimmers t'/u_l and t'/u_h and obtain: $u_l = 11\,100$ and $u_h = 16\,649$.

This allows us to shrink the initial interval:

M_0 = [54861240687936886832559362511872092700743926359323320701120019

8845619738175967294716517569953636279361328472533787211174495818386

2744647903224103718245670299614498700710006264535590197791934024641

5125412623597951915939539289081689902927585003914562122604525965755

09589842140073806143686060649302051520512,

8229186103190533024883904376780813905111588953898498105168002982684

2960726395094207477635493045441904199270880068081676174372757941169

7185483615557736850544942174805106500939680338529668790103696226881

1893539692787390930893362253485439137750587184318390678894863264384

763210110709215529090973953077280767]

by a factor of 7 933 to:

M_0 = [5756970554352151872303166257101676898903290578679261617357171

0956340348741204207647369339232612210648655441874536517439284029662

7326573502493282523514644753602789917710541918642437429043967995666

8859286542699720215467241023776788504408112703834205582018896562604

642856079391673846302851822922799125775758,

57573163391172640202923730062311802730938667868946907279114000085474

61148063247386201570654231445175686906764436500783014469901402479097

13286873326717450606492068720636706285386304474357743235500849943665

90155770068790655922808293950624708101891718986494460788856570096535

45039914766201144312847945472183]

We proceed by searching for the first conforming multiplier s_1 and can find $s_1 = 20\,840$ with a single oracle call because of the Skipping Holes method combined with the successful trimming of M_0. This shows that even when finding good trimmers, a very bad case for step 2c can arise. Without the improvements, step 2a would have taken 6 262 oracle calls. We note that we find the same value for s_1 with or without the optimizations.

Because $|M_1| = 1$, we skip step 2b and continue directly with step 2c. Until now, the attack has been highly efficient as we have only needed 426 oracle calls, whereof 25 were required for the binary search of the Trimmers improvement.

For step 2c, the following pattern of oracle calls required to find the next s_i occurs:

- We need to perform two oracle calls to find s_2 and another two oracle calls for s_3.
- Then, for 16 rounds we only have to perform a single oracle call to find the next conforming multiplier s_i.
- Then, we require 28 705 563 calls to find the next s_i.
- Then, we require 3 633 812 calls to find the next s_i.
- Then, we require 5 776 136 calls to find the next s_i.
- Then, we require 5 897 519 calls to find the next s_i.
- After this, we always need to perform between 5 776 136 and 5 776 138 oracle calls to find the next conforming multiplier s_i.

We note that if we do not perform any improvements, this pattern for step 2c is identical.

Overall, we end up with 5 629 537 199 oracle calls for step 2c.

In total, the attack complexity with all improvements applied is 5 629 537 625.

5.3 Heuristic to Improve Step 2c

Based on the previous observation, we devised a heuristic to improve the performance of step 2c. As we can see from the three examples in Section 5.1, it only makes sense to exploit the structure of required oracle calls for the first message. For the other two examples, we would still need too many oracle calls to decrypt it because we need an enormous number of oracle calls for finding one s_i before the pattern occurs. Hence, for the "bad" cases, we want to learn the pattern to improve their performance and for the "very bad" cases of step 2c, we want to abort the attack.

We now explain how we implemented this heuristic:

Let c_e be the expected number of oracle calls to find the next conforming multiplier s_i in step 2c, see the theoretical analysis in Section 3.3. For the TTT oracle, for example, we have $c_e = 2$.

First, we perform steps 2a and 2b with all previously introduced improvements. Then, we do a single round of step 2c, meaning we search for the next conforming multiplier s_i. Let c_i be the number of oracle calls required for finding this value s_i. If this value is very large, we know that we have come across a very bad case. Hence, if $c_i > 500 \cdot c_e$, we abort the attack. Else if c_i is larger than expected, but not too large, i.e., $5 \cdot c_e < c_i < 500 \cdot c_e$, we want to improve the performance of step 2c. In this case, we learn the pattern of oracle calls required for finding the next s_i in step 2c. More precisely, we aim to identify the number of values we would always unnecessarily test with the oracle and then skip these tests. As soon as we have found this offset, step 2c should be efficient again. The third case which can arise is that $c_i \leq 5 \cdot c_e$. In that case, step 2c behaves as expected and we continue with the next round of step 2c. For this next step, we again consider c_{i+1} to decide whether we want to abort, start the pattern recognition or continue as usual.

Now, we describe how we aim to find the offset value if $5 \cdot c_e < c_i < 500 \cdot c_e$. We continue by searching for s_{i+1} and s_{i+2}. When doing this, we save the values c_{i+1} and c_{i+2}. From the observation in Section 5.1, it follows that these numbers should follow a pattern after some time. If we find the desired offset and always skip testing this number of s_i values we would otherwise send to the oracle, we can significantly lower the complexity of step 2c for these bad cases. We provide an easy heuristic with the aim to find the pattern and show the experimental results in the following section. First, we compute $c_{min} = \min(c_{i+1}, c_{i+2})$. Then, we define $c_{skip} = \max(c_{min} - c_e, 0)$. This is the number of values that we, from now on, do not test with the oracle. If we have already computed a value for c_{skip} and observe that $5 \cdot c_e < c_i < 500 \cdot c_e$ later in the attack, we also count c_{i+1} and c_{i+2} in the following two rounds. The new number of values to skip is then defined as: $c_{skip} = \max(c_{skip} + c_{min} - c_e, 0)$.

This is a heuristic procedure to improve the performance of step 2c by exploiting its pattern of oracle calls. All values were derived by using experimental results and one could possibly find better values. However, as we will see in the experimental results for the heuristic, this procedure already brings down the mean of step 2c nicely and eliminates the very bad cases that are still too expensive. Because these cases are rare, the success probability remains high.

5.4 Experimental Results

We now look at the experimental results obtained by using this procedure. In particular, we compare the performance of Bleichenbacher's attack with all improvements to its complexity when adding the heuristic.

Since the technique only modifies step 2c, the complexity of steps 2a and 2b stays the same. We can see that the heuristic lowers the mean of step 2c a lot, bringing it close to its median. This results from the fact that we improve the "bad" cases to roughly reach the expected performance and eliminate the "very bad" cases of step 2c. Additionally, the experimental results show that the attack still has a high success probability with this heuristic as the extremely bad cases are rare.

The improvement on the mean of the total complexity depends on how much this complexity is influenced by step 2c. Hence, we see a significant decrease in the total mean for the TTT and TFT oracles, whereas the improvement is not as notable for the FFT oracle and the BVO.

For this heuristic, we include additional plots for visualizing the improvement on step 2c. In these figures, the x-axis still describes the number of oracle calls. However, on the y-axis, we now display the frequency of experiments requiring at least that number of oracle calls to complete step 2c. Essentially, we plot $1 - CDF$, where CDF is the cumulative distribution function of the number of oracle calls for step 2c over the experiments.

5.4.1 TTT Oracle

For the TTT oracle, we can see in Table 5.2 that we are able to reduce the mean of step 2c significantly. The mean and median values for step 2c are now almost identical. In Figure 5.1, it is visible that the long tail of the distribution of oracle calls required for step 2c, present in Figure 3.1, is gone. Additionally, this distribution is more concentrated around the expected number of oracle calls for step 2c.

The influence on the mean of the total complexity is also significant for this oracle because, after all improvements, the total performance depends on steps 2a and 2c, see Figure 5.1. The effect is visible in Table 5.1. Furthermore, the success probability is very high. We only abort the algorithm for 0.3% of the messages, see Table 5.1.

Figure 5.2 further illustrates that the probability of running into a bad case is a lot lower with the heuristic. For example, without this heuristic, the probability of having to perform at least 2^{12} oracle calls for a keylength of 512 bits in step 2c is larger than 5%. However, with the heuristic, this probability is almost zero. Additionally, the figure also visualizes that the heuristic brings the mean value very close to its median.

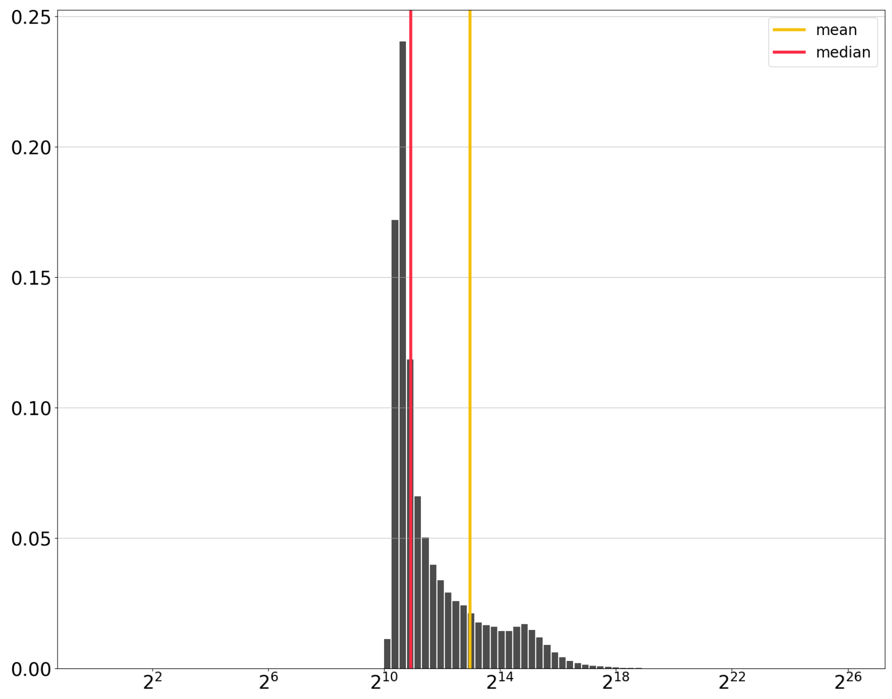
We omit the figures for the other keylengths as this heuristic produces a similar shape for their distribution of step 2c.

keylength	success probability	median total with heuristic	mean total with heuristic	median total with improvements	mean total with improvements
512	0.997	1 933	7 998	2 114	10 007
1024	0.997	3 192	9 228	3 384	12 641
2048	0.997	5 639	11 704	5 860	19 012

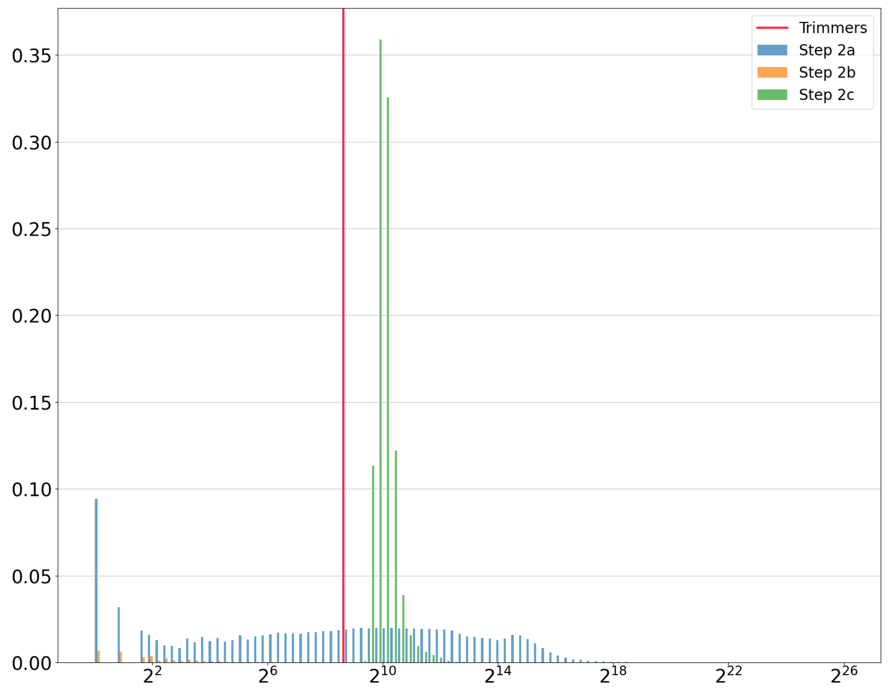
Table 5.1: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – TTT Oracle – 10^6 Simulations – Total

keylength	median 2c with heuristic	mean 2c with heuristic	median 2c with improvements	mean 2c with improvements
512	1 021	1 112	1 029	3 121
1024	2 107	2 285	2 126	5 711
2048	4 278	4 635	4 320	11 948

Table 5.2: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads and Heuristics for 2c – TTT Oracle – 10^6 Simulations – Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 5.1: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – TTT Oracle – 10^6 Simulations – keylength 512 – success probability of 0.997

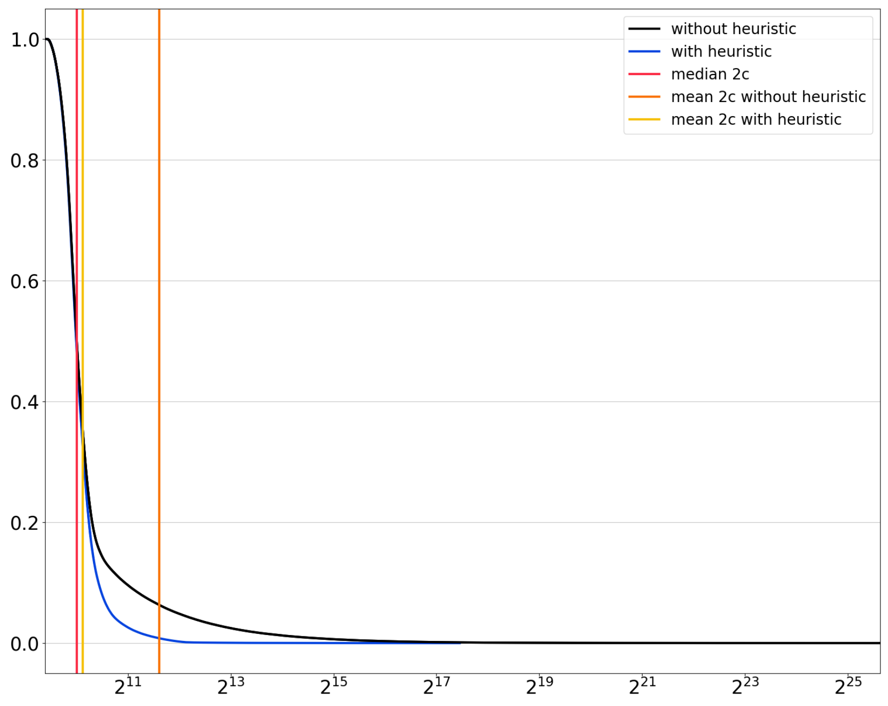


Figure 5.2: 1 - CDF of Step 2c – TTT Oracle – 10^6 Simulations – keylength 512 – success probability of 0.997 with heuristic

5.4.2 TFT Oracle

The observations are exactly the same as for the TTT oracle. Because the influence of the heuristic on the distribution of step 2c looks identical to Figures 5.1 and 5.2, we omit them.

keylength	success probability	median total with heuristic	mean total with heuristic	median total with improvements	mean total with improvements
512	0.997	2 254	8 935	2 479	10 980
1024	0.997	3 590	10 232	3 797	14 278
2048	0.997	6 214	12 852	6 355	19 605

Table 5.3: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – TFT Oracle – 10^6 Simulations – Total

keylength	median 2c with heuristic	mean 2c with heuristic	median 2c with improvements	mean 2c with improvements
512	1 069	1 163	1 071	3 209
1024	2 208	2 397	2 213	6 348
2048	4 485	4 862	4 497	11 543

Table 5.4: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads and Heuristics for 2c – TFT Oracle – 10^6 Simulations – Step 2c

5.4.3 FFT Oracle

From Table 5.6, we can see that also for the FFT oracle, the heuristic brings the mean of step 2c close to its median. However, for this oracle type, because its total complexity is significantly larger than the complexity required for step 2c, the heuristic does not substantially improve the total mean of the attack, see Table 5.5. It is important to note that for this oracle type, the impact of the heuristic increases with increasing keylength. Furthermore, we obtain an even higher success probability of 99.8% or 99.9%.

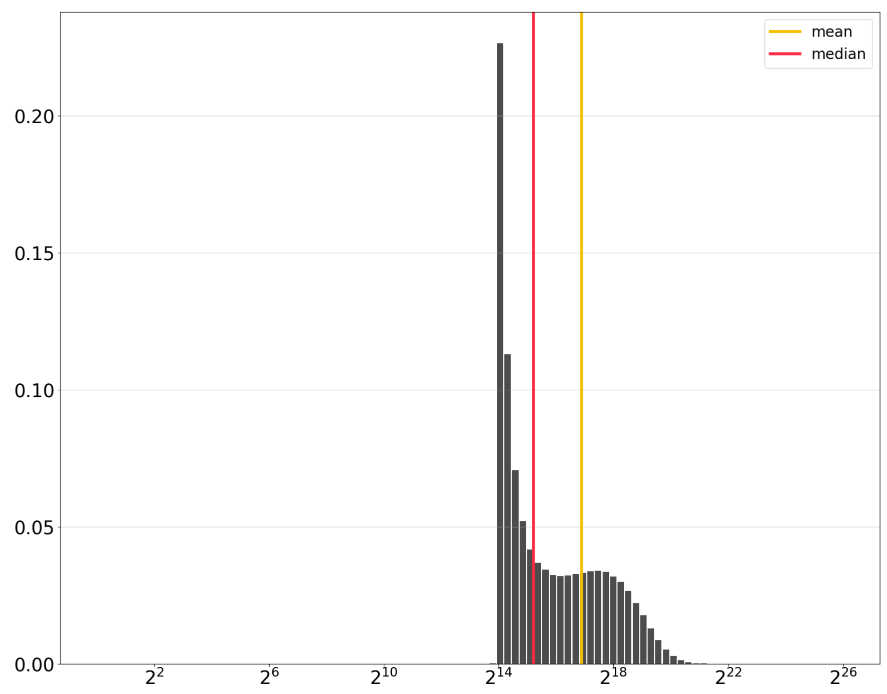
Figures 5.3, 5.5 and 5.7 show that the long tail of the distribution of oracle calls required for step 2c is removed compared to Figures 4.11, 4.12 and 4.13. Furthermore, Figures 5.4, 5.6 and 5.8 visualize that the heuristic improves the "bad" cases and removes the "very bad" cases.

keylength	success probability	median total with heuristic	mean total with heuristic	median total with improvements	mean total with improvements
512	0.999	37 770	122 142	38 464	123 821
1024	0.999	15 933	50 827	16 137	54 058
2048	0.998	11 428	28 016	11 393	34 118

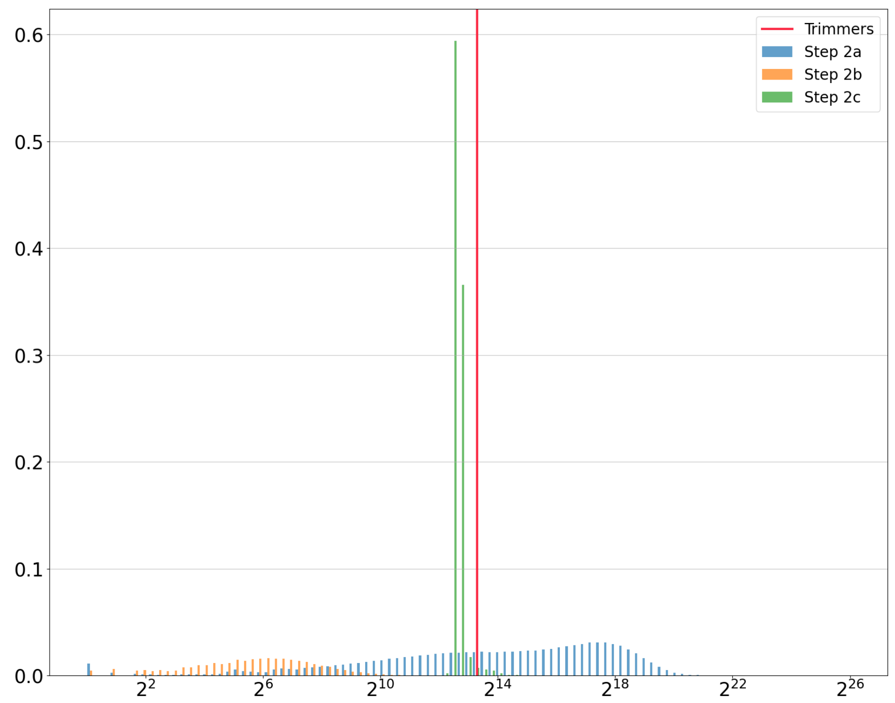
Table 5.5: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – FFT Oracle – 10^6 Simulations – Total

keylength	median 2c with heuristic	mean 2c with heuristic	median 2c with improvements	mean 2c with improvements
512	6 143	6 337	6 141	7 881
1024	6 405	6 780	6 396	9 789
2048	7 567	8 225	7 545	14 231

Table 5.6: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads and Heuristics for 2c – FFT Oracle – 10^6 Simulations – Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 5.3: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – FFT Oracle – 10^6 Simulations – keylength 512 – success probability of 0.999

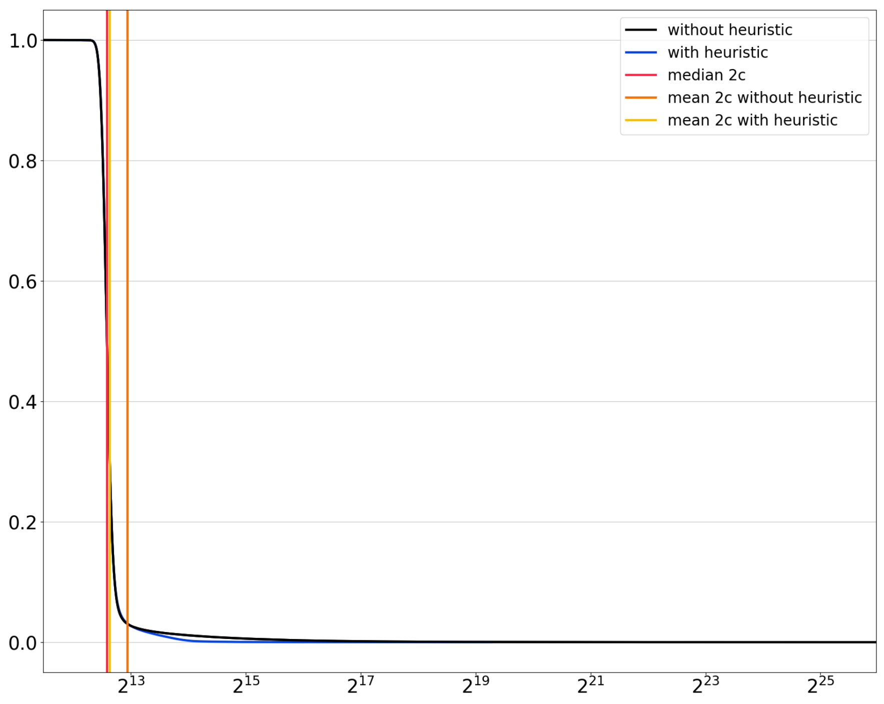
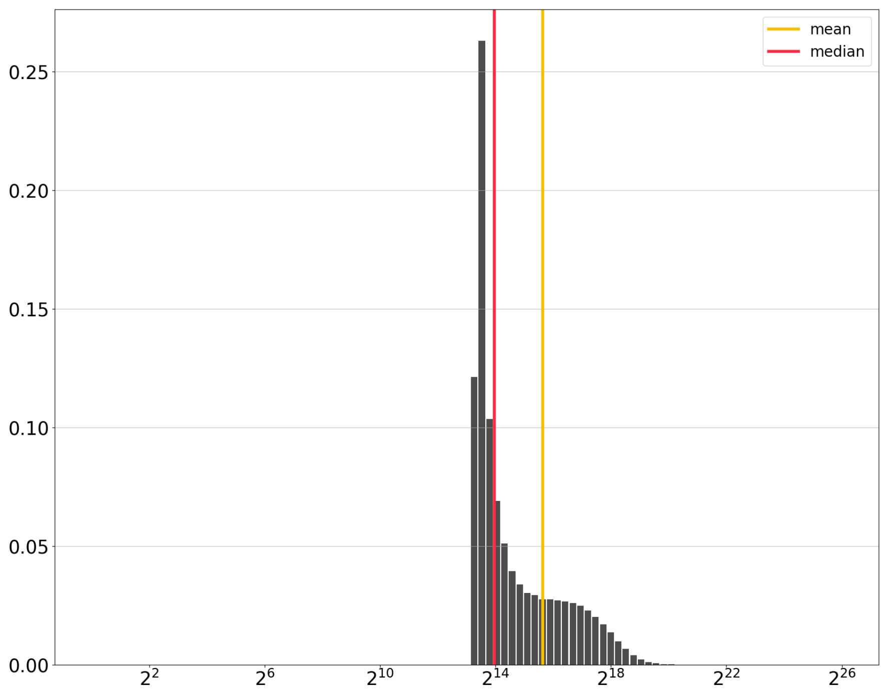
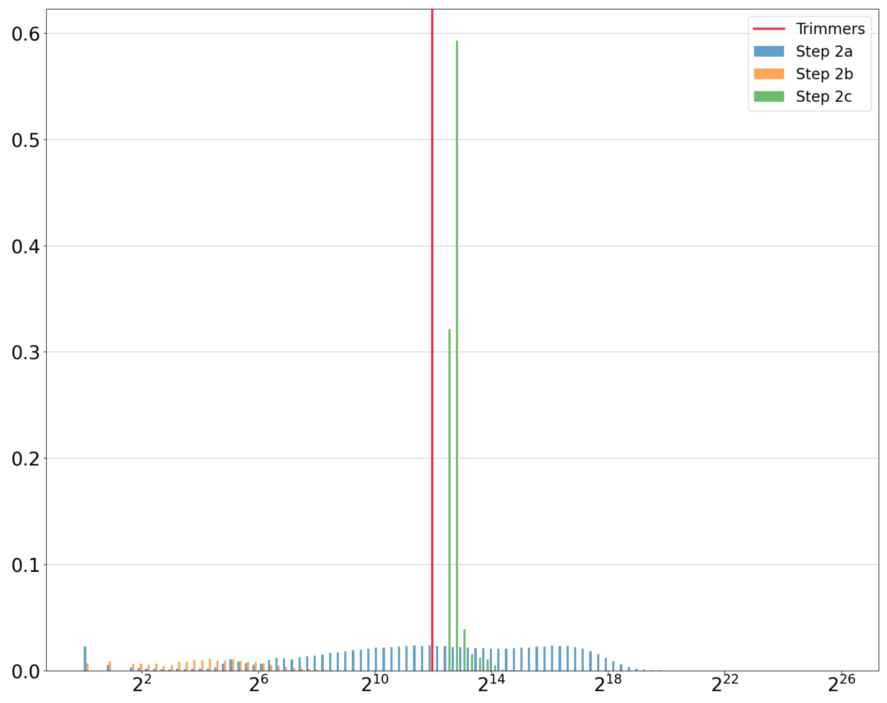


Figure 5.4: 1 - CDF of Step 2c – FFT Oracle – 10^6 Simulations – keylength 512 – success probability of 0.999 with heuristic



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 5.5: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – FFT Oracle – 10^6 Simulations – keylength 1024 - success probability of 0.999

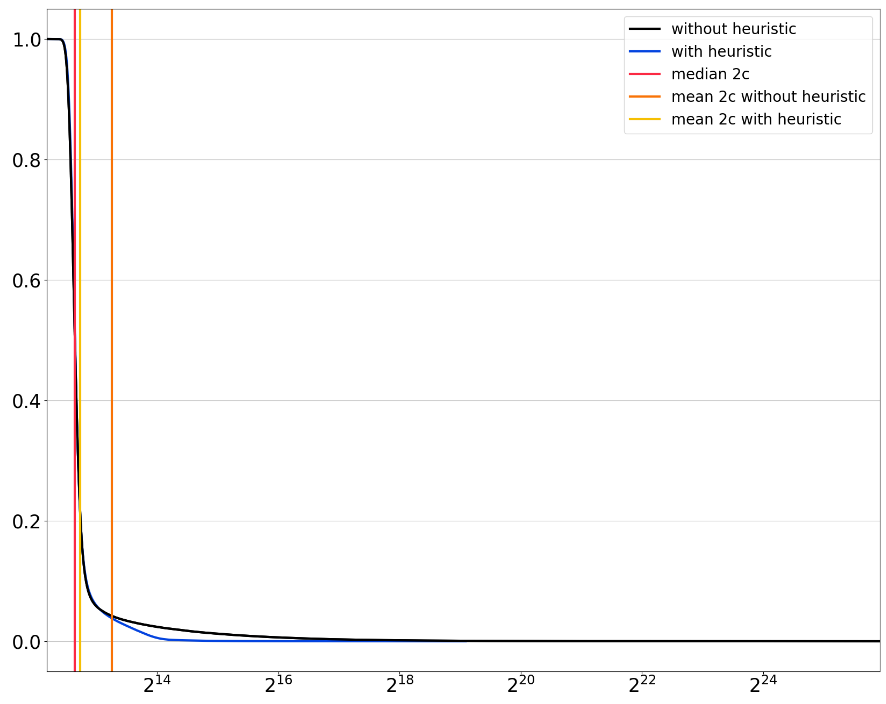
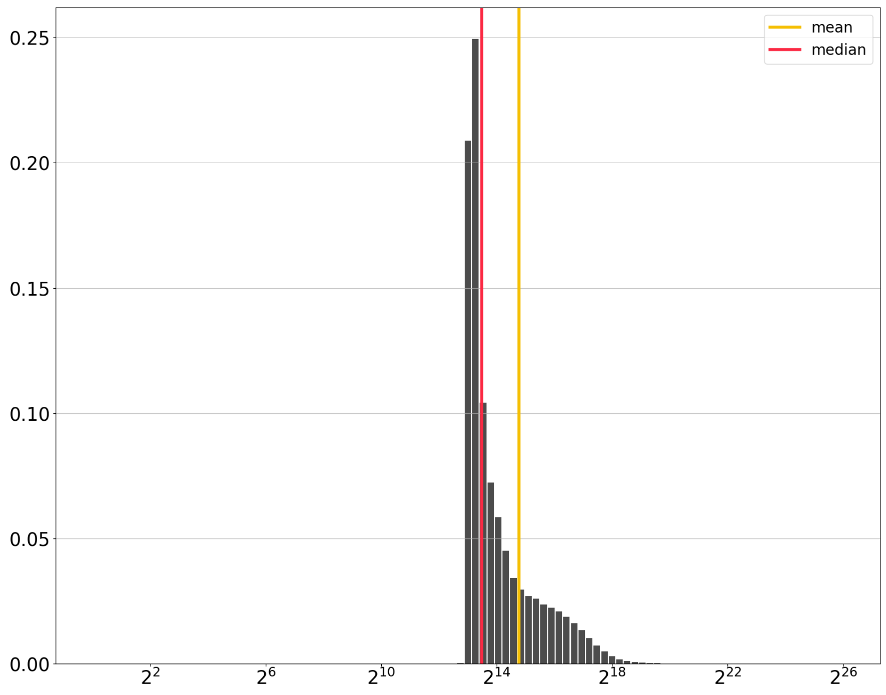
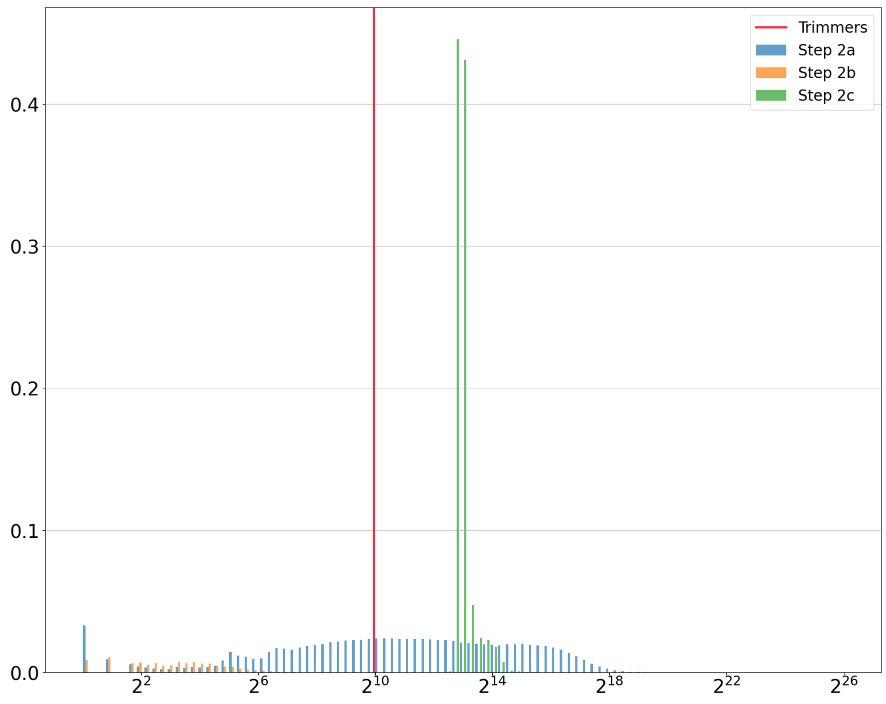


Figure 5.6: 1 - CDF of Step 2c – FFT Oracle – 10^6 Simulations – keylength 1024 - success probability of 0.999 with heuristic



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 5.7: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – FFT Oracle – 10^6 Simulations – keylength 2048 - success probability of 0.998

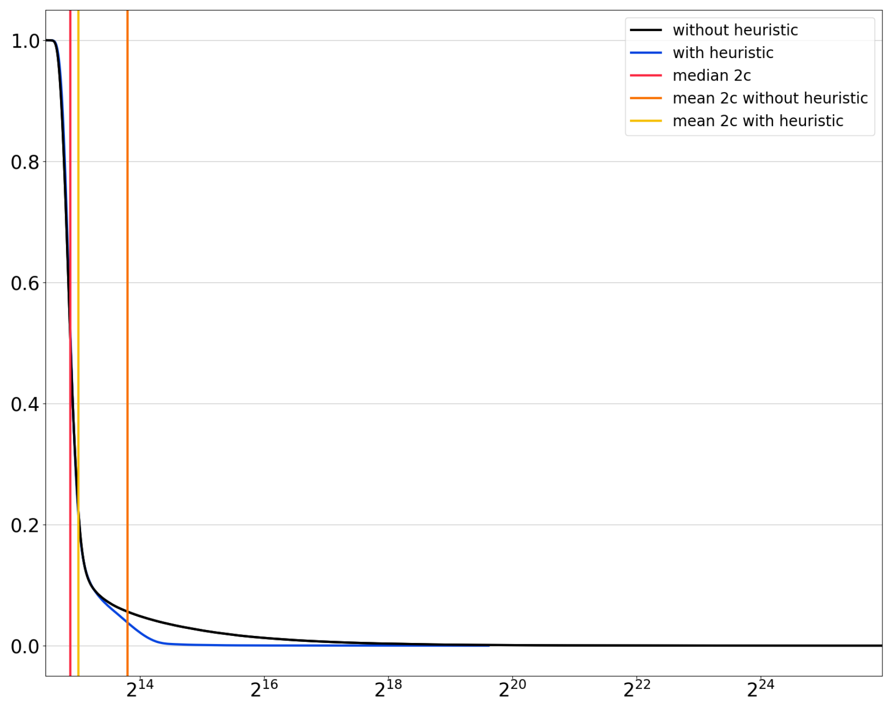


Figure 5.8: 1 - CDF of Step 2c – FFT Oracle – 10^6 Simulations – keylength 2048 - success probability of 0.998 with heuristic

5.4.4 Bad Version Oracle

For the BVO, we can observe that the heuristic improves the mean of step 2c significantly, see Table 5.8. However, as the total complexity for this oracle is a lot larger, this does not provide a notable improvement to the total mean, see Table 5.7. We note that the mean values are also affected by randomness of the experiments because we only performed 10 000 simulations for the BVO.

Figure 5.9 removes the long tail of the number of oracle calls for step 2c compared to Figure 4.14 without the heuristic. Additionally, Figure 5.10 showcases that the heuristic improves the attack complexity for a lot of the bad cases significantly.

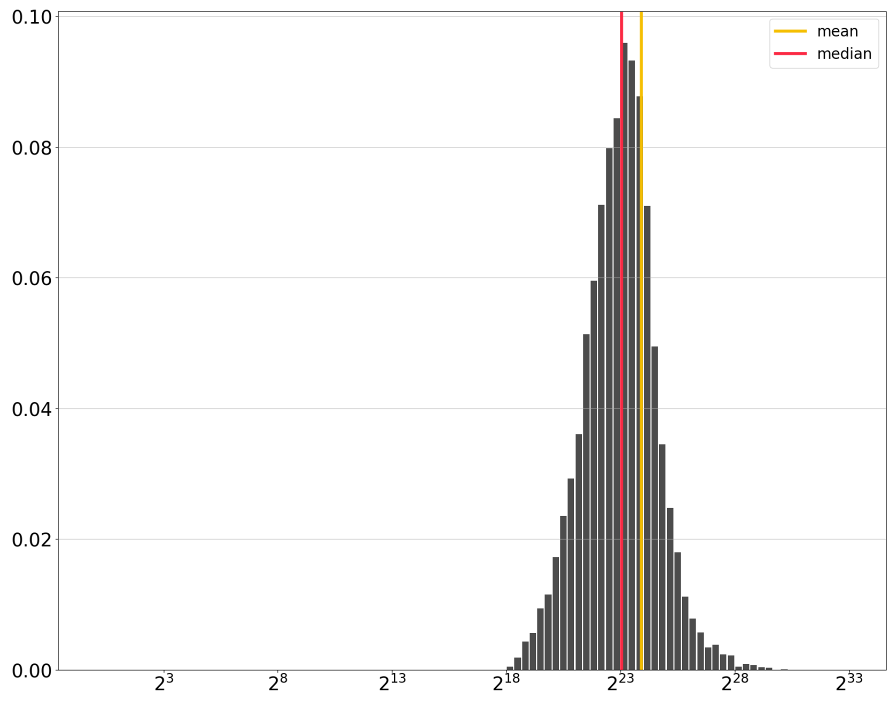
We do not provide the figures for the additional keylenghts as the improvement on step 2c looks very similar.

keylength	success probability	median total with heuristic	mean total with heuristic	median total with improvements	mean total with improvements
512	0.996	8 677 741	15 847 517	8 889 976	17 057 401
1024	0.998	11 654 632	19 591 262	12 033 457	21 122 774
2048	1	22 121 516	34 508 968	20 563 963	33 706 806

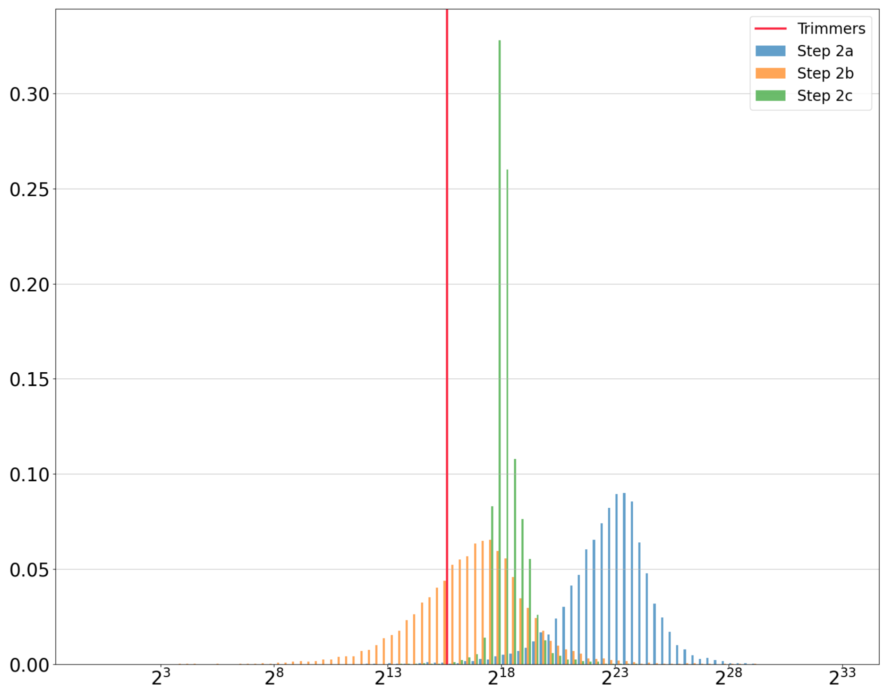
Table 5.7: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – BVO – 10^4 Simulations – Total

keylength	median 2c with heuristic	mean 2c with heuristic	median 2c with improvements	mean 2c with improvements
512	272 088	365 335	274 367	841 390
1024	731 733	942 994	742 132	1 400 068
2048	2 546 302	2 920 683	2 583 223	3 706 822

Table 5.8: Bleichenbacher's Attack with Skipping Holes, Trimmers and Parallel Threads and Heuristics for 2c – BVO – 10^4 Simulations – Step 2c



(a) Total Distribution



(b) Distribution of Individual Steps

Figure 5.9: Bleichenbacher's Attack with Skipping Holes, Trimmers, Parallel Threads and Heuristics for 2c – BVO – 10^4 Simulations – keylength 512 - success probability of 0.996

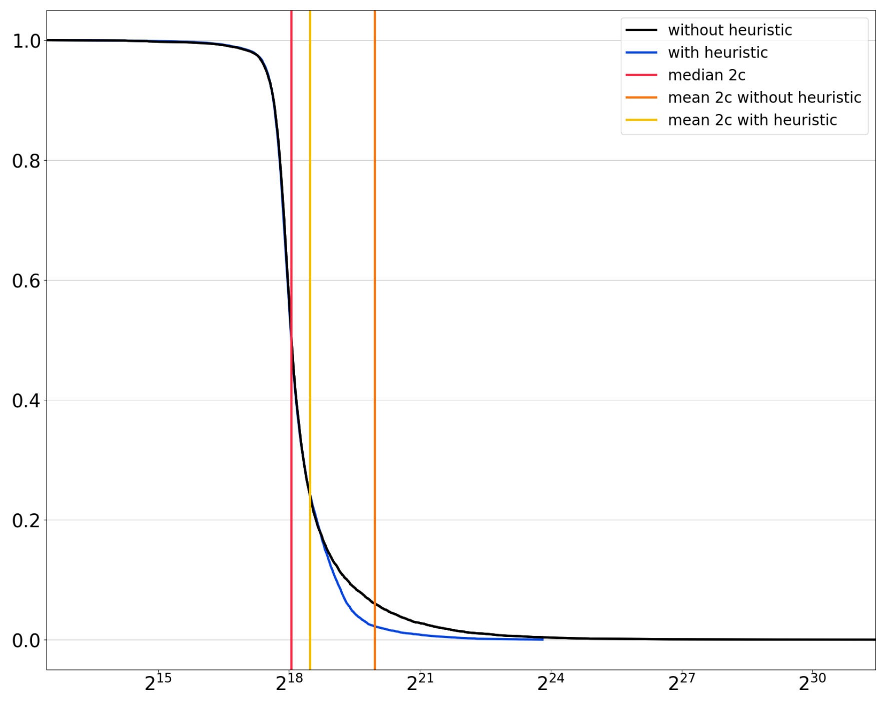


Figure 5.10: 1 - CDF of Step 2c – BVO – 10^4 Simulations – keylength 512 - success probability of 0.996 with heuristic

Chapter 6

Improvements Overview

In this chapter, we provide tables containing the experimental results of all improvements we tested. In each table, one row corresponds to a particular oracle type and keylength combination. The column headers describe the applied improvements on the attack and whether this column displays the median or mean value. We use the following abbreviations for the different optimizations: Parallel Threads Method (PT), Skipping Holes (SH), Trimmers (T), Heuristic for Step 2c (H2c). For the improvements, we used the best parameters found during the experiment runs, see Chapter 4. We note that when applying the heuristic for step 2c (H2c), we might abort for some messages. Hence, we have a slightly lower success probability. However, as seen in the experimental results in section 5.4, the likelihood of aborting is very low, namely at most 0.4%.

The first table presents the median and mean values of the total number of oracle calls required for the different attack variants, see Table 6.1. Additionally, we also provide a table for each step of the algorithm, see Tables 6.2, 6.3 and 6.5. Finally, one table displays the number of rounds required of step 2b when different optimizations are used, see Table 6.4.

We point out that for the TTT, TFT and FFT oracles, the improvements reduce the median of the total attack complexity by up to a factor of 10, see Table 6.1. Hereby, we are able to optimize the attack more for smaller keylengths. For the BVO, we obtain an improvement of up to a factor of 3 when applying all optimizations.

oracle and keylength	median total	mean total	median total PT	mean total PT	median total SH	mean total SH	median total PT, SH, T	mean total PT, SH, T	median total PT, SH, T H2c	mean total PT, SH, T H2c
TTT										
512	21 114	36 272	21 142	30 294	14 746	28 937	2 114	10 007	1 933	7 998
1024	23 000	40 815	23 048	33 626	16 741	35 314	3 384	12 641	3 192	9 228
2048	26 439	49 767	26 474	39 417	20 230	41 225	5 860	19 012	5 639	11 704
TFT										
512	22 077	39 530	22 091	31 888	15 658	32 854	2 479	10 980	2 254	8 935
1024	24 010	42 981	23 920	35 041	17 637	37 243	3 797	14 278	3 590	10 232
2048	27 486	51 362	27 519	41 426	21 143	42 475	6 355	19 605	6 214	12 852
FFT										
512	359 198	434 687	151 822	222 568	338 383	416 462	38 464	123 821	37 770	122 142
1024	153 859	200 970	76 510	113 759	134 492	185 089	16 137	54 058	15 933	50 827
2048	50 191	109 099	46 787	71 346	42 177	94 632	11 393	34 118	11 428	28 016
BVO										
512	24 227 787	50 146 387	8 928 141	17 141 061	24 351 955	47 491 472	8 889 976	17 057 401	8 677 741	15 847 517
1024	31 076 365	62 343 618	11 392 567	19 728 955	31 158 826	58 692 156	12 033 457	21 122 774	11 654 632	19 591 262
2048	51 200 904	75 265 298	22 487 282	34 072 121	50 090 158	71 788 213	20 563 963	33 706 806	22 121 516	34 508 968

Table 6.1: Bleichenbacher's Attack – 10^6 Simulations for TTT, TFT, FFT / 10^4 Simulations for BVO – Total

oracle and keylength	median 2a	mean 2a	median 2a PT	mean 2a PT	median 2a SH	mean 2a SH	median 2a PT, SH, T	mean 2a PT, SH, T	median 2a PT, SH, T H2c	mean 2a PT, SH, T H2c
TTT										
512	19 178	27 180	19 212	27 253	12 417	23 121	316	6 474	319	6 473
1024	19 318	27 361	19 362	27 447	12 497	23 236	329	6 518	330	6 531
2048	19 642	27 841	19 391	27 970	12 699	23 900	338	6 651	338	6 656
TFT										
512	20 115	28 801	20 145	28 925	13 373	24 656	520	7 258	523	7 258
1024	20 259	29 017	20 180	28 987	13 392	24 796	540	7 416	545	7 322
2048	20 517	29 421	20 591	29 583	13 553	25 175	550	7 549	555	7 478
FFT										
512	144 030	213 858	144 645	214 488	137 592	208 176	21 153	105 768	21 257	105 636
1024	68 647	103 614	68 537	103 646	61 751	97 733	4 677	40 203	4 659	39 977
2048	37 019	56 882	37 101	57 075	30 014	51 431	1 804	18 849	1 808	18 759
BVO										
512	8 019 905	14 499 301	8 133 738	14 860 436	8 149 334	15 560 490	8 097 901	14 317 720	8 085 056	14 455 793
1024	10 548 557	18 231 944	9 974 007	17 012 623	10 369 855	17 609 473	10 511 671	17 590 261	10 391 351	17 332 908
2048	17 708 014	27 544 262	18 459 221	29 204 890	17 328 475	27 369 100	16 249 408	28 613 532	18 455 472	27 404 393

Table 6.2: Bleichenbacher's Attack – 10^6 Simulations for TTT, TFT, FFT / 10^4 Simulations for BVO – Step 2a

keylength	median 2b	mean 2b	median 2b PT	mean 2b PT	median 2b SH	mean 2b SH	median 2b PT, SH, T	mean 2b PT, SH, T	median 2b PT, SH, T H2c	mean 2b PT, SH, T H2c
TTT										
512	0	5 634	0	2.63	0	2 642	0	0.82	0	0.90
1024	0	5 675	0	4.02	0	2 617	0	0.67	0	0.58
2048	0	5 829	0	2.89	0	2 658	0	0.61	0	0.54
TFT										
512	0	6 969	0	6.68	0	3 904	0	0.76	0	0.72
1024	0	7 000	0	5.32	0	3 816	0	2.14	0	0.67
2048	0	6 911	0	4.27	0	3 970	0	0.80	0	0.63
FFT										
512	121 714	212 733	45	132	107 318	200 315	0	43	0	40
1024	32 483	84 762	6	33	19 590	75 148	0	12	0	9
2048	0	33 371	0	15	0	27 117	0	9	0	3
BVO										
512	11 113 772	34 805 695	149 303	1 606 714	10 917 494	30 539 779	112 155	1 661 303	113 933	976 258
1024	13 778 521	42 711 607	219 051	1 486 774	13 719 988	39 515 107	190 402	1 856 419	185 176	1 265 260
2048	22 443 601	44 014 214	610 865	1 589 241	21 853 72	41 223 785	492 769	1 721 121	488 930	4 103 730

Table 6.3: Bleichenbacher's Attack – 10^6 Simulations for TTT, TFT, FFT / 10^4 Simulations for BVO – Step 2b

keylength	median rounds 2b	mean rounds 2b	median rounds 2b PT	mean rounds 2b PT	median rounds 2b SH	mean rounds 2b SH	median rounds 2b PT, SH, T	mean rounds 2b PT, SH, T	median rounds 2b PT, SH, T H2c	mean rounds 2b PT, SH, T H2c
TTT										
512	0	0.21	0	0.18	0	0.21	0	0.03	0	0.03
1024	0	0.21	0	0.18	0	0.21	0	0.03	0	0.03
2048	0	0.21	0	0.18	0	0.21	0	0.03	0	0.03
TFT										
512	0	0.23	0	0.20	0	0.24	0	0.04	0	0.04
1024	0	0.23	0	0.20	0	0.23	0	0.04	0	0.04
2048	0	0.23	0	0.20	0	0.23	0	0.04	0	0.04
FFT										
512	1	0.95	1	0.92	1	0.95	0	0.34	0	0.34
1024	1	0.75	1	0.71	1	0.75	0	0.21	0	0.21
2048	0	0.51	0	0.48	0	0.52	0	0.12	0	0.12
BVO										
512	1	2.68	1	1.28	1	2.39	1	1.23	1	1.26
1024	1	2.48	1	1.27	1	2.11	1	1.25	1	1.25
2048	1	1.78	1	1.24	1	1.66	1	1.23	1	1.22

Table 6.4: Bleichenbacher's Attack – 10^6 Simulations for TTT, TFT, FFT / 10^4 Simulations for BVO – Rounds of Step 2b

keylength	median 2c	mean 2c	median 2c PT	mean 2c PT	median 2c SH	mean 2c SH	median 2c PT, SH, T	mean 2c PT, SH, T	median 2c PT, SH, T H2c	mean 2c PT, SH, T H2c
TTT										
512	1 030	3 458	1 030	3 038	1 029	3 210	1 029	3 121	1 021	1 112
1024	2 126	7 779	2 126	6 174	2 126	9 461	2 126	5 711	2 107	2 285
2048	4 320	16 097	4 318	11 444	4 321	14 666	4 320	11 948	4 278	4 635
TFT										
512	1 071	3 759	1 071	2 956	1 071	4 295	1 071	3 209	1 069	1 163
1024	2 213	6 964	2 214	6 048	2 214	8 632	2 213	6 348	2 208	2 397
2048	4 497	15 030	4 496	11 839	4 497	13 330	4 497	11 543	4 485	4 862
FFT										
512	6 134	8 097	6 128	7 947	6 134	7 971	6 141	7 881	6 143	6 337
1024	6 394	12 593	6 392	10 080	6 394	12 208	6 396	9 789	6 405	6 780
2048	7 545	18 846	7 544	14 256	7 542	16 083	7 545	14 231	7 567	8 225
BVO										
512	274 367	841 390	273 782	673 912	274 748	1 391 203	273 119	1 028 229	272 088	365 335
1024	742 132	1 400 068	753 824	1 229 558	745 842	1 537 576	742 771	1 625 982	731 733	942 994
2048	2 583 223	3 706 822	2 589 309	3 277 990	2 476 109	3 456 412	2 595 186	3 292 010	2 546 302	2 920 683

Table 6.5: Bleichenbacher's Attack – 10^6 Simulations for TTT, TFT, FFT / 10^4 Simulations for BVO – Step 2c

Chapter 7

Conclusion

In this thesis, we gave a detailed description of Bleichenbacher's attack with the goal to provide intuition behind its mechanism and why it succeeds. Following this, we explored the effects that different keylengths and oracle types have on the attack complexity by providing a theoretical analysis and tying it to practical results of large-scale experiments. This was done to analyze different situations that arise in practice. Hereby, we applied a larger number of tests than in the existing literature to lower the influence of statistical fluctuations.

Furthermore, we provided a comprehensive overview of all improvements, describing their technique and mathematical background precisely. We gave details relevant for implementing the different optimizations that are absent from the original papers. Additionally, we devised slight variations of the described improvements to increase their efficiency. For this, we performed extensive research into the best parameters for the implementation of the different optimizations, especially for the Trimmers method. Finally, we conducted a large number of simulations for the presented improvements on their own and in combination with each other to analyze their influence on the Bleichenbacher attack. The results have shown that combining the optimization methods significantly lowers the median and mean complexity for all oracle types and keylengths.

Based on rare experiments observed in the large number of simulations, we then introduced a novel heuristic for improving step 2c. The experimental results for the optimizations with the heuristic suggest a further improvement in the attack's mean and median values. We now compare these to the results of previous publications.

Klíma, Pokorný and Rosa [8] presented practical results of the Bad Version Oracle. They conducted 1 200 experiments for both a keylength of 1024 and 2048 bits. We observe that for a keylength of 1024 bits, our achieved median and mean values of 11 654 632 and 19 591 262 are slightly better than the values of 13 331 256 and 20 835 297 they obtained. However, for a modulus of 2048 bits, we obtained a median of 22 121 516 and a mean of 34 508 968, which is higher compared to their values of 19 908 079 and 28 728 801. We note that the randomness of the experiments heavily influences the results since they only performed 1 200 simulations. We conducted 10 000 simulations, which may still not be enough to obtain a reasonable estimate of the mean value. Additionally, for a keylength of 2048 bits, our implementation without the heuristic has a significantly lower median value compared to the results with the heuristic, also showcasing the influence of the small number of experiments on the median values for the BVO.

Bardou et al. [2] performed 1 000 experiments with a keylength of 1024 bits for the TTT, TFT, FFT and FFF oracles. For the TTT oracle, our final implementation with a median of 3 192 and a mean of 9 228 brings an improvement to their median of 3 768 and mean of 9 374. The same holds for the TFT oracle, where we obtain a median of 3 590 and a mean of 10 232 compared to their median of 4 014 and mean of 10 295. On the other hand, for the FFT oracle, our median and mean values of 15 933 and 50 827 are slightly higher than their values of 14 501 and 49 001. Finally, for the BVO, our median of 11 654 632 is lower than the median of 12 525 835 from their experiments. However, we obtained a higher mean value of 19 591 262 compared to a mean of 18 040 221 obtained by Bardou et al. We note that they experimented with an FFF oracle whose complexity is slightly lower compared to the BVO because we do not check the version numbers with an FFF oracle. Additionally, we can observe that our mean values are only close to values achieved by Bardou et al. after applying our new heuristic for step 2c. This suggests that because of their significantly lower number of simulations, they did not run into the very bad cases and hence their mean was already lower. Overall, our median values suggest an improvement to their algorithm for the TTT oracle, TFT oracle and BVO. However, because of the randomness and their small number of conducted experiments, this may not actually be the case. Furthermore, our implementation aborts for a small fraction of messages. However, this probability in practice was at most 0.4% and one can see that even without the heuristic, we obtained lower mean values than Bardou et al. for the TTT oracle, TFT oracle and BVO.

We can conclude that we confirmed the experimental results of [2, 8] and possibly improved some of the presented optimization techniques.

Finally, the thesis also showcased that the attack is still relevant over twenty years later. New Bleichenbacher-like attacks have appeared in practice recently and will probably also in the future. So, we point out the importance of implementing the decryption process of a message carefully without leaking any side-channel information. Furthermore, a solution to the attack would be to switch away from PKCS #1 v1.5 to a better encoding scheme, e.g., RSA-OAEP [11], which is provably IND-CCA secure [7]. However, also for the RSA-OAEP encryption scheme, it is essential to implement it carefully as otherwise, the padding scheme is vulnerable to Manger's attack [9]. Lastly, one should consider using a key exchange method different from RSA to thwart the Bleichenbacher attack. This is already done in TLS 1.3 [13], where RSA is not an option anymore and one has to use the Diffie-Hellman key exchange protocol.

Appendix A

Appendix

A.1 Implementation of the Bleichenbacher attack

```
import os
import math
import time
import numpy as np
import multiprocessing as mp
from aes_random import AES_Random
import pandas as pd

#histograms for this experiment, will contain the results for all
    simulations
hist = []  #total oracle calls
hista = [] #oracle calls for step 2.a
histb = [] #oracle calls for step 2.b
histc = [] #oracle calls for step 2.c

hist_roundsb = [] #number of rounds of 2b required
hist_intervals2b = [] #how many intervals there were before each
    round of 2b

hist_utmin = [] #(u, t) pair used for trimming lower bound on
    message
hist_utmax = [] #(u, t) pair used for trimming upper bound on
    message
histt = [] #oracle calls required for trimmers
histtrim_value = [] #how much larger initial interval was compared
    to trimmed interval

index = [] #to keep track which index returned which result

#keylength, k and B for these simulations
keylength = 0
k = 0
B = 0
```

```
#RSA keys
n = 0
e = 0
d = 0

#Random number generator
aes_random = AES_Random((0).to_bytes(16, 'big'))

#Bounds on the message
a = 0  #lower bound on the message
b = 0  #upper bound on the message

#Specify which oracle is used, initially all are set to false and we
     set the corresponding one to True when initializing
#every oracle checks that the first two bytes are 0x00 and 0x02
check_eight_nonzero = False   #if True oracle also checks whether
    bytes 3-10 are nonzero
check_one_zero = False   #if True oracle also checks whether there
    exists one zero byte in bytes 11-k (used in combination with
    check_eight_nonzero)
BVO = False #if True bad version oracle for SSL/TLS messanges,
    checks whether plaintext is S-PKCS conforming and the version
    number is wrong (used on its own with check_eight_nonzero and
    check_one_zero = False)

v_major = 3 #major version number for SSL/TLS session
v_minor = 3 #minor version number for SSL/TLS session

#Improvements

#Improvements for Step 2.a (and 2.b)

#Skipping Holes
skipping_holes = False #skip all holes we can identify using the
    bounds on the message for step 2a and 2b, also start at an
    improved value of ceil((n+2B)/(b)) for 2a and += ceil(n/bmax) -
    1 for 2b

#Trimmers
trimmers = False #use trimmers to improve the performance of the
    attack
trimmer_values = 0 #how many different oracle calls we allow for
    generating and testing trimmer denominators
t_slack = 0  #slack used when testing/generating trimmer
    denominators
bs_slack = 0 #slack used in binary search for trimmer numerators

#Improvements for Step 2.b

#Parallel Threads
do_parallel_threads = False
pt_max_intervals = 0  #maximum number of intervals such that we
    perform parallel_threads, otherwise do normal 2b
```

```
#Improvements for Step 2.c

#Heuristic for Step 2.c
improve_2c = False
exp_tries = 0  #how many tries we expect to need for finding the
    next s_i value in step 2c
cutoff_2c = 0  #after how many tries to step 2c we abort


def ceil(a1, b1): #function for computing ceil(a1/b1)
    return (a1 // b1) + (a1 % b1 > 0)

def gen_message(k): #generate random message for keylength of k
    bytes

    if BVO:
        msize = 48  #where the first two bytes contain the major and
             minor version number
        message = bytearray([v_major, v_minor])
        message.extend(aes_random.random_bytes(46)) #extend major
            and minor version numbers with 46 random bytes
        return(message, msize)

    else:
        msize = aes_random.randint(k-11)  #message size in bytes,
            can be at most k-11
        return (aes_random.random_bytes(msize), msize) #generate and
             return random message of msize bytes

def pad_message(message, msize): #pad message of msize bytes
    according to PKCS#1 v1.5 standard

    padsize = k - 3 - msize

    padding = aes_random.random_nonzero_bytes(padsize)

    EB = b'\x00\x02' + padding + b'\x00' + message

    return EB

def next_s(curr_s, m, step2b, M):
    #Step 2.a - Starting the Search or Step 2.b - Searching with
        more than one interval left
    #curr_s = next s value to test, step2b = False if in step 2.a
        and True if in step 2.b

    calls = 0

    s = curr_s

    global a
    global b

    if skipping_holes:
        if (not step2b): #improved starting value if skipping holes
            and in step 2a
            s = ceil(n+2*B, b)

        else:
            #we are in step 2b so we know we can increase the
                starting value

            #compute interval in which we know the message has to
                lie
            amin = b
            bmax = a

            for (a1, b1) in M:
                if a1 < amin:
                    amin = a1
                if b1 > bmax:
                    bmax = b1

            #update bounds using the computed interval
            a = amin
            b = bmax

            #skip all s values we know cannot work (which are ceil(n
                , m) - 1 >= ceil(n, b) - 1 many)
            s += ceil(n, b) - 1

    #compute first interval which can be skipped
    j = 1
    low = ceil(3 * B + j * n, a)
    high = ceil(2 * B + (j + 1) * n, b) - 1

    while True:
        #test and increment s until we find a working s value

        if skipping_holes and not(j == -1): #check whether we can
            skip current s value (if we have not already skipped all
             possible intervals we can identify for skipping holes)
            s, j, low, high = check_and_skip_s(s, j, low, high)

        calls += 1

        if oracle(s, m):
            return s, calls

        s += 1

def next_s2(a1, b1, curr_s, m, skip_nr):
    #Step 2.c - Searching with one interval left

    r  = ceil(2*(b1*curr_s - 2*B), n)
    calls = 0
    while True:
        slow = ceil((2*B + r*n), b1)
        shigh = ceil((3*B + r*n), a1)

        for s in range(slow, shigh):

            if improve_2c and calls > cutoff_2c: #if we are applying
                 heuristic and calls to find next s_i is very high
                we abort as we are in a "very bad" case that cannot
                be saved
                return Exception
            elif improve_2c and skip_nr > 0: #skip testing some s_i
                values if we are applying heuristic and have
                identified a skip_nr (and there are still values we
                should skip)
                skip_nr -= 1
            else:
                calls += 1
                if oracle(s, m):
                    return s, calls

        r += 1

def one_step_next_s2(a1, b1, r, curr_s, m, calls):
    #version of step 2.c which always only does one step and returns
         whether this step has been successful or not for the
        Parallel Threads Method
    #r = current r value, curr_s = current s value to test

    calls += 1

    #compute highest s value for this r
    shigh = ceil((3*B + r*n), a1) - 1

    while curr_s > shigh: #if curr_s is higher than shigh we keep
        increasing r until we find a curr_s which we can test
        r += 1
        curr_s = ceil((2*B + r*n), b1)
        shigh = ceil((3 * B + r * n), a1) - 1

    if oracle(curr_s, m):
        return True, curr_s, r, calls #if curr_s works we return
            this s value and return True (now, we can update all
            intervals in the parallel threads method)
    else:
        #if curr_s does not work we return False, the current r
            value as well as the next s value to test which is
            curr_s + 1
        curr_s +=1
        return False, curr_s, r, calls

def parallel_threads(M, message, s, intervals2b):
    #perform parallel threads improvement
    calls = 0
    rounds_b = 0 #counts how many s values have to be found to be
        done with 2b / until we have only one interval left and can
        continue with step 2c

    state = gen_state(M, s)

    while True: #we continue searching in a round-robin fashion
        until we find an s value accepted by the oracle / we have
        only one interval left

        for i in range(len(M)):
            #do one step for the current interval
            success, curr_s, r, calls = one_step_next_s2(M[i][0], M[
                i][1], state[i][0], state[i][1], message, calls)

            if success:
                #update all intervals if we found an s value (and
                    eliminate empty ones)
                M = update_intervals(M, curr_s)
                rounds_b += 1

                if len(M) == 1:  #if we only have one interval left
                    we can continue with step 2c
                    return calls, curr_s, M, rounds_b, intervals2b

                else: #if we have more than one interval left after
                    the update we update the states and start with
                    the parallel threads method for the new
                    intervals
                    intervals2b.append(len(M))
                    state = gen_state(M, curr_s)
                    break

            else: #update only the state of this interval if we have
                not found a working s value
                state[i] = (r, curr_s)

def gen_state(M, s):
    #generate state for each interval in M to perform parallel
        threads method
    #the state are the current r and s values, we keep them to be
        able to always only do one step of step 2c for a single
        interval and then proceed with the next interval

    state = []

    for i in range(len(M)):
        (a1, b1) = M[i]
        r = ceil(2 * (b1 * s - 2 * B), n)
        slow = ceil((2 * B + r * n), b1)
        state.append((r, slow))

    return state
```

```
def oracle(s, m):
    #oracle which returns whether m*s mod n is conforming (according
         to the different oracles / more or less restrictive tests)

    r = (m*s) % n
    in_interval = (2*B <= r) and (r < 3*B) #checks whether first
        byte is 0x00 and second byte is 0x02

    if not in_interval: #return False if first test fails (made by
        all oracles), make it faster for simulation (in practice not
         a good idea to immediately return False if this test fails)
        return False

    EB = int.to_bytes(r, k, 'big') #transform r into a bytes object,
         note: indexing in python starts with 0 but we start
        counting bytes for EB with index 1

    if check_eight_nonzero: #oracle checks that EB1 = 0x00, EB2 = 0
        x02 and EB3 - EB10 are nonzero

        eight_nonzero = all((x != 0) for x in EB[2:10])

        if not eight_nonzero: #if does not pass the check we return
            False
            return False

        if check_one_zero: #oracle also checks that at least one
            byte from EB11 to EBk is 0x00

            one_zero = any((x == 0) for x in EB[10:])
            return one_zero

        return True

    elif BVO: #Bad Version Oracle for SSL/TLS sessions
        #in addition to testing that EB1 = 0x00 and EB2 = 0x02 also
            tests that EB3 to EB(k-49) are nonzero, EB(k-48) = 0, EB
            (k-47) = major and EB(k-46) = minor

        #first check zero byte and if not zero directly return False
             (to make simulations faster)
        zero = EB[k - 49] == 0
        if not zero:
            return False

        #next check that there is some wrong version number and if
            not directly return False (again, to make simulations
            faster)
        wrong_vnumbers = EB[k - 48] != v_major or EB[k - 47] !=
            v_minor
        if not wrong_vnumbers:
            return False
        #lastly, check that EB3 to EB(k-49) are nonzero
        nonzeros = all((x != 0) for x in EB[2:(k-49)])

        return nonzeros

    return in_interval  #oracle that only checks that EB1 = 00 and
        EB2 = 02

def check_overlaps(newM, low, high):
    #check that the new interval [low, high] does not overlap with
        any prior interval, if yes we union them into one interval

    for i in range(len(newM)):

        (a1, b1) = newM[i]

        if (a1 <= high) and (b1 >= low):
            newa = min(a1, low)
            newb = max(b1, high)

            newM[i] = (newa, newb)

            return newM

    newM.append((low, high))

    return newM

def update_intervals(M, s):
    #update the interval(s) in M using the found s value according
        to Step 3

    newM = []
    for (a1, b1) in M:

        rlow = ceil((a1*s - 3*B + 1), n)
        rhigh = (b1*s - 2*B)//n + 1

        for r in range(rlow, rhigh):
            interval_low = max(a1, ceil((2*B+r*n), s))
            interval_high = min(b1, (3*B-1+r*n)//s)

            newM = check_overlaps(newM, interval_low, interval_high)

    return newM

def gen_and_test_trimmers(message):
    #generating working denominators t

    trims_den = [] #array containing all working denominators
    maxt = ceil(2*n,9*B) #maximum value of t until which we test
    count = 0

    #we test u in range t - t_slack - 1 to t + t_slack + 1
```

```
#t_slack specified for different oracle types and keylengths to
    have good performance
slack = t_slack + 1

for t in range(3, maxt):

    t_inv = pow(t, -1, n)

    umin = (2 * t) // 3 + 1 #smallest u value which can work for
         this t
    umax = ceil(3 * t, 2) - 1 #largest u value which can work
        for this t

    for s in range(1, slack+1):
        #test t-1, t+1, ..., t-slack, t+slack
        #as soon as one working u value is found we append t to
            the denominator array and directly continue with
            next t
        #stop and return found denominators t as soon as we have
             reached the allowed number of oracle calls to
            search for trimmer denominators (trimmer_values)

        if t-s < umin and t+s > umax: #if we are already below
            umin and above umax we know this t cannot work
            break

        if count >= trimmer_values: #if we have already tested
            trimmer_values many trimmers with the oracle we are
            not allowed to test any more trimmers and return the
             found denominators in trims_den
            return trims_den, count

        if t-s >= umin and math.gcd(t-s, t) == 1: #only test u =
             t-s if it is larger than umin and the gcd of u and
            t is 1
            count += 1

            if oracle(((t - s) * t_inv) % n, message): #if u = t
                -s works we know t is a valid denominator
                trims_den.append(t)
                break

        if count >= trimmer_values:
            return trims_den, count

        if t+s <= umax and math.gcd(t+s, t) == 1: #only test u =
             t+s if it is smaller than umax and the gcd of u and
             t is 1
            count += 1

            if oracle(((t + s) * t_inv) % n, message): #if u = t
                +s works we know t is a valid denominator
                trims_den.append(t)
                break
    return trims_den, count

def find_low_high(t, message):
    #t = lcm of all working trimming denominators found
    #want to find best possible trimming pairs (u_l, t) and (u_h, t)
         to trim the lower respectively upper bound on the message

    calls = 0

    t_inv = pow(t, -1, n)

    low = (2*t)//3 + 1 #lowest possible u value we have to test for
        finding u_l (a lower value cannot work)
    high = ceil(3*t, 2)-1 #highest possible u value we have to test
        for finding u_h (a higher value cannot work)

    #find u_l by performing binary search in [low, t], want u_l as
        small as possible
    #want u/t >= 1 to trim interval, so upper bound on u_l is t
    u_l, calls = bin_search_trimmers(low, t, t_inv, message, True,
        calls)

    # find u_h by performing binary search in [t, high], want u_h as
         large as possible
    #want u/t <= 1 to trim interval, so lower bound is t
    u_h, calls = bin_search_trimmers(t, high, t_inv, message, False,
         calls)

    return (u_l, t), (u_h, t), calls

def bin_search_trimmers(low, high, t_inv, message, findmin, calls):
    #perform binary search for the best trimmer pairs for the t in
        the interval [low, high]
    #t_inv = inverse of the lcm
    #findmin = True if we want to find the u_l and False if we want
        to find u_h

    if low >= high:
        if findmin: #we know that our high bound always works if we
            want to find the minimum u_l
            return high, calls
        else: #we know that our low bound always works if we want to
             find the maximum u_h
            return low, calls
    else:
        #if there are at least two values left we look at middle
            value mid and test whether the trimmer pair (mid, t)
            works
        if findmin:
            mid = (low + high)//2
        else:
            mid = ceil(low + high, 2)
```

```
calls += 1

if oracle((mid * t_inv) % n, message):
    #if trimmer pair (mid, t) works we can update the bounds
        , for u_l we can update the high bound (as we want
        to find the minimum) and for u_h the lower bound (as
         we want to maximize u_h)
    if findmin:
        return bin_search_trimmers(low, mid, t_inv, message,
             findmin, calls)
    else:
        return bin_search_trimmers(mid, high, t_inv, message
            , findmin, calls)

else:
    #if the trimmer pair (mid, t) does not work we could
        have a false negative (depending on the oracle type)
         and we do not want to procees with the wrong
        interval (and hence throw away a good trimmer)
    #for u_l we want to find the smallest working u value,
        hence if mid does not work (to make the false
        negative probability smaller) we try values mid - 1,
         ..., mid - bs_slack
    #for u_h we want to find the largest working u value,
        hence if mid does not work (to make the false
        negative probability smaller) we try values mid + 1,
         ..., mid + bs_slack
    #bs_slack is set differently for the different oracle
        types and keylengths to obtain the best performance
        (in terms of required oracle calls)

    if findmin:
        #if mid does not work we look at values mid-1, mid
            -2, ... mid-bs_slack

        for s in range(1, bs_slack+1):

            if mid - s >= low:
                #if the u value is above the lower bound we
                    test it
                calls += 1
                if oracle(((mid-s) * t_inv) % n, message): #
                    as soon as we have found a working value
                     we return the updated interval and
                    continue searching there
                    return bin_search_trimmers(low, mid - s,
                         t_inv, message, findmin, calls)
            else:
                #if we have tested all values until the
                    lower bound we know that all of them do
                    not work and proceed to search in the
                    upper interval
                    return bin_search_trimmers(mid + 1, high,
                        t_inv, message, findmin, calls)

            #if we have not found a working trimmer look in
                upper interval
            return bin_search_trimmers(mid + 1, high, t_inv,
                message, findmin, calls)

        else:
            #if mid does not work we look at values mid+1, mid
                +2, ... mid+bs_slack

            for s in range(1, bs_slack + 1):

                if mid + s <= high:
                    #if the u value is below the upper bound we
                        test it
                    calls+=1
                    if oracle(((mid + s) * t_inv) % n, message):
                         #we have found a working u value, so
                         update the interval and continue
                         searching there
                         return bin_search_trimmers(mid + s, high
                             , t_inv, message, findmin, calls)
                else:
                    #if we have tested all values below the
                        upper bound we proceed to search in the
                        lower interval
                    return bin_search_trimmers(low, mid-1, t_inv
                        , message, findmin, calls)

            #if none of the tested u values worked we continue
                searching in the lower interval
            return bin_search_trimmers(low, mid - 1, t_inv,
                message, findmin, calls)

def check_and_skip_s(s, j, low, high):
    #checking whether we know that we can skip the current s value
        and if so we skip all of the ones we know cannot work
        according to the Skipping Holes technique
    #j = current index, low =  next lower bound on s values we know
        we can skip, high = next upper bound on s values which can
        be skipped

    while True:
        if s >= low and s <= high: #if s lies in the next interval
            we can skip, we jump to the first value above this
            interval
            s = high + 1
        elif low > high:  #if the interval is empty we know that we
            cannot identify any more intervals to skip, so we return
             an index j of -1
            return s, -1, 0, 0
        elif s < low:
            return s, j, low, high #if s is below the lower bound we
                 know that we cannot skip and return the same s
                value, index j, low and high bound

        j += 1
        low = ceil(3 * B + j * n, a)
        high = ceil(2 * B + (j + 1) * n, b) - 1

def bleichenbacher_p(i, rn_key, rsa_keys):
    #parallel version of bleichenbachers algorithm which generates
        message (reproducible) and uses specified rsa key from
        pregenerated keyfile

    #try catch block because of heuristic (when aborting we return
        an exception which tells the processor to not append
        anything to the solution arrays but continue with next
        simulation)
    try:

        #GENERATE MESSAGE

        #initialize random number generator for this simulation of
            the experiment
        global aes_random
        aes_random = AES_Random(rn_key)

        #extract keys from input rsa_keys
        global n, e, d
        n, e, d = int(rsa_keys[0]), int(rsa_keys[1]), int(rsa_keys
            [2])

        #generate a message for a keylength of k bytes, message is a
             bytes object with msize bytes
        (message, msize) = gen_message(k)

        #pad the message according to the PKCS#1 v1.5 standard
        EB = pad_message(message, msize)

        #convert the message to an integer for processing in the
            decryption algorithm
        message = int.from_bytes(EB, 'big')

        #ciphertext = pow(message, e, n)  #encryption not necessary
            as we will take a shortcut in the oracle to improve
            performance

        #initialize oracle calls for the different parts (calls
            needed for step 2a, 2b and 2c)
        oracle_calls_a = 0
        oracle_calls_b = 0
        oracle_calls_c = 0

        #initialize how many rounds of 2b were needed, how many
            intervals there were before a round of 2b (if a round of
     2b had to be made)
rounds_b = 0
intervals2b = []

#initialize number of oracle calls required for finding and
    determining the best trimmers
oracle_calls_t = 0

#START ATTACK

global a, b
a = 2*B #lower bound on the message (that the attacker knows
    )
b = 3*B-1 #upper bound on the message (that the attacker
    knows)

low = (1, 1) #initialize trimmer pair to trim lower bound (u
    , t), want to maximize t/u
high = (1, 1) #initialize trimmer pair to trim upper bound (
    u, t), want to minimize t/u

#perform trimmer improvement
if trimmers:

    #generate valid trimmer denominators t (in array
        trims_den)
    trims_den, calls = gen_and_test_trimmers(message)
    oracle_calls_t += calls

    #if we have found at least one valid denominator we
        compute the lcm of all found denominators and search
         for the best u_l and u_h for this lcm
    if (len(trims_den) != 0):
        lcm = int(np.lcm.reduce(trims_den))
        low, high, calls = find_low_high(lcm, message)
        oracle_calls_t += calls
        a = (2*B*low[1])//low[0]  #trim the lower bound with
             the best found trimming pair (u_l, lcm)
        b = (3*B - 1)*high[1]//high[0] + 1 #trim the lower
            bound with the best found trimming pair (u_h,
            lcm)

#initialize M
M = [(a, b)]

#How much larger the initial interval was compared to the
    one obtained by trimming the interval
trim_value = (B-1)/(b-a+1)

#Step 1 only needed for signatures, our ciphertext is
    already PKCS conforming so we skip this step

#Step 2
```

```
    #Step 2.a
    s, calls = next_s(ceil(n, (3*B)), message, False, M)
    oracle_calls_a += calls

    M = update_intervals(M, s)

    #initialize variabes used for heuristic for step 2c
    skip_nr = 0 #initialize how many values in step 2c we will
        skip (which we would otherwise test with the oracle)
    calls_2c_1 = 0
    count_2c = False

    while True:
        # Step 2.b
        if len(M) > 1:
            intervals2b.append(len(M)) #save the number of
                intervals that were present before performing a
                round of step 2b

            #perform parallel threads on the intervals if we do
                not have more that pt_max_intervals intervals
            if do_parallel_threads and len(M) <=
                pt_max_intervals:
                oracle_calls_b, s, M, rounds_b, intervals2b =
                    parallel_threads(M, message, s, intervals2b)
                continue

            #count how many times we had to perform a round of 2
                b
            rounds_b += 1

            s, calls = next_s(s+1, message, True, M)
            oracle_calls_b += calls

        # Step 2.c
        elif len(M) == 1:

            (a1, b1) = M[0]

            #Step 4
            if a1 == b1: #if the interval only contains one
                number we are done and have found the desired
                message
                assert a1 == message
                return i, oracle_calls_a, oracle_calls_b,
                    oracle_calls_c, oracle_calls_t, rounds_b,
                    trim_value, intervals2b, low, high

            s, calls = next_s2(a1, b1, s, message, skip_nr)
            oracle_calls_c += calls

            if improve_2c:
                    if calls > 5*exp_tries and not count_2c: #we are
                         not already counting calls to see structure
                         and found a case where we have too many
                        calls , so we start "pattern recognition"
                        count_2c = True

                    elif count_2c and calls_2c_1 == 0: #we save the
                        value of the first call to 2c (if we are
                        counting calls)
                        calls_2c_1 = calls

                    elif count_2c and calls_2c_1 != 0:
                        #we set the nr to skip the previous skip
                            number + to the minimum observed value (
                            out of two) - the expected number of
                            calls to 2c
                        #we add to the skip number if we, for some
                            reason underestimated the number of
                            calls to skip in the first round
                        skip_nr += min(calls_2c_1 , calls) -
                            exp_tries
                        skip_nr = max(0, skip_nr)
                        count_2c = False
                        calls_2c_1 = 0

            #Step 3
            M = update_intervals(M, s) #after we have found a
                working s value we update all of the intervals

    except Exception as e:
        if not improve_2c:
            print(e)
        return Exception


def collect_result(result):
    #functions called whenever one simulation of the experiment is
        done, saves all the results for one simulation in the
        correct arrays
    #this function is called sequentially for each finished
        simulation , so we have the same order of the results in the
        different arrays
    #try and catch block for the heuristic as we will not append any
         results whenever we abort

        try:
            #extract the result from a simulation
            global hist , hista , histb , histc , hist_roundsb ,
                hist_intervals2b , hist_utmin , hist_utmax , histt ,
                histtrim_value , index
            (i , oracle_calls_a , oracle_calls_b , oracle_calls_c ,
                oracle_calls_t , rounds_b , trim_value , intervals2b ,
                utmin, utmax) = result
            #compute total number of oracle calls required for the
                simulation
            oracle_calls = oracle_calls_a + oracle_calls_b +
                oracle_calls_c + oracle_calls_t

            #append results to the arrays
            hist.append(oracle_calls)
            hista.append(oracle_calls_a)
            histb.append(oracle_calls_b)
            hist_roundsb.append(rounds_b)
            histc.append(oracle_calls_c)
            histt.append(oracle_calls_t)
            histtrim_value.append(trim_value)
            hist_intervals2b.append(intervals2b)
            hist_utmin.append(utmin)
            hist_utmax.append(utmax)
            index.append(i)


        except Exception:
            pass

def init_process(oracle, keylength_, SH, T, PT, I2C, T_N, P_M,
    t_slack_, bs_slack_):
    #initialize the global variables of the processes for one
        experiment (these will stay the same for all the simulations
         of one experiment)

    #initialize keylength, k, B, a and b
    global keylength, k, B, a, b
    keylength = keylength_
    k = keylength // 8
    B = 2 ** (8 * (k - 2))
    a = 2 * B #lower bound on message we want to find
    b = 3 * B - 1 #upper bound on message we want to find

    #initialize oracle. trimmer slack values and expected tries for
        step 2c (depending on oracle and keylength)
    global check_eight_nonzero, check_one_zero, BVO, t_slack,
        bs_slack, exp_tries, cutoff_2c
    if oracle == 'TTT':
        check_eight_nonzero = False
        check_one_zero = False
        BVO = False
        t_slack = t_slack_
        bs_slack = bs_slack_
        exp_tries = 2
        cutoff_2c = 1000
    elif oracle == 'TFT':
        check_eight_nonzero = True
        check_one_zero = False
        BVO = False
        t_slack = t_slack_
        bs_slack = bs_slack_
```

```
            exp_tries = 2
            cutoff_2c = 1000
    elif oracle == 'FFT':
        check_eight_nonzero = True
        check_one_zero = True
        BVO = False
        if keylength == 512:
            t_slack = t_slack_
            bs_slack = bs_slack_
            exp_tries = 11
            cutoff_2c = 5500
        elif keylength == 1024:
            t_slack = t_slack_
            bs_slack = bs_slack_
            exp_tries = 5.6
            cutoff_2c = 2800
        elif keylength == 2048:
            t_slack = t_slack_
            bs_slack = bs_slack_
            exp_tries = 3.34
            cutoff_2c = 1670
    elif oracle == 'BVO':
        check_eight_nonzero = False
        check_one_zero = False
        BVO = True
        t_slack = t_slack_
        bs_slack = bs_slack_
        if keylength == 512:
            exp_tries = 1.05*pow(2, 9)
            cutoff_2c = 1000*1.05*pow(2, 8)
        elif keylength == 1024:
            exp_tries = 1.37*pow(2, 9)
            cutoff_2c = 1000*1.37*pow(2, 8)
        elif keylength == 2048:
            exp_tries = 2.28*pow(2, 9)
            cutoff_2c = 1000*2.28*pow(2, 9)

            #initialize improvements and heuristics for this
                experiment
    global skipping_holes, trimmers, do_parallel_threads,
        trimmer_values, pt_max_intervals, improve_2c
    skipping_holes = SH
    trimmers = T
    do_parallel_threads = PT
    trimmer_values = T_N
    pt_max_intervals = P_M
    improve_2c = I2C

def bleichenbacher_simulations(R, E, oracle, keylength, N, path, SH,
    T, PT, I2C, T_N, P_M, t_slack_, bs_slack_):
    #R = Experiment Round, E = Experiment Number, keylength, N =
        number of simulations, path = where to save the generated
        data
```

```
#SH = Skipping Holes, T = Trimmers, PT = Parallel Threads, I2C =
     Improve Step 2c, T_N = number of trimmers, P_M = maximum
    number of intervals to start PT method
#t_slack_ = slack value used for generating and testing trimmers
    , bs_slack_ = slack value used for binary search for
    trimmers

processes = 50 #how many processes used to parallelize
    simulations

#initialize all the histograms to be empty when starting an
    experiment
global hist, hista, histb, histc, hist_roundsb, hist_intervals2b
    , hist_utmin, hist_utmax, histt, histtrim_value, index
hist = []
hista = []
histb = []
histc = []
hist_roundsb = []
hist_intervals2b = []
hist_utmin = []
hist_utmax = []
histt = []
histtrim_value = []
index = []

#read keys from generated keyfile
keys_path = '/home/lcapol/Daisen␣Code/Generated␣Keys/rsa␣keys␣-␣
    keylength␣%d.csv' %keylength
#keys_path = '/Users/Livia/Desktop/Daisen Code/Generated Keys/
    rsa keys - keylength %d.csv' % keylength

#read in the pregenerated keys with the correct keylength
#we use a file of 1000 pregenerated keys and will use them in a
    round-robin fashion for the simulations
keys = pd.read_csv(keys_path)
keys = keys.drop(keys.columns[0], axis=1).to_numpy()

'''
#step through one experiment (to inspect it)
i = 20119
rn_key = (R).to_bytes(4, 'big') + (E).to_bytes(4, 'big') + (i).
    to_bytes(4, 'big') + (0).to_bytes(4, 'big')
init_process(oracle, keylength, SH, T, PT, A2b, A1S2b, A1R2b,
    I2C, T_N, P_M, t_slack_, bs_slack_)
result = bleichenbacher_p(i, rn_key, keys[i % 1000])
'''

#parallelize using a pool of processes, initialize the global
    variables of the processes with the desired parameters (
    oracle type, keylength, etc.)
pool = mp.Pool(initializer=init_process, initargs=(oracle,
    keylength, SH, T, PT, I2C, T_N, P_M, t_slack_, bs_slack_),
    processes=processes)
```

```
#print start time of experiment
t = time.localtime()
current_time = time.strftime("%H:%M:%S", t)
print("Starting Experiment %d: %s" %(E, current_time))

#run N simulations of the experiment asynchronously on the
    specified number of processes and save the results for each
    run in the histograms
for i in range(N):
    #specify the key for the random number generator for this
        simulation of the experiment, this key is unique for
        each simulation and experiment such that we always
        generate different random values and messages
    rn_key = (R).to_bytes(4, 'big') + (E).to_bytes(4, 'big') + (
        i).to_bytes(4, 'big') + (0).to_bytes(4, 'big')
    #apply the bleichenbacher function using this random number
        generator key and rsa key
    pool.apply_async(bleichenbacher_p, args=[i, rn_key, keys[i %
         1000]], callback=collect_result)

#close the processor pool and wait for all of them to finish
pool.close()
pool.join()
# now all of the data is in the respective histograms

#calculate metadata to store in metadata file
success_prob = len(hist) / N
mean = np.average(hist)
median = np.median(hist)
mean_2a = np.average(hista)
median_2a = np.median(hista)
mean_2b = np.average(histb)
median_2b = np.median(histb)
mean_2c = np.average(histc)
median_2c = np.median(histc)
mean_rb = np.average(hist_roundsb)
median_rb = np.median(hist_roundsb)
mean_t = np.average(histt)
median_t = np.median(histt)
mean_trimf = np.average(histtrim_value)
median_trimf = np.median(histtrim_value)

#create array containing metadata and lables for the respective
    metadata
metadata = np.array([R, E, N, success_prob, mean, median,
    mean_2a, median_2a, mean_2b, median_2b, mean_2c, median_2c,
    mean_rb, median_rb, mean_t, median_t, mean_trimf,
    median_trimf])
labels = ['Experiment Round', 'Experiment Number', 'Number of
    Simulations', 'Success Probability', 'Mean Total', 'Median
    Total', 'Mean 2a', 'Median 2a', 'Mean 2b', 'Median 2b','Mean
     2c', 'Median 2c', 'Mean Rounds 2b', 'Median Rounds 2b', '
    Mean Trimmers', 'Median Trimmers', 'Mean Trimmer Interval
```

```
        Improvement', 'Median␣Trimmer␣Interval␣Improvement']

    #create metadata file and store metadata there
    results_path_md = os.path.join(path, 'metadata.csv')
    results_md = pd.DataFrame(metadata, index=labels)
    results_md.to_csv(results_path_md)

    #store data collected from all of the simulations for the
        experiment in a file
    results_path = os.path.join(path, 'experimental␣results.csv')
    results = pd.DataFrame({'i': index, 'Total␣Oracle␣Calls': hist,
        'Oracle␣Calls␣2a': hista, 'Oracle␣Calls␣2b': histb, 'Oracle␣
        Calls␣2c': histc, 'Number␣of␣intervals␣before␣call␣to␣2b':
        hist_intervals2b, 'Rounds␣of␣2b': hist_roundsb, 'Oracle␣
        Calls␣Trimmers': histt, 'Interval␣Improvement␣Trimmers':
        histtrim_value, '(umin,␣t)': hist_utmin, '(umax,␣t)':
        hist_utmax})
    results.to_csv(results_path)

    #print end time of experiment
    t = time.localtime()
    current_time = time.strftime("%H:%M:%S", t)
    print("Experiment␣%d␣Complete:␣%s" % (E, current_time))
```

A.2 Random Number Generator based on AES

```
from Cryptodome.Cipher import AES

class AES_Random:
    #random number generator based on AES with ECB mode

    count = (0).to_bytes(16, 'big') #current counter (to be
        encrypted using AES), is incremented after each requested
        block of random data
    key = (0).to_bytes(16, 'big') #key used for AES
    cipher = AES.new(key, AES.MODE_ECB) #AES module

    def __init__(self, key_):
        self.key = key_
        self.cipher = AES.new(self.key, AES.MODE_ECB)

    def ceil(self, a, b): #function for ceil(a/b)
        return (a // b) + (a % b > 0)

    def random_bytes(self, n_bytes):

        n_blocks = self.ceil(n_bytes, 16) #how many AES blocks (128
            bits) we will need to provide n_bytes random data
        res = b''

        #generate n_blocks number of blocks of 128 random bits using
            AES
        for i in range(n_blocks):

            res += self.cipher.encrypt(self.count)

            #increment counter after each generated datablock
            self.count = (int.from_bytes(self.count, 'big') + 1) %
                pow(2, 128)
            self.count = self.count.to_bytes(16, 'big')

        return res[:n_bytes] #return n_bytes random bytes

    def randint(self, n): #return a random integer 0 <= r <= n,
        assumes that 0 <= n < 256 (in our case n is at most 256 -
        11)
        r = n+1

        while r > n:
            rb = self.random_bytes(1) #only generate one random byte
                 as n is always < 256 in our case
            r = int.from_bytes(rb, 'big')

        return r

    def random_nonzero_bytes(self, n_bytes): #generate n_bytes
        nonzero bytes

        remaining = n_bytes
        res = b''

        while len(res) < n_bytes:
            r_bytes = self.random_bytes(remaining) #generate
                remaining number of bytes we need

            res += self.remove_zero_bytes(r_bytes) #filter out zero
                bytes
            remaining = n_bytes - len(res) #update how many
                remaining bytes we need

        return res

    def remove_zero_bytes(self, bytes): #filter out zero bytes
        res = b''

        for i in range(len(bytes)):
            if bytes[i] != 0:
                res += bytes[i:i+1]

        return res
```

Bibliography

[1] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS Using SSLv2. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 689–706. USENIX Association, 2016.

[2] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012, Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer, 2012.

[3] Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[4] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of Bleichenbacher's Oracle Threat (ROBOT). In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 817–832. USENIX Association, 2018.

[5] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption. In

Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 752–769. Springer, 2012.

[6] J. Mary Jones and Gareth A. Jones. *Elementary Number Theory*. Springer Undergraduate Mathematics Series. Springer, 1998.

[7] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, third edition, 2020.

[8] Vlastimil Klíma, Ondrej Pokorný, and Tomáš Rosa. Attacking RSA-Based Sessions in SSL/TLS. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2003.

[9] James Manger. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer, 2001.

[10] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schnizel, and Erik Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 733–748. USENIX Association, 2014.

[11] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[12] Kenneth G. Paterson and Thyla van der Merwe. Reactive and Proactive Standardisation of TLS. In Lidong Chen, David A. McGrew, and Chris J. Mitchell, editors, *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*, volume 10074 of *Lecture Notes in Computer Science*, pages 160–186. Springer, 2016.

[13] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[14] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[15] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[16] Eyal Ronan, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 435–452. IEEE, 2019.

[17] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 990–1003. ACM, 2014.