BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DETECTION OF TIMING SIDE-CHANNELS IN TLS

DETEKCE ČASOVÝCH POSTRANNÍCH KANÁLŮ V TLS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. JAN KOSCIELNIAK

SUPERVISOR
VEDOUCÍ PRÁCE

prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

23189

Student:	**Koscielniak Jan, Bc.**
Programme:	Information Technology Field of study: Information Technology Security
Title:	**Detection of Timing Side-Channels in TLS**
Category:	Software analysis and testing

Assignment:

1. Get familiar with the SSL/TLS protocol, with the tlsfuzzer tool, and its use for testing the protocol.
2. Study side-channel vulnerabilities, caused by timing leaks, and existing attacks on SSL/TLS based on them.
3. Design and implement an automated framework (as a part of tlsfuzzer) for testing presence of timing leaks in arbitraty TLS implementations by using timing of network responses.
4. Design and implement test cases for tlsfuzzer for selected existing attacks on TLS using the framework.
5. Verify the tests on known vulnerable versions of popular TLS libraries as well as on their fixes.
6. Evaluate the obtained results, especially the framework's abilitity to detect a timing side-channel, discuss possibilities of further development.

Recommended literature:

- Ristić, I.: Bulletproof SSL and TLS. Feisty Duck, 2014.
- Böck, H., Somorovsky, J., Young, C.: Return Of Bleichenbacher's Oracle Threat (ROBOT), In Proc: 27th USENIX Security Symposium (USENIX Security 18), USENIX Association 2018.
- The tlsfuzzer homepage. URL: https://github.com/tomato42/tlsfuzzer

Requirements for the semestral defence:

- Items 1 and 2 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Consultant:	Kario Hubert, Mgr. Ing., RedHatCZ
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	May 20, 2020
Approval date:	October 31, 2019

Abstract

The TLS protocol is complex and widely used, which necessitates thorough testing, as there are many devices relying on it for secure communication. This thesis focuses on timing side-channel vulnerabilities, which seem to come back every few years in different variations of the same attacks. It aims to help correctly fix those vulnerabilities and prevent the creation of new ones by providing an automated side-channel testing framework that is integrated into the tlsfuzzer tool, and by expanding its test suite with test scripts for known attacks that exploit timing side-channels. The extension utilizes tcpdump for collecting the timing information and statistical tests and supporting plots to make a decision. The extension and the new test scripts were evaluated and shown to be accurate at spotting side-channels. Both the extension and the test scripts are now a part of the tlsfuzzer tool.

Abstrakt

Protokol TLS je komplexní a jeho použití je široce rozšířené. Mnoho zařízení používá TLS na ustanovení bezpečné komunikace, vzniká tak potřeba tento protokol důkladně testovat. Tato diplomová práce se zaměřuje na útoky přes časové postranní kanály, které se znovu a znovu objevují jako variace na už známé útoky. Práce si klade za cíl usnadnit korektní odstranění těchto postranních kanálů a předcházet vzniku nových vytvořením automatizovaného frameworku, který pak bude integrován do nástroje tlsfuzzer, a vytvořením testovacích scénářů pro známé útoky postranními kanály. Vytvořené rozšíření využívá program tcpdump pro sběr časových údajů a statistické testy spolu s podpůrnými grafy k rozhodnutí, zda se jedná o možný postranní kanál. Rozšíření bylo zhodnoceno pomocí nových testovacích skriptů a byla předvedena jeho dobrá schopnost rozlišit postranní kanál. Rozšíření spolu s testy je nyní součástí nástroje tlsfuzzer.

Keywords

TLS, fuzzing, tlsfuzzer, side-channels, timing side-channel, testing, Lucky 13, Bleichenbacher's attack

Klíčová slova

TLS, fuzzing, tlsfuzzer, postranní kanály, časový postranní kanál, testování, Lucky 13, Bleichenbacherův útok

Reference

KOSCIELNIAK, Jan. *Detection of Timing Side-Channels in TLS.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Za poslední dvě dekády došlo k masovému rozšíření sítě internet, která se stala součástí každodenního života mnoha lidí a je téměř nepostradatelnou pro fungování moderní společnosti. V porovnání se začátky internetu jsou na něj dnes kladeny daleko vyšší požadavky na bezpečnost, ať už při návštěvě běžných stránek, nebo při provádění transakcí v internetovém bankovnictví.

Technologií, která současnou úroveň zabezpečení napříč internetem umožňuje, je protokol dříve známý jako SSL, dnes už jako TLS. Kromě jiných využití funguje jako standardní forma zabezpečené komunikace po internetu, jako součást protokolu HTTPS. Právě z důvodu tak širokého rozšíření se jedná o kriticky důležitou součást internetu, neboť každá další nalezená chyba v jeho návrhu nebo implementaci může dopadnout na velké množství potenciálních cílů. Proto je nutné těmto chybám co možná nejvíce předcházet. Jedním ze způsobů, jak toho lze dosáhnout, je důkladné testování implementací TLS protokolu. Testování může probíhat na úrovni jednotlivých funkcí ve zdrojovém kódu knihovny, na úrovni modulů, nebo lze knihovnu testovat jako komplexní celek. V případě protokolu TLS však tento přístup nemusí být dostačující, z důvodu vysoké komplexity protokolu a množství možných konfigurací kryptografických parametrů.

Jedním z méně obvyklých přístupů je tzv. *Fuzz testování*, které je zaměřeno na hledání chyb v implementaci pomocí vstupů, které testovaný systém neočekává, nebo je považuje za chybné. Typickou vlastností *fuzzeru* (nástroje provádějícího fuzz testování) je vysoká míra automatizace a využití náhodného generování vstupních dat.

Tato diplomová práce se zabývá rozšířením nástroje *tlsfuzzer* – fuzzeru testujícího protokol TLS. Cílem práce je umožnit testování méně obvyklých útoků vedených pomocí časových postranních kanálů. Postranní kanál je vlastnost implementace kryptografického systému, přes kterou z něj nechtěně unikají informace o jeho vnitřním stavu. Jedná se tedy o vlastnost jeho implementace, čímž se odlišuje od klasické kryptoanalýzy, která se zaměřuje na nedostatky v návrhu kryptografického systému. Existuje více druhů postranních kanálů, ale relevantní pro tuto práci je ten časový, který se může projevit např. různou dobou odpovědi serveru na vstupy, které by měly mít tuto dobu shodnou. Ačkoliv se může zdát, že se jedná o zanedbatelné nedostatky, které nejsou zneužitelné pro skutečný útok, opak je pravdou. Při dostatečném počtu nasbíraných vzorků a následné statistické analýze je možné např. získat šifrovací klíč, kterým je komunikace zabezpečena, čímž může být odhalena potenciálnímu útočníkovi. Obrana proti tomuto typu útoků je obtížná a často není implementována korektně, i z toho důvodu, že je obtížné chyby tohoto typu testovat.

Cílem této práce je testování časových postranních kanálů v případě protokolu TLS usnadnit rozšířením nástroje *tlsfuzzer*. V rámci této práce byl nejdříve studován protokol TLS. Dále práce zahrnuje studium útoků pomocí postranních kanálů, zvláště pak těch časových, a věnuje se také technikám fuzz testování.

Samotný předmět této diplomové práce, rozšíření pro nástroj *tlsfuzzer*, bylo nejdříve modelováno na prototypu, na kterém byly zkoušeny možné směry řešení. Jednalo se o jednoduchou aplikaci, skládající se z klienta a serveru, mezi nimiž probíhala komunikace pomocí jednoduchého protokolu. Z vývoje tohoto prototypu se vyprofiloval nástroj *tcpdump* jako nejvhodnější pro zachytávání komunikace s vysokou přesností časových údajů o jednotlivých zprávách. Také byly vybrány dva statistické testy (Kolmogorov-Smirnov test a Box test), které vyhovovaly záměru rozpoznat rozdíly v sesbíraných skupinách vzorků a určit, zda se jedná o potenciální postranní kanál. Dále byly identifikovány podpůrné grafy, které mohou asistovat při tomto rozhodování.

Následně byl vybraný způsob řešení implementován v podobě rozšíření, které bylo začleněno do nástroje *tlsfuzzer*. Součástí cílů této práce bylo též poskytnout testovací skripty pro známé útoky časovými postranními kanály, což bylo splněno v podobě vytvoření skriptů pro útok známý jako Lucky 13 a pro zranitelnosti vycházející z Bleichenbacherova útoku. Testovací skripty a samotné rozšíření pak bylo validováno na třech nejpoužívanějších implementacích protokolu TLS – OpenSSL, GnuTLS a NSS.

Výsledky ukazují na dobrou schopnost navrženého rozšíření nástroje `tlsfuzzer` rozlišit potenciální postranní kanál. Navržené rozšíření má navíc přijatelnou míru falešných pozitiv. Rozšíření je spolu s testy v době odevzdání součástí nástroje `tlsfuzzer`.

Detection of Timing Side-Channels in TLS

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by Mgr. Ing. Hubert Kario. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jan Koscielniak
June 10, 2020

Acknowledgements

I would like to express my gratitude towards my academic supervisor prof. Ing. Tomáš Vojnar, Ph.D. for providing helpful insights regarding the thesis, and towards Mgr. Ing. Hubert Kario for technical consultation and advice.
Last but not least I would like to thank my family, my girlfriend and my friends for their continued support that got me this far.

Contents

Chapter 1

Introduction

In the last two decades the internet has experienced a quick rise in popularity, became a daily part of the lives of many people and is now a technology it is almost impossible to imagine a modern world without. In the early days of the internet, security was almost a secondary concern, but with the widespread adoption quickly became an issue.

One of the basic needs of an internet user is to be able to communicate privately without an adversary listening in on the conversation and being able to see any and all data transmitted over the network. The adversary also should not be able to compromise the integrity of the data, perhaps changing something before the communication reaches the recipient. Equally important is for the communicating parties to be able to verify each other's identity, preventing the adversary from impersonating any of the parties. These goals can be summarized as confidentiality, integrity and authentication.

This is where the Secure Sockets Layer (SSL) protocol was introduced in order to fulfil these goals. Later evolving into the Transport Layer Security (TLS) protocol, it is nowadays *the* mainstream way to establish a secure channel between two parties, most notably on the internet as HTTPS – also known as HTTP over TLS. It is widely relied on, which magnifies the effect of any vulnerability as it affects a large part of the websites on the internet. Because of such widespread adoption, it is critical that this software is designed and maintained with security as the primary factor in every step of the development process. Setting aside the design, a significant part of keeping a library implementing the TLS protocol secure is thorough and rigorous testing. Testing can be done on multiple levels – separate functions in the library's source code, individual modules of the library, or a whole library can be tested as one complex system. However, TLS has an additional property of being a highly complex network protocol, which calls for perhaps a less common approach to testing. An example of this can be a great number of configurations of cryptographic properties of the secure connection being established.

One of the possible approaches to covering such a complex configuration of an already complex protocol with tests is Fuzz testing, also known as fuzzing. Fuzzing aims to discover vulnerabilities by providing the system under test with invalid or unexpected data in order to cause a failure. A typical property of a fuzzer (a tool that performs fuzz testing) is a high degree of automation that usually involves a certain degree of randomization in generating the invalid data. Tlsfuzzer is such fuzzer, specialized for the TLS protocol.

This thesis is concerned with extending the tlsfuzzer's capabilities to allow testing for a specific type of vulnerability called timing side-channel. Side-channel describes a situation when a system involuntarily leaks additional information about its internal state, in addition to the usual output. Timing side-channel is a case where this additional information is leaked through time, e.g. the latency of server response to different inputs. While such a vulnerability can seem irrelevant and not exploitable in a real-world scenario, it has been proven again and again that this is not the case and with enough samples and statistical analysis, this information can be used to e.g. recover session keys, effectively revealing the communication to a possible adversary. Attacks utilizing these vulnerabilities are notoriously hard to fix in a correct way, which is even further obstructed by the fact that it is also hard to test for them.

This thesis aims to change that in the context of TLS and tlsfuzzer, by providing it with tools necessary for testing for timing side-channels and extending its test suite with test scripts for some of the known attacks utilizing timing side-channels.

After researching relevant topics such as the TLS protocol and side-channel attacks, a prototype was implemented to research various possible solutions to problems involved. The prototype served as a basis for the actual implementation into tlsfuzzer. The framework was extended to allow for timing side-channel analysis and the test suite expanded to cover two known attacks exploiting timing side-channels. The test scripts were then evaluated against popular TLS libraries with results indicating that extension is accurate in detecting side-channels, with an acceptable rate of false positives. The extension, along with tests, became a part of the `tlsfuzzer` framework.

The text first introduces the TLS protocol in Chapter 2. Chapter 3 covers an overview of different kinds of side-channel attacks with emphasis on timing side-channels. Chapter 4 describes the basics of fuzz testing. Next, the process of finding the right tools for timing side-channel discovery with results is described in Chapter 5. Implementation is then covered with details on the resulting extension architecture and added tests in Chapter 6. Finally, the extension is evaluated and results are discussed in Chapter 7. The thesis then concludes with summary and discussion on possible further development.

Chapter 2

SSL/TLS Protocol

The purpose of the SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols is to create a secure channel for communication between two peers, with the only requirement being that the communication is carried over reliable, in-order data stream [22]. SSL is a now-deprecated predecessor of TLS, and its use is considered insecure. Even though the version 1.0 of TLS was based on SSL 3, the name was changed due to a dispute between Netscape and Microsoft.

2.1 Description of the TLS Protocol

Information in this section is taken from the official TLS specification [12], unless specified otherwise. The protocol intends to provide the following properties to the secure channel it establishes between the communicating parties: [22]

- Authentication – Server always authenticates itself to the client, clients optionally can authenticate themselves to the server.
- Confidentiality – After the secure channel is established, the data transmitted over it shall be readable only to its endpoints.
- Integrity – Data transmitted over the secure channel shall not be modified by an adversary without detection.

Although establishing a secure channel is the main goal of TLS, it defines three more:

- Interoperability – The protocol should be implementation independent, meaning that two applications using different implementations of TLS should be able to communicate without knowledge of the other application's TLS implementation.
- Extensibility – TLS intends to provide a framework that is able to incorporate new encryption methods as necessary without the need to create a new protocol.
- Relative efficiency – TLS aims to perform as few cryptographic operations as possible, as they are highly CPU intensive, and to reduce network activity as well.

The protocol itself consists of several subprotocols that will be described here in detail, in their respective subchapters. Version 1.2 will be used to describe the subprotocols, even though version 1.3 exists, for two reasons. Firstly, as of now, there are still all TLS versions in use, with a plan to deprecate versions 1.0 and 1.1 in March 2020 [7], so TLS 1.2 will become the lowest supported version. Secondly, most of the attacks described in this thesis were devised against version 1.2, or lower. Differences between versions 1.2 and 1.3 will also be described in a separate section.

2.1.1 Record Protocol

The record protocol is used to describe the general structure of a TLS message. A high-level overview will be provided in this section. As seen in figure 2.1, TLS record consists of a Header and a Data section:

- Type – Identifies the subprotocol type of messages in the Data section with types relevant for this thesis enumerated below:
  - Change Cipher Spec protocol – Used for switching to encrypted communication.
  - Alert protocol – Used for error reporting.
  - Handshake protocol – Used for negotiation of the connection parameters.
  - Application Data protocol – Used for data transmission.
- Version – Specifies the TLS version using a major and a minor number.
- Length – Specifies the length of the Data section.

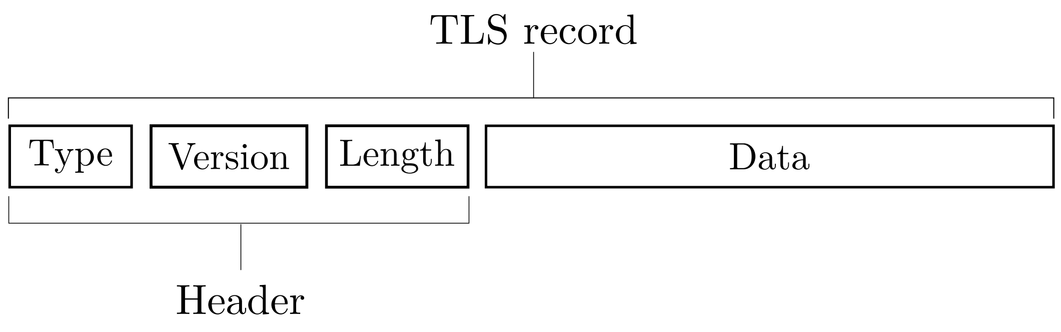

Figure 2.1: Schematic depiction of TLS record structure

The record protocol allows for fragmentation of the subprotocol messages, as the Data section in figure 2.1 is actually a TLS fragment that has the maximum size of 2^{14} bytes. This means that longer subprotocol messages must be split into multiple TLS records, resulting in fragmentation. At the same time, multiple subprotocol messages can be sent in the same TLS record, provided they are of the same type.

2.1.2 Handshake Protocol

The Handshake protocol in TLS is used for negotiating parameters of the secure channel and establishing it between two communicating parties. Its purpose is for the peers to agree on a TLS protocol version, cryptographic algorithms to use, optionally authenticate each other and generate shared secrets using public-key cryptography (or pre-shared keys).

First, the structure of a handshake message is visualized in table 2.1. This structure is encapsulated in Data section of a TLS record message. Next, the handshake will be explained in all its variants.

Table 2.1: Handshake record structure (Taken from [1])

Byte/+	Byte +0	Byte +1	Byte +2	Byte +3
0..4	Message type	Handshake message data length		
5..(length-1)	Handshake message data			

Full handshake

Every new TLS connection begins with a full handshake process. As previously mentioned, its purpose is for the two parties to negotiate parameters of a secure connection and then establish it. In Figure 2.2, the Client-Server communication is shown, annotated with message types from the handshaking protocol. The conversation is dissected message by message below.

`ClientHello` This message is used by the Client to initiate the handshake, or optionally to renegotiate the parameters if connection already exists. The message contains the following information:

- TLS version – highest version supported by the Client
- Random structure – contains 28 random bytes and a timestamp
- Session ID – optional, used in case the Client wishes to resume an already negotiated session
- List of supported cipher suites
- List of supported compression methods
- List of optional extensions the Client would like to use

`ServerHello` *ServerHello* is the Server's response to *ClientHello*, where the Server chooses from offered options and generates a session ID. The message contains the following fields:

- TLS version – highest version supported by the Server
- Random structure – contains 28 random bytes and a timestamp, must be generated independently from the Client's random structure
- Session ID – generated ID for this connection
- Chosen cipher suite

- Chosen compression method
- List of supported extensions – only out of those that the Client requested

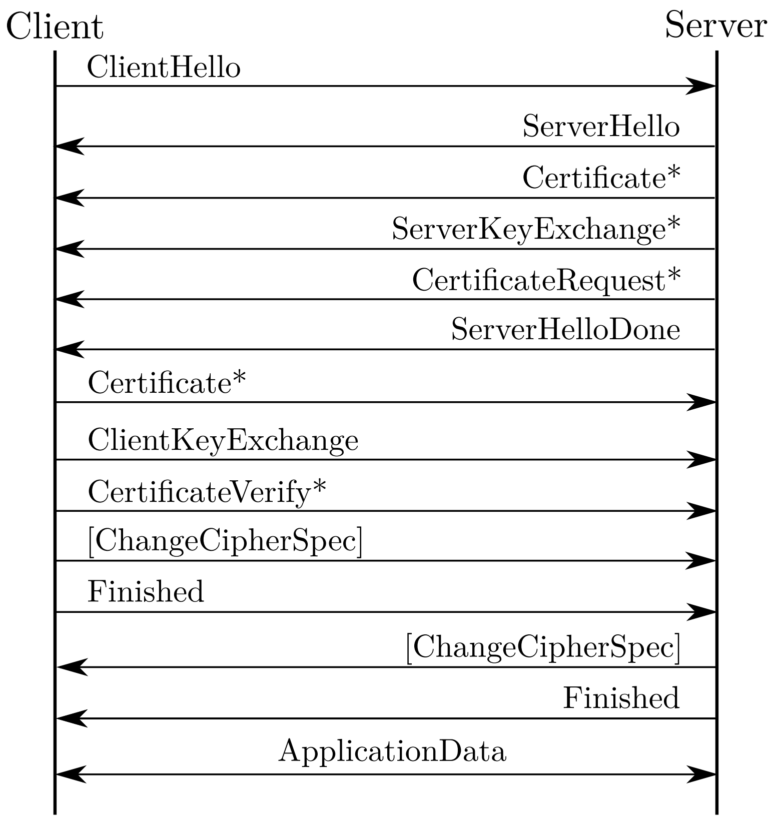

Figure 2.2: Communication scheme between the Client and the Server during full TLS handshake (Taken from [12]). Optional messages are marked with `*`. Messages that are not a part of the handshake protocol are marked with brackets.

`Certificate* (Server)` This message immediately follows *ServerHello* in case the chosen cipher suite's signature algorithm requires certificates to work (which is all of them except *anonymous* and those based on pre-shared keys). The Server selects the appropriate certificate based on the signature algorithm and appends the certificate chain to it, where each certificate directly validates the one before it.

`ServerKeyExchange*` This message is sent in case the *Certificate* message does not provide enough information for the Client to generate a pre-master secret (key exchange algorithms for which this is true are defined in the RFC [12]).

`CertificateRequest*` Under the condition that it authenticates itself (sending *Certificate*), the Server can request the Client to do the same.

`ServerHelloDone` With this message, the Server lets the Client know, that it is done sending *ServerHello* and related messages, and is now listening for the Client's response.

`Certificate (Client)*` If the Client received a *CertificateRequest* message, this will be its first response to the Server. As before, a Client certificate along with its certificate chain is contained in this message. The Client can choose to send this message without any contents, then it is in the Server's discretion to either continue the handshake, or respond with an error.

`ClientKeyExchange` This is a required message by the Client that sets the pre-master secret, either by sending it encrypted or giving the Server enough information to generate the same pre-master secret as the Client did.

`CertificateVerify*` This message is used for explicit verification of the Client's certificate. It contains a signature over all of the handshake messages sent or received so far, signed with certificate's private key for the Server to verify.

`ChangeCipherSpec (Client)` This message actually is not a part of the handshake protocol, but rather the only specified message in *Change Cipher Spec* subprotocol that indicates the party is now considering the connection authenticated and is switching to encryption for future messages.

`Finished (Client)` Finally, this message indicates that the handshake is considered complete from the Client's side. It also serves as a sanity check for the connection. That is achieved by taking the master secret, hash of all sent or received messages and a finished label (indicating either the Client or the Server) as inputs to Pseudo-random function (PRF), which generates at least 12 bytes (or more, depending on the nature of PRF in the selected cipher suite). Note that this message is already encrypted, as it is sent after *ChangeCipherSpec*. The Server will perform the sanity check by decrypting the contents of the message and comparing them with expected contents.

`ChangeCipherSpec & Finished (Server)` The Server performs the same actions as the Client after receiving the Server's *Finished* message. After the Client receives the Server's *Finished* message and verifies its contents, the handshake is complete and both parties can begin sending *Application Data* protocol messages, as described in section 2.1.4.

Session resumption

Because the full handshake is CPU-intensive, as a number of cryptographic operations must be executed in order to establish a connection, the protocol allows caching of session IDs in an effort to save both time and computing resources. Session resumption is performed via abbreviated handshake. As shown in Figure 2.3, the exchange begins again with the *ClientHello* message, except this time the Client is reusing a previously issued session ID. The Server then searches its cache for a match. If it is found and the Server is willing to resume the connection, it will respond with a *ServerHello* message with that particular session ID and follow that up with a *ChangeCipherSpec* and *Finished* messages. The Client will respond with the same set of messages, as it would in a full handshake. If the

Server cannot or wishes not to resume the session, it generates a new session ID and the conversation continues as a full handshake.

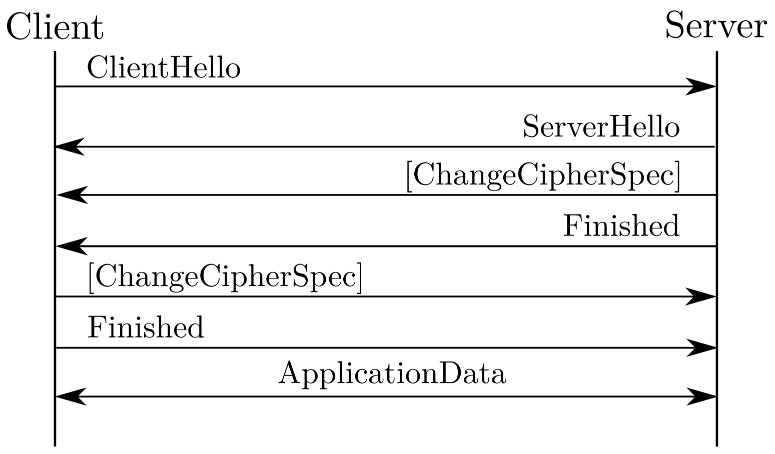

Figure 2.3: Communication scheme between the Client and the Server during the abbreviated TLS handshake (Taken from [12]).

2.1.3 Alert Protocol

Alert protocol is used to notify the other party about events that occurred during the connection, which are (with one exception) errors. The message structure is shown in table 2.2.

Table 2.2: Alert record structure (Taken from [1])

Byte/+	Byte +0	Byte +1	Byte +2	Byte +3
0..4	Level	Description		

Each message contains *Level*, which conveys the severity of the alert and can be either `warning` or `fatal`. Messages with `fatal` severity must result in immediate termination of the connection and invalidation of the session ID. The message also contains *Description*, which is further description of the alert, as defined in TLS specification.

Closure alert

There is a special alert message for when either one of the parties in the connection decides to terminate it. Any party can simply send `close_notify` alert, and the other side will respond with canceling any pending messages to be sent and sending a `close_notify` back. The initiating party is not required to wait for this message to terminate the connection on its side. Closure messages are included to avoid truncation attacks. An example of a truncation attack would be an adversary intercepting logout request from a user, leading the user to believe that they have been logged off, while in fact, the session is still alive [23].

2.1.4 Application Data Protocol

Last of the TLS subprotocols is the Application Data protocol. Its purpose is to simply carry whatever data either of the sides wants to send. An example of this can be the HTTPS protocol, which is the HTTP protocol carried over TLS.

2.2 TLS 1.3 Differences

Even though most attacks concerning this thesis are aimed at TLS 1.2, it is important to also describe the latest TLS version 1.3, as the goal of this thesis is to provide a framework, that will enable timing side-channel testing for any attacks that might appear in the future.

TLS 1.3 was released in August 2018, ten years after its predecessor TLS 1.2. Perhaps the most important changes were made to the handshaking protocol, aiming at lower latency and more security. Other changes include removing no longer secure cryptographic algorithms and fixing flaws in design that were exposed by various attacks on TLS. These changes are described in detail below. Information in this chapter is taken from [22] unless stated otherwise.

2.2.1 Changes to `ClientHello`

Innovations to the *ClientHello* are numerous and aim to correct some of the mistakes of the previous versions, like the support for compression, which was found to leak information about data being compressed, and as a result is no longer allowed. The more significant changes are described below.

Cipher suites

One of the major changes is the split of the supported cipher suites field into separate fields:

- Cipher and HKDF Hash
- Key exchange
- Signature algorithm

This change was made because in previous versions there had to be an assigned value for every valid combination of these three fields, which resulted in a large number of new values when just one cipher was added. To preserve backwards compatibility, the Cipher field stayed in the place `CipherSuite` field was, and the rest of the fields were moved into mandatory TLS extensions, located at the end of a TLS message. This was done to preserve backwards compatibility regarding the format of the message.

Many ciphers were also removed as they were deemed insecure. The ciphers that are left all fit into the Authenticated Encryption with Associated Data (AEAD) category of algorithms, that in addition to providing confidentiality for encrypted plaintext, also add means to check its integrity and authenticity, as well as of some unecrypted associated data.

A significant change was also made to the supported key exchange methods. The RSA key exchange is no longer supported in TLS 1.3, because it was exploited again and again, most recently in 2017 using the ROBOT attack [6]. The other reason is the lack of forward secrecy property in most exchanges, as it is not usual to generate new key pairs for each session. That is why the only option in TLS 1.3 available is the Diffie-Hellman key exchange that uses ephemeral keys – keys that are randomly generated for each new connection, so the forward secrecy property can be maintained.

Session IDs

The session resumption mechanism was also reworked and the `SessionID` field is now kept only for backwards compatibility. Session resumption is now performed using Pre-Shared keys (PSKs), that are established upon request during an existing connection, and can be later used to perform 0-RTT handshake (see respective Section in 2.2.2 for details).

Version negotiation

As of TLS 1.3, the `Version` field loses its purpose as a way of negotiating the version of TLS protocol to be used in the connection. This change was implemented as a reaction to common incorrect implementation of the protocol, as many servers rejected otherwise acceptable *ClientHello* because the `version` was higher than the server's highest supported version, instead of responding with that version in *ServerHello*. TLS 1.3 therefore uses the same `version` value as TLS 1.2 to preserve backwards compatibility and the actual negotiation was moved to `supported_versions` extension, in which client lists supported versions and the server (TLS 1.3 compliant) must choose from those and ignore the `version` field.

2.2.2 Changes to the Handshaking Protocol

The most significant changes were made to the TLS handshake. The full version of the handshake was shortened from two round trips to just one. The session resumption with the abbreviated handshake was also innovated and 0-RTT handshake was introduced. All of them are described in detail below.

Full handshake

The main motivation for changing the way how the handshake works were speed and security. Additional security is provided by encrypting a larger part of the handshake, to prevent downgrade attacks, such as FREAK[1] or LogJam [3]. Downgrade attacks exploit this vulnerability by tricking the server into using a weak cipher suite or older TLS version. In TLS 1.3, this is no longer possible, because the server uses *CertificateVerify* message to sign the contents of the handshake so far with the private key that is corresponding to the public key sent in the *Certificate* message. This ensures the integrity of the unencrypted as well as the encrypted part of the handshake.

The speedup is achieved by the client taking a guess on which key share the server is likely to select, and sending that in the *ClientHello* message using the `key_share` extension for Diffie-Hellman based key share, or the `pre_shared_key` extension in case of session resumption. Since the Diffie-Hellman key exchange is restricted to just a small set of options, the guess is likely to be right. In case the client guessed wrong, the server will ask for another option explicitly with *HelloRetryRequest* message.

[1] https://www.mitls.org/pages/attacks/SMACK#freak

Figure 2.4 illustrates the entire process of the full handshake with special attention to which data are encrypted. New messages and notable extensions are described below.

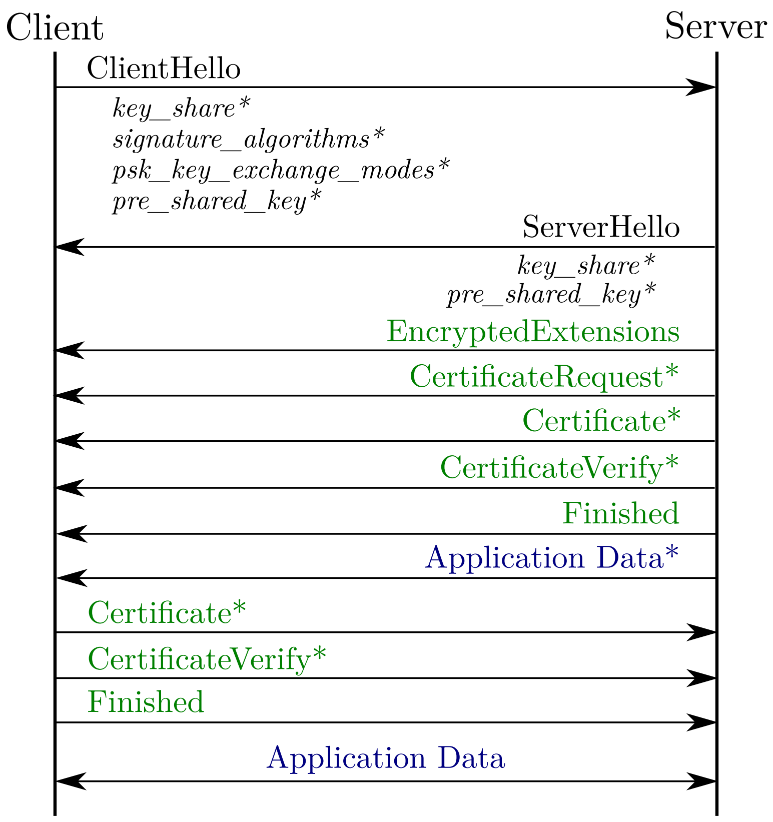

Figure 2.4: Communication scheme between the Client and the Server during the full TLS 1.3 handshake (Taken from [22]). Optional messages are marked with *. Green (handshake traffic keys) and blue (application traffic keys) colors indicate encryption. Cursive indicates notable extensions.

`key_share` extension This extension contains cryptographic parameters needed for the other party to complete the key exchange. Client can send as many key share groups as they like, provided there is only one set of key exchange parameters for each group, and that these parameters are generated independently.

`psk_key_exchange_modes` extension This extension is a requirement for the `pre_shared_key` and it indicates pre-shared key exchange modes client is supporting. This is important later during setup for the session resumption.

`pre_shared_key` extension This extension is used for establishing the connection based on pre-shared keys. This contains the identities client is offering for the server to choose from. This type of key exchange is supported because it is needed for session resumption in TLS 1.3.

`EncryptedExtensions` This is the first encrypted message server sends after *ServerHello*. It contains any extensions that are not needed for the key exchange itself but should be hidden from any potential eavesdroppers.

`CertificateVerify` As mentioned before, this message allows the sending party to prove that they possess the private key to the public key sent in the *Certificate* message. This applies to both server and client (in case of client authentication). It also allows the receiving party to check the integrity of the handshake transcript and that it has not been tampered with. In TLS 1.2, this used to be done for some key exchanges using the *ServerKeyExchange* message, where only signed content from the previous communication was client's random and server's random. This allowed for chosen prefix attack, where attackers could select the client's random.

Handshake with session resumption

This type of handshake is the equivalent of the abbreviated handshake from TLS 1.2. The main difference is that the resumption is not done by `sessionID` field in *ClientHello*, but rather a Pre-Shared Key (PSK). This key can be established out of band but is more commonly obtained during a previous TLS connection, when server, after completing the handshake, may send a *NewSessionTicket* message that creates an association between the ticket value and a secret PSK, which is sent in the message to the client. This enables the client to use the abbreviated handshake flow for the session resumption as illustrated by Figure 2.5.

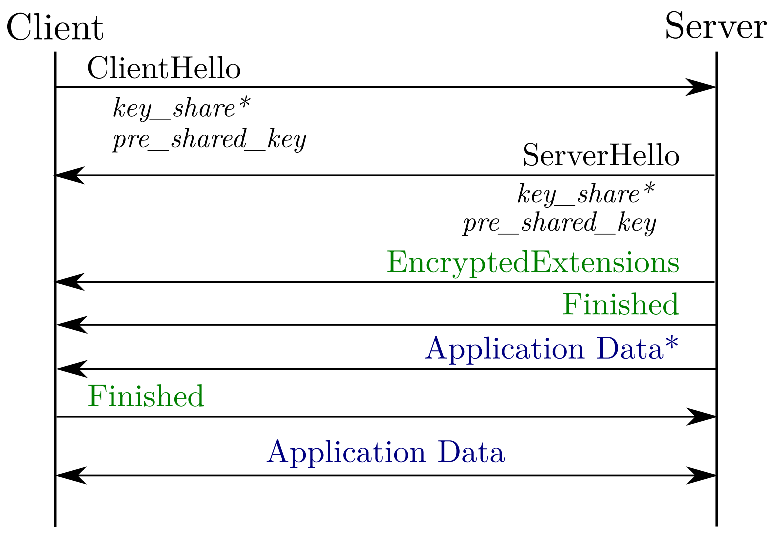

Figure 2.5: Communication scheme between the Client and the Server during the session resumption TLS 1.3 handshake (Taken from [22]). Optional messages are marked with *. Green (handshake traffic keys) and blue (application traffic keys) colors indicate encryption. Cursive indicates notable extensions.

0-RTT handshake

TLS 1.3 enables the client to go one step further and send the application data encrypted with the *ClientHello* message using PSK, hence the name 0-RTT handshake. This provides notable speedup and brings secure connection on par with an unencrypted one (e.g. HTTPS vs. HTTP). This is achieved by utilizing the `early_data` extension that has the same parameters as the PSK being used. Server can either ignore this early data, and continue with a regular session resumption handshake, or reply with its own early data, that is attached as an extension to the *EncryptedExtensions* message. Meanwhile, the client is permitted to send early Application Data until it receives server's *Finished* message. After that, the client must send the *EndOfEarlyData* message indicating that the client is done sending early data and is now switching from early traffic keys to the application traffic keys for the application data encryption. This process is illustrated in Figure 2.6.

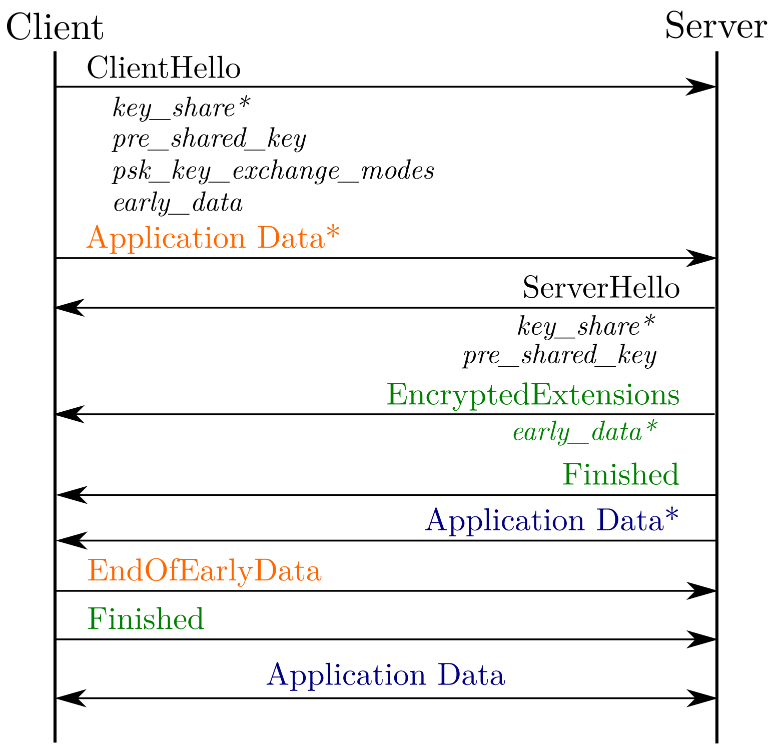

Figure 2.6: Communication scheme between the Client and the Server during the session resumption TLS 1.3 handshake (Taken from [22]). Optional messages are marked with *. Green (handshake traffic keys), blue (application traffic keys) and orange (early traffic keys) colors indicate encryption. Cursive indicates notable extensions.

0-RTT handshake also has some downsides to it. It is vulnerable to replay attacks, meaning that an adversary might capture a *ClientHello* message with early data and replay it with a chance that the server will accept that as valid. That is why there is a need to carefully decide what can and what cannot be sent as early data, because TLS provides little protection against this. Applications such as HTTPS clients can reduce the risk of a replay attack by sending only idempotent methods (i.e. methods that do not change the server's state) as early data.

Chapter 3

Side-Channel Attacks

This thesis focuses on testing against side-channel vulnerabilities. As opposed to classical cryptoanalysis, which focuses on cryptographic system as a black box with input and output, analysis of side-channels is concerned with implementation of such systems and information they involuntarily give out in addition to their output.

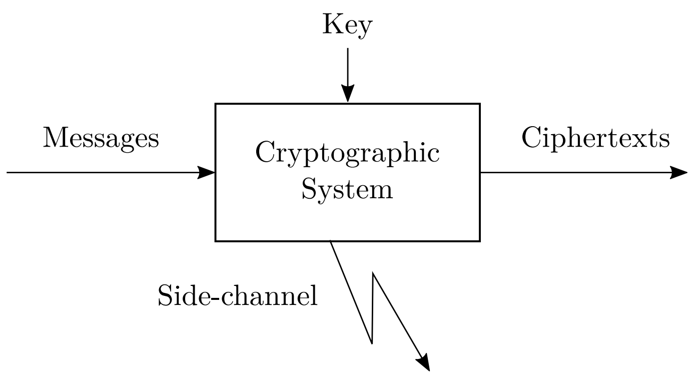

Figure 3.1: A scheme of cryptographic system and its side-channel. Taken from[1].

Figure 3.1 shows a scheme of such a cryptographic system leaking additional information through a side-channel, that the attacker is able to observe. Side-channels are generally created due to the nature of the system's implementation, both in software and in hardware. Even though the algorithm might be unbreakable in theory, it still has to be implemented in some way and run on some kind of hardware, all of which creates a new potential attack vector for the adversary. What follows is a non-exhaustive overview of different kinds of side-channels. Information for each type of side-channel is taken from [16] and from their respectable entries in [20], unless stated otherwise.

3.1 Timing Side-channel

Timing side-channel attacks exploit the fact, that certain parts of the cryptographic system might take a longer time to compute under certain circumstances. A prime example of timing side-channel might be a naive approach to password verification that focuses on delivering the result as fast as possible. Consider an algorithm, that would compare the entered password and the stored password bit by bit and in case any bits in such pair don't

[1]https://koclab.cs.ucsb.edu/teaching/ccs130h/2017/sidechannel.pdf

match, immediately return a failure to match passwords. The attacker is able to select the input to the system – the entered password and in order to conduct the attack they would permute the first character of the password and record the duration of verification for each such password. Given this process would be repeated enough times to account for noise that might be caused by other programs running on the system or operating system's context switching, and that some statistical tools would be applied to the recorded results (e.g. median or mean), one of the characters would have a slightly higher processing time, because the algorithm would also check the second pair of characters since the first pair would be correct. Repeating this process for each next character, the attacker would eventually be able to recover the whole password.

While the previous example is more of a theoretical one, to demonstrate the mechanism of timing side-channel attack, more practical attacks were described that manage to break implementations of cryptographic algorithms – most notably the first timing attack ever published that focused on exploiting Montgomery multiplication and Chinese remainder theorem in implementation of RSA and other cryptographic systems [19].

The countermeasures for avoiding creating timing side-channels are often hard to get right and often come at the cost of losing speed. One of the ineffective, yet often the first one that comes to mind is adding a random delay. This is however successful only in increasing the number of samples needed for the attack, but not avoiding it completely. The probability distribution of the random generator in most cases approaches normal distribution, therefore with enough samples, the timing signal is not lost. The correct mitigation of timing side-channels is making the operations that depend on secret information constant time. In the first example with password verification, this would mean always comparing the whole password, and setting a flag in case of failure to match any of the characters. Alternatively, as Kocher suggests [19] in preventing timing attack on RSA, the internal state can be *blinded*, meaning that the attacker would not be able to reproduce the internal computations. This however, as Kocher notes, might not prevent *all* of the possible timing attacks.

3.2 Power Side-channel

Power side-channel focuses on the correlation between power consumption, the input data and cryptographic operations that run on the observed hardware. This typically applies to less complicated devices, such as smart cards. As opposed to timing side-channel attacks, the implementation itself is often not known. The device, such as smart card reader, is monitored through an oscilloscope by inserting a resistor in series with the ground or power supply pin. The obtained measurement is called a *power trace*. Most methods operate under *Information Leakage Model* – assumption that the consumed power is related to e.g. Hamming weight of the data (this includes the input data, instruction, addresses, etc.), or Hamming distance of the XOR of the current state and the previous state, resulting in the number of flipping bits between the states. Other possible models exist.

The first and least complicated method is *Simple Power Analysis* (SPA), that can be used to identify individual instructions in the power trace. It works by using different input data, measuring the power trace for each one of them and then plotting them all in one graph. At certain places the traces will differ because of different inputs. These places are called *signatures*. Assuming the leakage model is known (let it be Hamming distance for the sake of this example), then reverse engineering of the instruction would be performed as follows:

1. Measure the power consumption at the place of chosen signature for all possible input values and plot that into graph A (`x`, consumption(`x`)).

2. Plot a graph showing the Hamming distance of $\mathtt{x} \oplus \mathtt{y}$, where `y` is an instruction code, for each value of `x`. Doing this for all of the instructions would result in a set of graphs $B_{y_0}, \ldots, B_{y_n}$.

3. Choosing the most similar graph B_{y_i} to A would identify the instruction – y_i.

Although the SPA is often effective on single-purpose devices as smart cards, it usually fails when it comes to more complicated systems, where noise is present. This where *Differential Power Analysis* (DPA) is usually better applicable. It makes use of statistical methods, applied on a large number of power traces. It can be used to recover a secret key. As mentioned before, it takes a number of random power traces (e.g. a number of cipher texts for the same key), that are then divided into two sets depending on the value of a selected bit. Next, the average consumption for each set is computed for a selected time interval – the *DPA trace*. This trace will show spikes for when the selected bit was manipulated, due to Hamming weight properties of both sets. This allows the attacker to follow the changes to a chosen bit and this technique is thus called *bit tracing*. This attack was successfully mounted against DES[18] and AES[13].

Although both methods seem hard to defend against, there are countermeasures that can be taken both in hardware and in software. In hardware, it can be aimed at making the signal too noisy to use or using desynchronization to misalign a set of traces. The other approach is called *precharged dual-rail logic* and focuses on doubling the circuitry in order to always have to switch in the transition to the next state, thus balancing the power consumption. In software it can be e.g data masking.

3.3 Cache Side-channel

Cache side-channel attacks are one of the more complicated ones, that can be executed even on modern computers with programs running in parallel. Cache in this context is the CPU cache, usually much smaller in storage than the main memory, but much faster. When reading data, the CPU first checks the cache and if the requested data is present (cache hit), it uses it. Else, the main memory has to be queried for the data (cache miss). The actual attack vector uses timings or power consumption of measurements of cache hit/cache miss. The third approach uses its own spy process to determine cache hits and misses.

Trace-driven attacks, that use power traces to determine a cache hit or miss require the attacker to first clear out the cache by reading or writing large chunks of data, so the attacked cryptographic system has no data in cache. Assuming the algorithm uses S-boxes (as e.g. AES does) that are accessed by deriving the index from certain bytes of the secret key and the plaintext that is being encrypted, and that the attacker is able to measure the trace indicating a hit or a miss, they are able to reduce the number of possible combinations for a secret key. The attack is based on the assumption, that if the index is derived as $P_i \oplus K_i$, where P_i and K_i are the ith bytes of the plaintext and the key, the first time that index will be accessed ($P_1 \oplus K_1$), it will be a cache miss, and the second time ($P_2 \oplus K_2$) will be a cache hit. This gives us the equation $P_1 \oplus K_1 = P_2 \oplus K_2$ and therefore $P_1 \oplus P_2 = K_1 \oplus K_2$, thus giving the attacker the value for $K_1 \oplus K_2$. Similarly, the cache miss ($P_1 \oplus K_1 \neq P_2 \oplus K_2$) indicates some impossible candidates. The information can then be used to brute-force the key in a significantly shorter time.

Time-driven attacks rely on the assumption that encryption time can be used to approximate the number of cache misses and that the number of cache misses for plaintexts for which $P_1 \oplus P_2 = K_1 \oplus K_2$ is true will be less than for those for which the equation is false. The attack can be then carried out as follows:

1. Obtain time measurements for a large number of encryptions with the same key along with their respective plaintexts.
2. Compute every possible combination of $P_1 \oplus P_2$ (assuming these are bytes, that means 256 combinations) and sort the plaintext into these subsets according to XOR result of the selected P_1 and P_2 bytes.
3. Plot the average encryption time for each subset. Given a large enough sample, one of the subsets should be clearly distinguishable. This is the one for which the equation $P_1 \oplus P_2 = K_1 \oplus K_2$ is true.

Both of these approaches operate under assumption, that each cache block can only contain one item from the S-box, but that is usually not true on modern systems. So instead of determining a single item from the S-box, rather a subset of items is determined. While this may not seem as effective, it can still be used to carry out a successful attack.

The third approach uses its own spy process to carry out the attack. In this example, it is assumed the cache is mapped as a two-way set associative. First, the spy process fills the cache with its own data. Then it triggers encryption and starts accessing the data and measuring the access times. If the cache was full and the victim's process requests data (e.g. the S-box data), the cache has to evict a block in order to make space for the requested block. The spy process can determine, by the measurements of time that it takes to access its data, which one of its blocks got evicted (therefore resulting in cache miss), identifying the victim's structure that took its place. Since memory access is dependent on the secret key, this information can be used to derive it.

3.4 Timing Side-Channel Attacks on SSL/TLS

The main concern of this thesis is timing side-channel attacks in implementations of the TLS protocol. As described in the previous section, timing information is the basis of many side-channel attacks, what differs is the means of obtaining the signal. This is even more complicated in the case of network protocols because the timing information is distorted along the way from server to client. While it may seem as if this would make it impossible to conduct a real-world attack, there are still ways to extract the approximation of the original timing signal. Crosby [10] proposes this model for timing information in network context:

$$responseTime = a \cdot processingTime + propagationTime + jitter$$

The *responseTime* is composed of the original timing information (*processingTime*) that is multiplied by server's CPU clock skew (a), added to *propagationTime* that indicates how long it takes for the information to arrive across the network and finally *jitter*, that accounts for the random delays occuring along the way. As Crosby found, the probability distribution of the timing information travelling trough a network is non-Gaussian, is highly skewed and multimodal. This means that e.g. averaging the timing information over a large number of samples is not very effective. This poses a challenge on how to analyze obtained samples in order to extract an approximation of the original timing signal.

Next, selected timing side-channel attacks on TLS will be described briefly. Both of these attacks can be categorized as chosen-ciphertext attacks. This category of attacks can be described as the attacker having the ability to choose ciphertext that is then decrypted with the target's secret key, resulting in attacker obtaining the corresponding plaintext. This process can reveal information that can be used to recover the secret key. Furthermore, chosen-ciphertext attack is called adaptive in case the attacker can choose the ciphertexts depending on the previous outcomes of the attack.

Both also use attack technique called *padding oracle*. This exploits an implementation that lets the attacker know if the padding on a message is valid or not, either explicitly or through a side-channel. Padding is used whenever the plaintext message is too short to fit the desired cipher's block size. The example usually given when talking about padding oracles is CBC (Cipher Block Chaining) method of encrypting messages longer than the cipher's block size and PKCS7 padding standard for symmetric ciphers, as this is the first attack conducted with padding oracles [25]. In this case, the attack can be used to decipher captured ciphertexts with the only condition being that the attacker has to be able to request decryptions from the server that has the associated secret key and also leaks indication if the padding is correct or not.

3.4.1 ROBOT

The ROBOT attack [6] (short for **R**eturn **O**f **B**leichenbacher's **O**racle **T**hreat) revives a previously introduced attack from 1998, originally published by Daniel Bleichenbacher [5], that exploited PKCS #1 v1.5 padding with RSA encryption and demonstrated it's functionality on SSL 3.0. This attack has been since improved on numerous times. The ROBOT attack was published in 2017, showing that even after 19 years from the original attack, the vulnerability has not been mitigated properly, and could be mounted on TLS 1.2.

This attack reuses the padding oracle originally discovered by Bleichenbacher's „million message attack". The attack's core is detecting the oracle which differs by implementation of the TLS protocol. The authors first request the use of TLS-RSA ciphersuite in *ClientHello* TLS message. Upon obtaining the server's certificate an item from the set of correctly formatted and malformed messages was sent in *ClientKeyExchange* message, and the server's response was monitored. If the server would adhere to the TLS standard, it would have responded to any malformed message with the same TLS alert. However, this was often not the case and the authors were able to obtain the server's private key using this oracle. As the authors point out, they didn't focus on timing variant of this attack. However, it is likely that timing side-channel was also present.

3.4.2 Lucky 13

Lucky 13 is another padding oracle attack, published in 2013 [4]. It exploits a known side-channel arising from MAC (Message Authentication Code) check, that was described in the RFC 5246 for TLS 1.2 [12], but was thought to be too weak to exploit. If incorrect padding is encountered during decryption, the MAC check has to be still performed, or a timing side-channel would be created, which was something the RFC was trying to mitigate by suggesting a zero padding is assumed in case of incorrect padding, so the MAC check can be executed. However, the MAC check is to a degree data size dependent, and this can be exploited in practice with the right length of the message. The attack allows for partial or full plaintext recovery, depending on the concrete TLS implementation. As for countermeasures, the authors suggest either using AEAD (which was implemented in TLS 1.3, see Section 2.2.1 for details) or making the CBC-mode decryption strictly constant time, which was the approach taken for mitigating this attack for TLS 1.2.

Chapter 4

Fuzz Testing

Testing is an inseparable part of modern software development process aimed at improving its quality. Fuzz testing (or fuzzing) is a specific kind of negative testing aimed at discovering bugs and security vulnerabilities. As opposed to positive testing, which inputs correct data to the system under test (SUT), negative testing provides the SUT with incorrect and unexpected inputs. In this chapter, fuzzing will be described in detail, followed by a description of the `tlsfuzzer` tool. Information in this chapter is obtained from [24] and [8] unless stated otherwise.

4.1 Introduction to Fuzz Testing

Fuzzing is a highly automated kind of testing, aimed at crashing the SUT by providing inputs to it. If we look at testing classification according to information the person has about the SUT, we find that it is hard to confide fuzzing to a single category:

- **White-box testing** – the tests are designed in a way that considers the internal structure of the code and in the ideal case, every path in the code is covered
- **Black-box testing** – the tests treat the SUT as black box with hidden inner workings, that has inputs and outputs.
- **Grey-box testing** – a combination of previously mentioned approaches. The tests are designed with partial knowledge about the SUT – e.g. the algorithms used in it

While there is not a consensus on where fuzzing falls, it usually leans more towards the black-box testing or the grey-box testing. Programs that generate or perform fuzzing tests are generally called fuzzers.

History of fuzzing

The fuzzing techniques first appeared at the late 1980s resulting in the publication of a paper [21] that described fuzzing various UNIX utilities with random inputs and found that many of such programs were crashing or hanging indefinitely. This inspired researches in Oulu University Secure Programming group and they later founded a project named *PROTOS*, that was aimed at providing vendors with free fuzzing test suites for popular protocols, such as LDAP, SNMP, SIP or DNS, that were successful in finding critical vulnerabilities and helped to introduce fuzzing as a useful testing technique.

Fuzzer types

One way of categorizing fuzzers could be according to test complexity:

- **Static and random template-based** – almost no awareness of the protocol, only for simple e.g. request-response testing purposes.
- **Block-based** – implement basic protocol structure, can contain some dynamic functionality like checksum calculation.
- **Dynamic generation or evolution based** – the protocol structure can be learned based on the feedback from the SUT.
- **Model-based or simulation-based** – the protocol is either modeled or fully implemented. Enables fuzzing of entire sequences of messages.

Another way of classification can be based on the attack vector. Although fuzzing is more of a black-boxing type, the box usually has multiple parts that provide multiple attack vectors, such as client-server type of applications. The fuzzers can support fuzzing clients, servers, or both, or even the middleboxes that might just serve as a proxy for the protocol.

Fuzzer structure

While the primary purpose of the fuzzer is to generate tests, modern fuzzers often go further than that in order to improve failure detection and test automation. The general structure is as follows:

- **Protocol modeler** – models the protocol in order to test for formats or message sequence. Can be templates or e.g. context-free grammars.
- **Anomaly library** – a collection of inputs known to trigger vulnerabilities, present in most fuzzers. If there's no such part of the fuzzer, random data is used.
- **Attack simulation engine** – uses a library of attacks or anomalies in order to generate tests for them or their random modifications.
- **Runtime analysis engine** – for monitoring the state of SUT.
- **Reporting** – if the fuzzer is successful in making the SUT misbehave, there should enough information provided about how it managed that in order to correct the SUT behavior.
- **Documentation** – includes the fuzzer documentation as well as documentation for the test cases, that might be dynamically generated.

Fuzzing cycle

Fuzzing cycle is a process of executing the fuzzer and its related parts and performing the fuzzing of a SUT. Its phases are enumerated below.

Target identification

The first stage is optional depending who the actor is in this case. For testers, the target is already chosen, as it is their task to test a particular system. However, for attackers the target is chosen accordingly to risk, impact and user base. Applications that present a high risk are usually those that receive input over the network and thus provide remote code execution. Another category of vulnerable applications is of those who run at a higher privilege level than the user, potentially vulnerable to privilege escalation. Lastly, there are applications that deal with confidential or valuable data, vulnerable to the attacker compromising their availability, confidentiality or integrity. Additionally, applications that share multiple categories of vulnerability make for an increasingly interesting target with high potential value.

Input identification

This phase is concerned with enumerating the attack surface of a system. Attack surface can be defined as a sum of all the interfaces, services and protocols available to all users, with emphasis on those available to remote and unauthenticated users. This is a critical stage because failure to properly identify the attack surface will result in not testing the attack surface in its entirety, thus leaving a potential attack vector. Some of the attack surfaces include command line arguments, environment variables, web applications, file formats, network protocols, memory, COM objects and inter-process communication. These input classes can be further categorized in those that result in a remote vulnerability and those that result in a local one.

Fuzz data generation

Another critical phase of the fuzzing life cycle is the generation of the fuzz data. The perfect fuzzer would be able to fully enumerate the full target input space to test it for all possible inputs. However, this is not a feasible approach because of the size of the input space. Hence, fuzzers take the approach of generating (possibly before the testing or on-demand) test case instances based on a set of rules defined by the user of the fuzzer. Each instance is then supplied to the SUT, which is monitored for failures, uncovering defects which can be reported back to developers to fix.

As mentioned in the subsection describing the classification of fuzzers based on test complexity, there are multiple approaches to generating test data, mostly varying by the level of awareness of the input space of the SUT.

Fuzz data execution and monitoring

The phases of test data execution and monitoring the SUT are both essential to each others functionality. Test execution greatly varies by a specific fuzzer tool but can be generally described as interacting with the specific input class in all feasible ways in order to cover as many real-world scenarios as possible.

As mentioned, supplying the SUT with fuzzed data is not enough in terms of discovering an actual vulnerability. Monitoring the SUT for any failures is an equally important part of this process. This is facilitated by an *oracle*, defined as a software component that monitors SUT and reports failures. The complexity of the oracle can range from a simple ping-based liveness check, through exception detection and classification, to a full system snapshot, however, this is not often feasible with regards to the number of tests being run and storage requirements that would be needed to save a snapshot for every single test. In the context of this thesis, it could be argued that a significant part of it is extending the fuzzer's monitoring system to be able to monitor the latency of the server's response.

Determining exploitability

The final stage of the fuzzing life cycle is utilizing the fuzzer's report of defects to identify actual vulnerabilities. This might consist of simply submitting the report to the developers of the SUT for correction. However, it might be also appropriate for the tester to further examine defects for exploitability and the possible impact for the end user.

4.2 Tlsfuzzer

Tlsfuzzer is a fuzzer aimed at testing the SSL/TLS protocol, maintained by Hubert Kario. The fuzzer is currently in alpha version and there are not any API stability guarantees. It is written in pure python with the only dependency being the `tlslite-ng` that implements the TLS protocol, making it a simulation-based fuzzer. It was created with the purpose of fixing the serious lack of testing coverage for the most popular TLS libraries. TLS poses a challenge in a sense that it is harder to test – it is a complex protocol that requires an implementation of it on the fuzzer side. Additionally, many attacks do not focus on just crashing the server, but rather on extracting a private key or decrypting a ciphertext using padding oracle, which is generally out of the scope of a typical fuzzer.

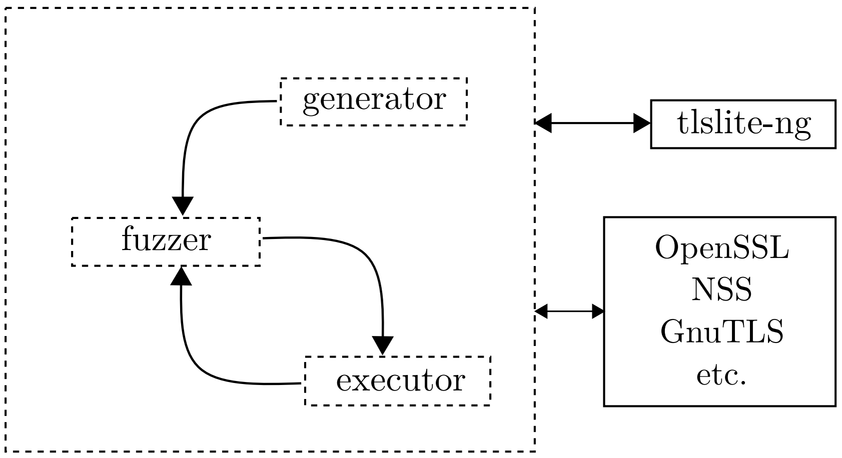

Figure 4.1: Tlsfuzzer architecture. Taken from [17]

The architecture of tlsfuzzer is shown in Figure 4.1. The fuzzing begins with the *generator* probing the peer for supported features and then taking connection templates and the detected features and using them to generate a sequence of messages and expected responses – a TLS conversation. This is then passed on to *fuzzer*, that has knowledge about which parts can be modified freely and which parts would cause the connection to abort, if modified. *Fuzzer* modifies the message generators according to this knowledge and passes them to the *executor*, that uses `tlslite-ng` in order to create, write and encrypt the protocol messages. However, at the time of writing this thesis, the only currently working part is the *executor* that is able to execute test scripts written using tlsfuzzer. Information in this section was taken from [17] unless stated otherwise.

Chapter 5

Design

This chapter describes the technical challenges that come with collecting high-precision timing information on network communication and with analyzing the collected data. Before moving directly to timing attacks on TLS (for that see Chapter 6) I used a prototype to test out various approaches to both recording timing information and statistical analysis. In this chapter, various approaches to each problem are described along with the selected solution.

5.1 Prototype

The prototype is a client-server application written in Python 3, that was publicly available on github[1] while it was being developed. Python was chosen because `tlsfuzzer` itself is written in it, and it is better to prototype within the limitations of that specific programming language. In its early stages, the prototype consisted of three modules with respective responsibilities:

- **Client** – packet collection, querying the server
- **Server** – responding to queries, simulating a timing side-channel
- **Analysis** – extracting timing information from packet capture and saving it in a parsable format

The client would start `tcpdump` as a separate process with a specified filter for the server as way to reduce the volume of unnecessary data to capture, and then query the server using a simple protocol many times over. Then, the analysis would extract timing information from the packet capture and sort them to their respective queries according to a log produced by the client. Then this data would be dumped to a csv file and the actual analysis would be performed on it using the R programming language for the sake of evaluating the methods of statistical analysis. Protocol used by the application is described in the Figure 5.1. The idea here is that BAD and BAAD should be indistinguishable from each other when looking at the response from the server, but should vulnerable to timing attack, with server simulating a timing side-channel with a slight delay when replying to a BAAD query. The server was originally implemented in Python, as well as the rest of the prototype, but later had to be rewritten in C, as simulating a consistent timing side-channel proved to be a

[1] https://github.com/kosciCZ/timing-analysis-poc

challenge by itself and was ultimately the reason why I moved from the prototype to a real implementation.

Table 5.1: Protocol used by the prototype application

Name	Query	Response
GOOD	00	0
BAD	01	1
BAAD	11	1

5.2 Collection of Timing Information

The first problem that needs to be solved is the collection of high-precision timestamps from the network communication. Several approaches can be taken, however, it is important, that the technical solution is available on a typical OS and does not require specialized hardware, such as custom network interface cards or FPGAs. There are several places, where the measurement can be taken.

The incoming packet first has to reach the physical device, which then, after processing the packet and pushing it to a kernel buffer, sends an interrupt to the kernel, that invokes the respective routines to handle the packet. Next, the processing is done on the IP layer, where the IP header is stripped and defragmentation is performed. Then processing on the Transport layer is done, where the protocol-specific checks and actions are performed and finally, the packet, again stripped of the transport protocol header, is passed to the target socket, where Application layer processing can begin [14].

Of course, each transition through a layer adds a small delay that varies depending on how busy the kernel is, how many interruptions take place at a given time, etc. The researched approaches will be described below, along with an explanation for which one was selected.

5.2.1 Application Layer Collection

A Large portion of projects and demo applications that were encountered during research tried to make use of the tools available to measure time in the Application layer. Some examples of such projects can be the Time Trial [11], where the relevant part is implemented in C, and uses `clock_gettime()` with `CLOCK_MONOTONIC` as a way to measure response latency. As a mitigation of the various issues that come with trying to measure time accurately in the Application layer, they dedicate an entire processor core to the application to avoid unnecessary context switching and most of the interrupts. This seems to be a common technique to improve accuracy.

Some projects take a different approach and use hardware counters to gather timing information. This includes the tool used in the Lucky 13 attack [4], the FAU-timer (now mona-timing-lib[2]), that again uses a C implementation for the timing information collection part. More specifically, they use inline assembler with the `rdtsc` instruction that utilizes a dedicated `Time Stamp Counter` register and the `cpuid` instruction that prevents out-of-order execution. However, as this somewhat relates to the CPU frequency (depending on the individual processor) it is affected by frequency changes e.g. power-saving, turbo-boost, etc, therefore these must be disabled to get a reasonably accurate measurement.

To summarize, the main advantage in measuring timing information in the Application layer is the convenience of recording the information and possibly processing it as it comes. However, the accuracy of the measurements might suffer, be it by travelling through the network stack and kernel or by the nature of the hardware counters.

5.2.2 Tcpdump

Tcpdump is another approach to acquiring timing information that uses lower network layers than the Application layer. Tcpdump uses `libpcap` to capture packets. The `Libpcap` library operates in the userspace, however it utilizes `BPF` (Berkeley Packet Filter) virtual machine that is running inside the kernel. This way it is able to copy packets directly from the kernel buffers that the physical devices write to. It also allows for effective filtering of the packets by implementing a language of the same name, that the filters can be supplied in.

In comparison to the approaches mentioned in the previous section, it is already an improvement in terms of when the packet gets to a userspace process, because it doesn't have to pass through the network stack in order to get there, instead it is passed directly by the `BPF`. And while it is possible to utilize this improvement by writing a program that interacts with the `BPF` through `libpcap` API and used the techniques for measuring time described in the previous section, it is better to make use of the `libpcap` functionality by using its native timestamps[2]. These timestamps can be categorized depending on where in the system the timestamp is created. The manual page defines `HOST` based timestamps that are created when the time-stamping code reaches the packet. Several subtypes are defined here, the difference being precision and whether the clock is monotonic, meaning that whether it can go backwards. The other type defined here are `ADAPTER` based timestamps, where the timestamping is done directly on the physical network interface, providing the highest precision possible, however still not accurately representing the time when the first bit of a packet reaches the interface, but rather anything between the first and last bit of the packet.

While this approach allows for more precision (especially with the right hardware), it comes with a requirement of additional processing of the packet capture file, associating it with the conversation between client and server that took place. This, unfortunately, adds a package dependency to the prototype, because a packet capture library is needed. During the prototyping process numerous libraries were tested (e.g *ScaPy*) but were found to have too much overhead and eventually *dpkt* was selected, because it was much faster in comparison. The packet parsing library is used for decoding TCP and IP protocol headers and matching the message to either client or server.

[2]https://github.com/seecurity/mona-timing-lib

5.3 Statistical Analysis

The goal of the statistical analysis performed on the gathered data is to identify a timing side-channel. This means identifying a case, for which the timing response should be similar to others, but instead is distinguishable from the other cases. However, to perform such analysis, enough samples have to be collected in an effort to account for the noise that network transmission and operating system introduces to the original signal.

While there are many possible approaches to this, the basic ones (e.g. median, mean) have to be ruled out because of the nature of the probability distribution of the responses over network. As pointed out in Section 3.3, the distribution is highly skewed, multimodal and non-Gaussian. This forces the use of more sophisticated methods for the analysis.

While it would be ideal to deploy some sort of statistical test and be able to accurately spot a timing side-channel, such tests are not always accurate and can provide false positives and false negatives. That is a reason why such tests should be only a part of the decision and supporting statistics and plots about the collected data should be provided. The information in the following subsections is taken from [15].

5.3.1 Relevant Statistical Plots

What follows is an enumeration of relevant plots that can aid in the decision whether there is a side-channel present. The attached plots in figures are taken directly from the output of the framework's extension. If there is a legend present in a plot, it contains only numerical indexes as individual plot labels. This is necessary in the context of the framework, because the label can be an encryption key, that when displayed would lessen the plot's readability because of its length. Instead, only numerical indexes are displayed and can be later associated with a label in one of the files the framework outputs.

Scatter Plot

Scatter plot is one of the more basic plots, that shows the direction of a relationship between two variables. In the context of repeatedly measuring timing differences, the variables here are the set of measurements on the y-axis and index of the measurement on the x-axis. This plot is useful in assessing how the measurement went in general, spotting spikes in latency, determining outliers, etc. While the plot does not provide that much information about the distribution of the values, it is good for the general overview and can be used as a basis for further analysis. An example scatter plot is shown in Figure 5.1. The plot shows ten thousand measurements for three classes of queries with class marked as 0 being clearly distinguishable from the other two.

Box Plot

Compared to scatter plot, box plot provides more useful information about how the values are distributed and if the difference is significant enough, can be by itself sufficient to spot a timing channel. Box plot for each set of measurements contains the following five values: the minimum value, the first quartile, the median, the third quartile, and the maximum value. The quartiles are displayed as the lower and upper boundary of the „box“, with a line inside the box signifying the median, and minimum and maximum values displayed as „whiskers“ coming out of the box on respective sides. Figure 5.2 shows what a boxplot looks like. One of the classes of collected samples marked as 0 is clearly distinct from the other two classes and would be pointing to a timing side-channel if the classes would be expected to be indistinguishable.

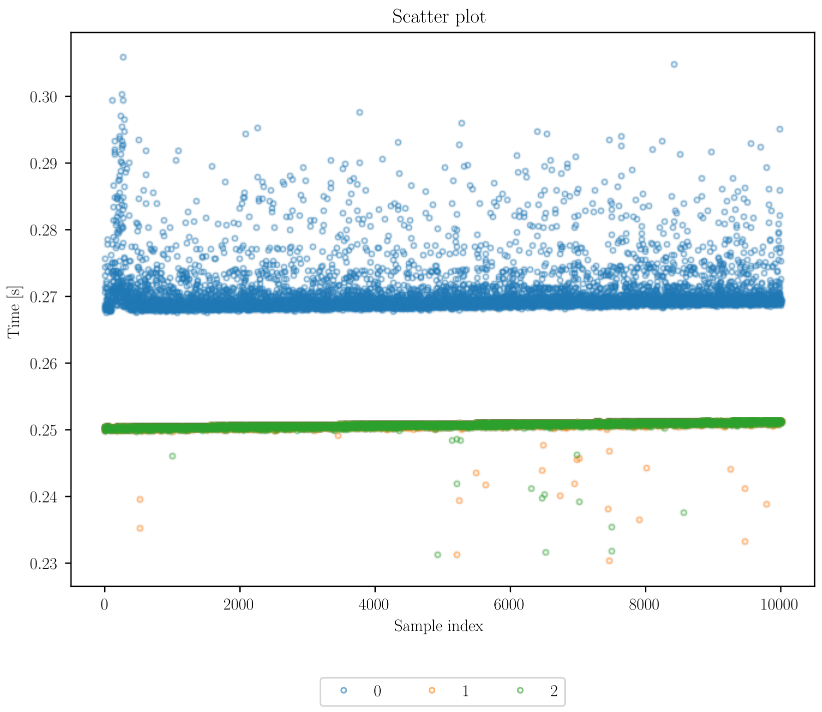

Figure 5.1: Example scatter plot of ten thousand measurements for three classes of queries.

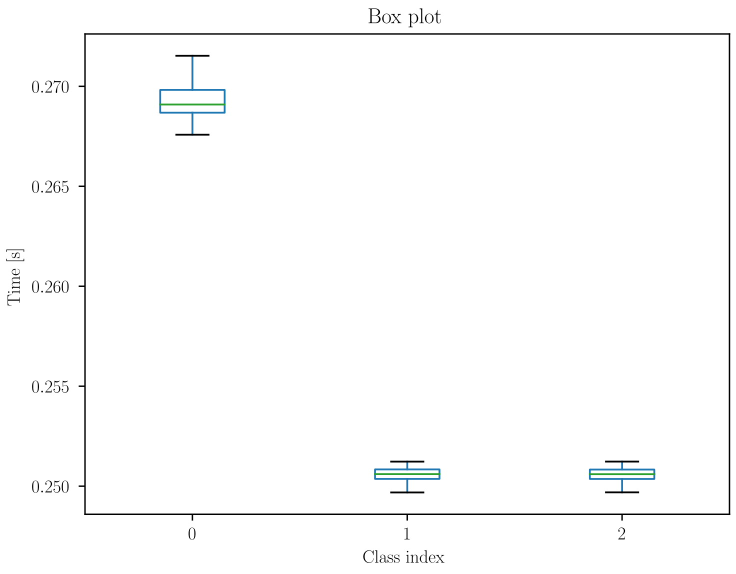

Figure 5.2: Example box plot of several sets of measurements, each with ten thousand samples.

Q-Q Plot

The Q-Q plot stands for Quantile-Quantile plot and is used to compare two probability distributions by plotting their quantiles against each other. Each point in the plot takes one coordinate from the first distribution from the second. If one were to compare two identical distributions with equal quantiles, the point would all be on the $y = x$ line. If one distribution would be just linearly transformed, the line would still be close to a straight line, however under a different angle. If the points form a line that curves at some point, that indicates that one of the distributions is more heavily skewed than the other. Such a plot could be useful in seeing how similar the distributions of the collected samples are when compared to each other. When comparing multiple classes of samples, it is useful to show a grid of plots where column indicates the variable on x-axis and row indicates the variable on the y-axis. On the diagonal, the class is compared to itself, thus resulting in a straight line. Figure 5.3 shows three classes cross-compared. As in previous plots of this data, samples from classes 1 and 2 can be observed to be similar, because their Q-Q plot loosely resembles a straight line.

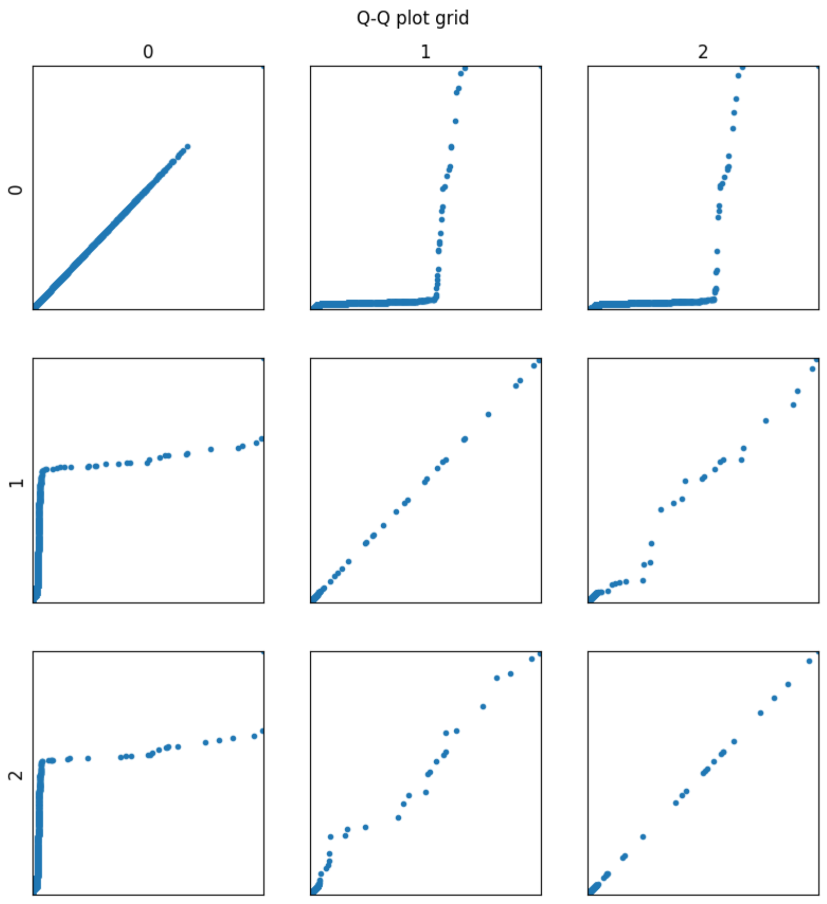

Figure 5.3: Example Q-Q plot grid comparing three classes of samples.

Empirical Cumulative Distribution Function Plot

Another useful plot for observing the probability distributions of the collected samples is that of the Cumulative Distribution Function (CDF). In this case, it is actually the Empirical CDF (ECDF) because the actual distribution is only estimated through the observed data. On the x-axis, the observed values are plotted and on the y-axis the probability of the value being less or equal to the value on the x-axis. In this context, it is useful for a more detailed look at the shape of the distribution than for example the box plot. An example with three distributions compared is shown in Figure 5.4. The distributions labeled as class 1 and class 2 overlap with each other because the difference from the distribution 0 is too significant to display small differences between them, effectively merging them together.

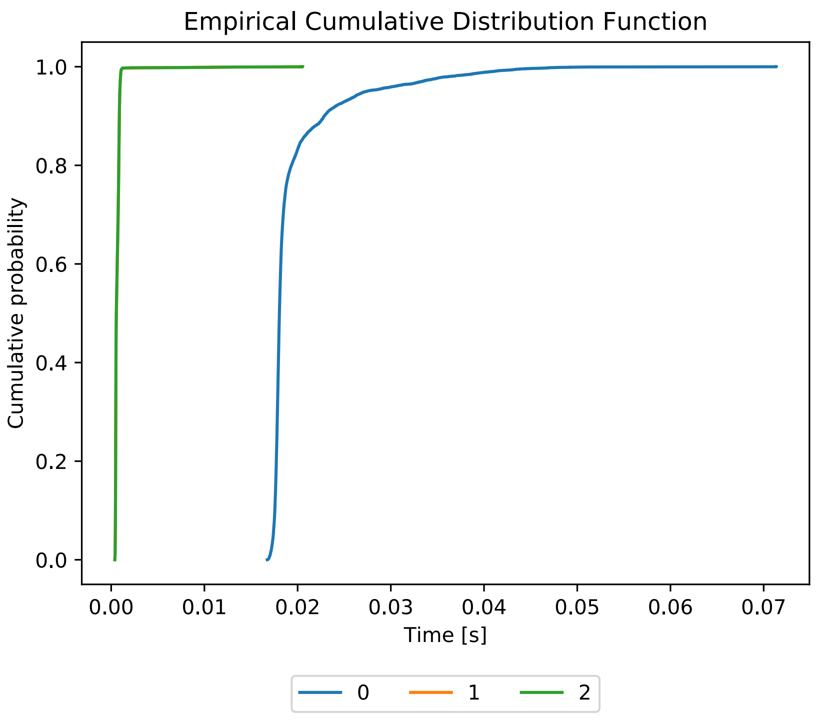

Figure 5.4: Example ECDF plot comparing three probability distributions.

5.3.2 Statistical Tests

An alternative to looking at plots and various other statistical properties of the collected data is statistical testing. Such a test is the closest thing to a binary answer on whether there is a significant statistical difference between the sets of measurements. Tests can be either one-sample, where the test is usually examining if the sample comes from a specific probability distribution (e.g. the Shapiro–Wilk test for normal distribution), or two-sample where two sets of samples are compared on various properties to determine if they came from the same distribution. As mentioned before, the probability distribution of packets travelling over a network does not resemble any of the well-known distributions, thus only two-sample tests will be considered from now on. Statistical tests are sometimes also called *Hypothesis tests*, because such test usually operates with two contradictory hypotheses.

- **The null hypothesis** H_0 – A claim about the two samples that implies there is no significant statistical difference between the two.
- **The alternative hypothesis** H_1 – A contradictory claim to H_0 that becomes the conclusion if H_0 is rejected.

The outcome of the test can be either to **reject** H_0 or **do not reject** H_0. The unability to actually accept either hypothesis implies the probabilistic outcome of the test. Table 5.2 describes all four possible outcomes of a hypothesis test, with two correct outcomes and two incorrect outcomes that are further differentiated by the type of mistake that occurred.

Table 5.2: Possible outcomes of hypothesis testing

	H_0 is	
Decision	True	False
Do not reject H_0	Correct result	Type II error (false negative)
Reject H_0	Type I error (false positive)	Correct result

As mentioned, (not) rejecting the null hypothesis is not a straightforward process. A value called the *level of significance* α must be chosen, describing the point at which the null hypothesis is rejected. As this is a strong statement, meaning there is a lot of supporting evidence for it, the α value is usually chosen as 0.05 or 0.01. The level of significance α also equals to the probability of Type I error occurring, meaning that the null hypothesis was incorrectly rejected. Another concept that is common in hypothesis testing is *p-value*, which, under the assumption that the null hypothesis is true, is the probability of the test statistic being equal or larger than the one observed. The significance level α relates directly to the p-value, where $p < \alpha$ means rejecting the H_0 and $p \geq \alpha$ means not rejecting the H_0. Relevant tests will be described in their respective sub-sections.

Box Test

The box test was first introduced by Crosby in [10] as an alternative to classical hypothesis testing and it is the result of an extensive research and testing. Their research showed that the lower percentiles contain the least amount of noise introduced by the network transmission and this is also the basis for the box test.

The test assumes two sets X and Y, each of N measurements. The test is parametrized by two parameters – quantiles i, j. Given these inputs, each subset is first sorted and then two intervals are formed, one for each subset; $I_X = [q_i(X), q_j(X)]$ and $I_Y = [q_i(Y), q_j(Y)]$. The sets X and Y are considered statistically different (signifying an exploitable side-channel) under two conditions – I_X and I_Y do not overlap and I_X is before I_Y in its entirety. Authors note that their testing yielded $i, j < 6\%$ as the best indicator for over half of the hosts under test.

Kolmogorov-Smirnov Test

In contrast to the box test, Kolmogorov-Smirnov test (also known as K-S test) is a classical hypothesis test. It useful in the context of this thesis because it is nonparametric, meaning it makes no assumption about the distribution under test and is not based only on statistical properties of the distribution such as mean or median. This works well with the highly skewed probability distribution of packets travelling over the network. K-S test can be both one-sample, where the null hypothesis is that the sample is drawn from the reference distribution, however for purposes of distinguishing network responses, the two-sample variant is more useful, where the null hypothesis is that the two distributions are drawn from the same distribution.

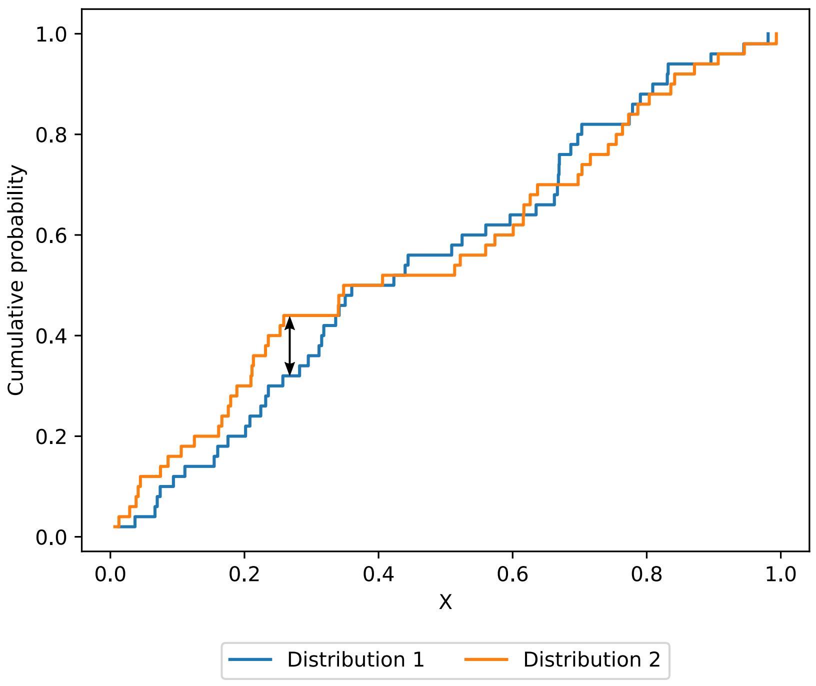

Figure 5.5: Visualization of the K-S test with the D value represented by an arrow.

The test is based on seeking out the largest difference between the empirical distribution functions (*EDFs*) of the two samples, as visualized in the Figure 5.5. EDF is the estimation of cumulative distribution function from observed values. The calculation of maximum difference is formally defined as follows:

$$D_{n,m} = \underset{x}{sup} |F_{1,n}(x) - F_{2,m}(x)|$$

Here, the n and m are the sizes of the respective samples, F is their EDF function and sup denotes the supremum function. The test statistic Z is then calculated like this:

$$Z = D_{n,m} \sqrt{\frac{n+m}{n \cdot m}}$$

A p-value is then computed using the Z value and the null hypothesis is then either rejected or not rejected depending on the selected level of significance α, as with any other hypothesis test. Information in this subsection was taken from [9].

Chapter 6

Implementation

The main goal of this thesis is to extend the tlsfuzzer tool with support for collecting timing information about the TLS conversations between the client and the server. This chapter describes how the knowledge gathered during creating a prototype (as described in Chapter 5) was used to implement the extension to tlsfuzzer, along with test cases for timing attacks described in Chapter 3.

6.1 Integration with Tlsfuzzer

Similar to the prototype created during the research part of this thesis, tlsfuzzer is implemented in Python, however, two of its main goals are to be portable, which means being able to run on older systems (as far as Python 2.6), and to avoid having too many dependencies to preserve compatibility and keep the framework as simple as possible. Keeping those goals in mind, the extension was designed to be as little intrusive as possible and to use only a minimal number of dependencies. The final structure is shown in Figure 6.1. This schematic view is based on the original tlsfuzzer structure from Figure 4.1. The extension consists of four modules, namely the Timing Runner, Log, Extraction and Analysis, all of which will be described in detail below.

6.1.1 Timing Runner

The core of the extension is the timing runner. Its responsibility is to wrap around the test executor and facilitate repeated runs for each requested test case while running tcpdump in the background. Timing runner is also the only part of the extension the person writing a testing script needs to interact with. Listing 6.1.1 shows how the timing runner can be integrated into a test script and demonstrates how it works.

First, a check is performed, whether tcpdump is available. This is done for compatibility reasons and not to disturb the script execution in case tcpdump is missing. The whole code block is wrapped in an if-statement, so the test will only fail on the timing part if the user has specifically requested timing of the test. Next, the TimingRunner class is initialized, mostly with command-line arguments and a set of testing conversations that utilize tlsfuzzer's capabilities. The network interface on which the communication will take place also has to be provided, because it is needed for tcpdump. Although Tcpdump's --interface any exists, the underlying packet parsing library in Analysis does not fully support the SLL protocol and thus a network interface needs to be specified in order to detect the correct link layer protocol.

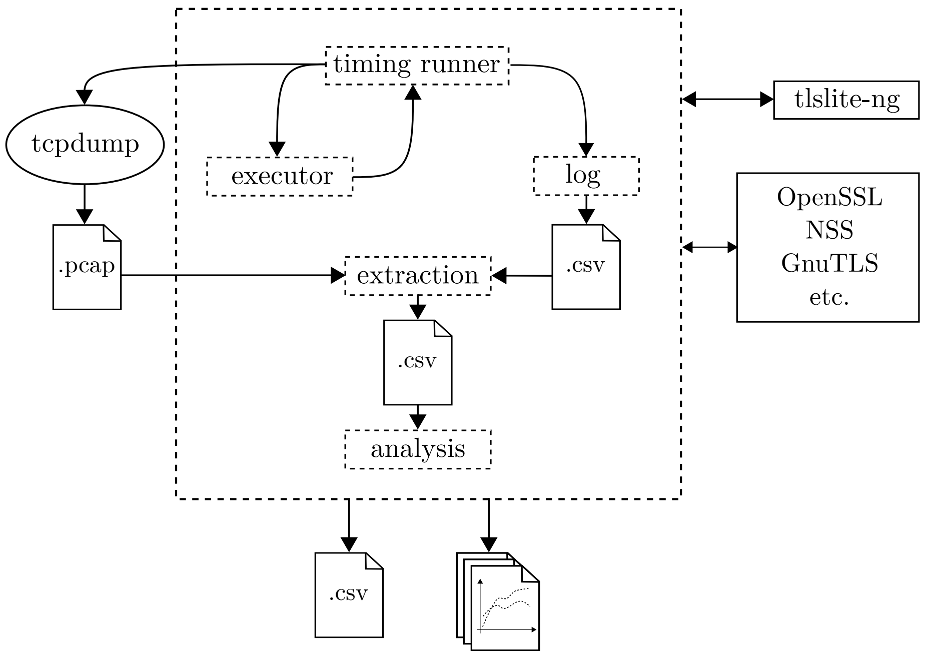

Figure 6.1: A schematic look at how the extension integrates into `tlsfuzzer`.

Next, a log is generated, again utilizing an existing test case structure with test case filtering that is standardized in `tlsfuzzer`, and the number of repetitions for each test case. The runner creates a log file, generates the requested number of *runs* and saves the log. A run in this context means a randomized order of classes, where *class* represents a group of test cases that are assumed to have the same timing characteristic. Random order is used here in an effort to spread any network or OS jitter more or less evenly over the classes.

```
# if regular tests passed, run timing collection and analysis
if TimingRunner.check_tcpdump():
timing_runner = TimingRunner(sys.argv[0],
sampled_tests,
outdir,
host,
port,
interface)
print("Running timing tests...")
timing_runner.generate_log(run_only, run_exclude, repetitions)
timing_runner.run()
else:
print("Could not run timing tests because tcpdump is not
↪  present!")
sys.exit(1)
```

Listing 6.1.1: Code excerpt showing how timing runner is integrated to a test scenario.

Finally, the runner is executed with the `run()` function. This launches `tcpdump` that is then watched by a separate thread in order to detect any unexpected failures, such as running out of space on the device. Next, the test cases are executed and grouped by their class according to the generated log. As a kind of outlier filtering measure, a fixed number of conversations is executed to warm up the system's cache. These *warm-up* conversations are then ignored during the analysis. After finishing running the tests, `tcpdump` is given time to finish writing all the buffered packets and terminated. If optional dependencies have been installed, extraction is launched, followed by analysis. If at any point dependencies are missing, the timing runner exits and both extraction and analysis can be later launched separately on a different, perhaps a more compatible device.

6.1.2 Log

The `Log` module takes care of effective reading and writing of information about the conversations taking place. It serves as a connecting part between the timing runner and the extraction. The log file is formatted as a csv file, where the first line contains names of classes that represent test cases. Each subsequent line represents a run in the form of a list of indexes into the class list from the first line. This is more effective than just logging the class of each conversation being run, especially considering the class might be a whole encryption key, considerably enlarging the file.

The module is also optimized for dealing with a large amount of data, keeping as little in operating system's memory as possible. This is achieved by writing runs directly to a file instead of keeping them in memory and using a generator to read the log file line by line, instead of loading the whole file into memory. This is also why the csv format was chosen because its simple syntax allows for iterative reading.

6.1.3 Extraction

The third module is `Extract`. It handles the extraction of timestamps from the packet capture file of the test conversations made with the timing runner according to a generated log. As mentioned in Section 5.2.2, the analysis module uses the `dpkt` library to parse the packet capture. It keeps track of individual TCP connections and looks for both the last server and the last client message to calculate the server response time to a client message. This approach was chosen because `tlsfuzzer` allows individual tests to control when the connection terminates, therefore the test just needs to terminate the connection right after receiving the server's response in order to collect the timing information. Another reason for this approach is the parsing speed. Because the TLS protocol allows multiple messages to be contained within a single packet, matching packets to actual TLS messages would require parsing the TLS protocol, which would slow down parsing the packet capture significantly.

The acquired timing value either gets skipped in case it was part of the warm-up conversations or gets matched to its class using the log file that was created by timing runner. When all of the information is extracted, it is written into a csv file, each row beginning with a class name, and then a list of measured values follows.

6.1.4 Analysis

The last module implements the analysis. It processes the file output by extraction and generates a report with statistical tests and a set of supporting plots as described with examples of such generated plots in Section 5.3. It uses `scipy` and related libraries to implement statistical testing and plotting. The report is a csv file containing cross-tested classes and results for each statistical test performed.

6.2 Test Cases for Timing Attacks

One of the main goals of this thesis is to extend the `tlsfuzzer` test suite by scripts utilizing the newly added extension for performing timing analysis. As described in Section 3.3, the chosen vulnerabilities were the Bleichenbacher-based ROBOT and the attack called Lucky 13, both of which will be described below.

6.2.1 ROBOT

The test script for the timing side-channel vulnerabilities exploited by variations of the original Bleichenbacher attack is located in `scripts/test-bleichenbacher-timing.py`. This test script is largely based on the `scripts/test-bleichenbacher-workaround.py` script, where the non-timing tests for these vulnerabilities are located.

As mentioned in Section 3.4.1, this attack focuses on exploiting the PKCS #1 v1.5 padding scheme in TLS-RSA key exchange. This attack is focused on the *ClientKeyExchange* message during the TLS handshake. Recall from Section 2.1.2 that this message contains client-generated encrypted pre-master secret (PMS). If RSA is the agreed-upon key exchange, the structure of the message before encryption will be as follows: the message starts with bytes `0x00` and `0x02`, then a non-zero sequence of at least 8 bytes follows. Then there's a single `0x00` byte, followed by the PMS, that consists of two bytes that signify the TLS version, and 46 random bytes.

The attack exploits a padding oracle that lets the adversary distinguish messages that are decrypted to correct padding from those that decrypt to incorrect padding, revealing information about the encryption key used. In the original attack, the padding oracle was available through alerts sent back to the client. The countermeasure suggested in the TLS specification is to generate a random PMS with every handshake, should the padding check fail, or use the received PMS, should the padding check succeed. However, even this countermeasure was often incorrectly implemented, opening up a possibility of timing side-channel.

This is why it is important to test for all kinds of padding errors. The test script provided by this thesis repurposes the following test scenarios for the timing side-channel test:

1. Invalid or incompatible value in block type (second byte of the padding)
2. Zero byte in the non-zero byte sequence
3. Missing `0x00` byte before PMS
4. Missing `0x00` byte before PMS, with non-zero PMS
5. Invalid TLS version in PMS
6. Too short PMS
7. Too long PMS
8. Missing PMS

9. Too short padding
10. Too long padding

In addition to these existing checks, it adds two new ones, that fuzz MAC in the *Finished* message that immediately follows the *ClientKeyExchange* one.

6.2.2 Lucky 13

The test script for the Lucky 13 attack is located in `scripts/test-lucky13.py` file. As mentioned in Section 3.4.2, this attack exploits incorrect attempt to mitigate a previous timing attack that was based on the server only performing MAC check when the padding was correct, leaving a significant timing side-channel open. The specification for TLS 1.2 correctly states that in order to mitigate this attack, MAC check has to be performed even in the case of wrong padding, suggesting the implementations assume zero-length pad in this case. It also acknowledges that this might include a small timing side-channel, however stating that it is thought to be small enough to be not exploitable. The authors of the Lucky 13 attack then prove that this is a false statement and that when certain conditions are met, the difference is significant enough to be exploitable.

A test for this kind of attack thus needs to assert, that incorrect MAC detection needs to be constant-time for any possible length of padding, whether the padding is valid or not. This yields the following test scenarios:

1. Entirely incorrect MAC with valid padding
2. Correct MAC with a single-byte error in the longest possible padding
3. Single-byte error in MAC with valid padding
4. Padding indicating MAC is out-of-bounds

These scenarios then each produce a number of individual test cases. In the case of the test scenario 1., all possible lengths of padding are iterated. In scenario 2., the single-byte error is created on every possible position in the padding. Scenario 3 does the same, but for all MAC positions, while keeping the padding static. Finally, the last scenario 4 iterates over all padding lengths that indicate that the MAC tag is located „before“ the start of the message.

To make the scenario 4 possible, the length of the plaintext message before encryption needs to be small enough for the padding (maximum length of 255 bytes) to be able to point outside of the message for the MAC tag location. Because of the different message length, the server needs to decrypt less data, thus making the timing characteristic inherently different from the plaintext used in scenarios 1–3. This is taken care of in the test by separating scenario 4 into a separate test run, so it is not compared to other scenarios.

For achieving the full test coverage, all ciphersuites that use the CBC mode in the MAC function need to be tested. At this time, this includes `md5`, `sha-1`,`sha-256` and `sha-384`. However, given the number of tests cases (over 500) and the need to gather several thousands of samples for each test case in order to account for noise, the test script has a `-C` option to select a ciphersuite, and a `--quick` option to allow only a reduced number of edge cases that are likely to find a vulnerability. These test cases are 4 in total, combining incorrect MAC at first and last byte, and padding of maximum and minimum length.

Chapter 7

Testing

This chapter describes how the implemented extension was tested to assure correct functionality, and how its accuracy in spotting timing side-channel was evaluated, with gathered results.

7.1 Implementation Testing

The `tlsfuzzer` library enforces numerous checks and code inspections to maintain a high quality of the code that makes up the library and the extension this thesis provides to `tlsfuzzer` had to follow that. `Tlsfuzzer` is placed in a GIT repository managed by Github[1]. Changes are made via pull requests that have to fulfil a set of requirements in order to be merged into the master branch, becoming a part of the library.

First of those requirements is full test coverage of the changes added in the pull request. An ideal addition should not bring down overall test coverage of the library, which at the time of writing this thesis sits at 97 %. The side-channel extension fulfils this requirement by a set of unit tests, testing each of the modules added. These tests are located in the `tests/` directory in the root of the repository, namely:

- `test_tlsfuzzer_timing_runner.py` – tests for the Timing Runner module
- `test_tlsfuzzer_extract.py` – tests for the Extraction module
- `test_tlsfuzzer_analysis.py` – tests for the Analysis module
- `test_tlsfuzzer_utils_log.py` – tests for the Log module

Test coverage of the extension was measured by running the `make test` command in the root of the repository, which yielded 99 % resulting test coverage for the changes made by the extension alone. Overall, this slightly increases the total test coverage of the `tlsfuzzer` library. The tests are then executed in a Continuous Integration (CI) pipeline that utilizes `Travis`. Before the extension was added, tests were run against all Python versions from 2.6 up to the current version – 3.8. To test all of the possible scenarios that might happen on a user machine, the `Travis` configuration was expanded to test against each Python version with and without dependencies needed for extraction and where applicable, with and without analysis dependencies. In addition to test coverage, code was checked for Python code style violations with `pylint` linter and thoroughly inspected with code review from the maintainer of the library.

[1]https://github.com/tomato42/tlsfuzzer

7.2 Evaluation

In this section the extension for collecting timing information for `tlsfuzzer` will be evaluated on its ability to distinguish a timing side-channel via the test scenarios for attacks exploiting such side-channels that were added to `tlsfuzzer`.

7.2.1 Setup and Testing Environment

The extension this thesis provides for `tlsfuzzer` will be evaluated by verifying that the newly added test scenarios for timing side-channel attacks fail when run against vulnerable versions of popular TLS implementations, namely OpenSSL, GnuTLS and NSS.

The system that was used for testing was running with Fedora 30 as an operating system, with Intel Core i3-9100F 4-core processor operating at 3.6 GHz, with 16 GB of RAM and an SSD drive. In order to create ideal testing conditions, 2 cores were isolated using the following kernel command-line parameters – `--isol-cpus=2,3 --rcu_nocbs=2,3 --processor.max_cstate=1 --poll=idle`. These parameters isolate core 2 and 3 from regular usage by the OS and reduce the number of interruptions those cores receive. The application can be then bound to a specific core with `taskset`. During the evaluation both the TLS server and `tlsfuuzer` were running each on a separate core.

To set up the testing environment, Python has to be present on the system, preferably version 3.5 and higher to be able to run the timing tests in full, including analysis. For only the packet capture and extraction, just Python 2.6 is enough. Next, the `tlsfuzzer` repository is cloned, along with the accompanying `tlslite-ng` library that provides the TLS implementation. In addition, the dependencies listed in `requirements-timing.txt` need to be installed.

Server Setup

In order to run a TLS server, an RSA certificate is needed. It can be generated with the command 7.1, taken from `tlsfuzzer` documentation. This command request the creation of an RSA certificate (`-newkey rsa`) according to X.509 standard (`-x509`), for the localhost domain (`-subj /CN=localhost`), skipping any questions (`-batch`) and disabling key encryption (`-nodes`). This produces the certificate (`-out localhost.crt`) and associated key (`-keyout localhost.key`).

```
openssl req -x509 -newkey rsa -subj /CN=localhost -nodes -batch \
    -keyout localhost.key -out localhost.crt
```

Listing 7.1: Command used to generate certificate for TLS server.

Now the TLS server can be launched with any of the implementations that are going to be tested. For OpenSSL, the command in Listing 7.2 can be used to start the server, providing the certificate and the key (`-cert` and `-key`), and instructing the server to respond to `ApplicationData` messages (`-www`). The server will be started on `localhost:4433`.

```
openssl s_server -key localhost.key -cert localhost.crt -www
```

Listing 7.2: Command used to start the OpenSSL server.

Listing 7.3 shows how the GnuTLS server can be started. Repeatedly providing the certificate and the key (`--x509certfile` and `--x509keyfile`), again making the server behave as a HTTP one (`--http`). The server will be started on `localhost:4433`, as specified with the `-p 4433` option. In addition, client-side authentication is disabled (`--disable-client-cert`).

```
gnutls-serv --http -p 4433 --x509keyfile localhost.key \
    --x509certfile localhost.crt --disable-client-cert
```

Listing 7.3: Command used to start the GnuTLS server.

Configuration for the Mozilla NSS library is slightly different, requiring a database with a server certificate to be created. Listing 7.4 shows the complete process of creating such database, starting with creating a new directory on line 1 and then a new database is created in that directory on line 2. Next, the client certificate for localhost is generated on line 3. Then the generated certificated is inserted into the database on line 4. Finally, the NSS server itself is started on line 5, given the database (`-d`), binding it to `localhost:4433` as per `-p` and `-n` options.

```
1    mkdir nssdb
2    certutil -N -d sql:nssdb --empty-password
3    openssl pkcs12 -export -passout pass: -out localhost.p12 -inkey \
         localhost.key -in localhost.crt -name localhost
4    pk12util -i localhost.p12 -d sql:nssdb -W ''
5    selfserv -d sql:./nssdb -p 4433 -n localhost
```

Listing 7.4: Command used to start the NSS server.

Test Execution

As mentioned previously, the evaluation involves running two test scripts to validate the extension's ability to detect a timing side-channel correctly. First of these test scripts is testing for the ROBOT vulnerabilities, located in `scripts/test-bleichenbacher-timing.py`. Second test script is aimed at vulnerabilities discovered by the Lucky 13 attack, located at `scripts/test-lucky13.py`. Both tests are executed with the same command shown in Listing 7.5, that uses the loopback interface with the sample size of 5000 per class, with the Lucky 13 script adding the `--quick` option to reduce the number of test cases and the Bleichenbacher script reduced to only two test cases in order to speed up evaluation (appending `"invalid MAC in Finished on pos 0" "set PKCS#1 padding type to 3"`). Both tests were executed with the `TLS_RSA_WITH_AES_128_CBC_SHA` ciphersuite.

```
sudo PYTHONPATH=. python3 $TEST_SCRIPT -i lo --repeat 5000
```

Listing 7.5: Command used to execute the test scripts.

7.2.2 Results

Because the outcome of the test is influenced by more factors than just the server's implementation, such as processor cache or system interruptions, and therefore is not deterministic, each test script was run 100 times against vulnerable implementation and 100 times against implementation where the vulnerability was fixed, in order to make evaluating the extension's ability to detect timing channel possible.

Lucky 13

Results for the Lucky 13 test script are shown in Tables 7.1 and 7.2, the first one showing results against the vulnerable implementations for this attack, and the second one against current versions of the libraries where the vulnerability is fixed. The results in Table 7.1 clearly show that this particular timing side-channel was significant enough to be detected when vulnerability was present in all measured cases. The second conclusion can be drawn from Table 7.2. The statistical analysis is rather sensitive to any differences in gathered results and at times can identify noise introduced by the OS as a possible side-channel. However, over 100 trials, the results gathered from a vulnerable implementation are distinguishable from its fixed counterpart. To sum up the results, for vulnerabilities related to Lucky 13, there is a 100 % accuracy in identifying the timing-side channel when it is present, and 16.5 % average false positive rate when the side-channel is not present.

Table 7.1: Evaluation results for the Lucky 13 test script against vulnerable versions of TLS libraries.

TLS library	False Negatives (%)	Correct results (%)
OpenSSL 1.0.1c	0	100
NSS 3.13.6	0	100

Table 7.2: Evaluation results for the Lucky 13 test script against current versions of TLS libraries.

TLS library	False Positives (%)	Correct results (%)
OpenSSL 1.1.1g	21	79
NSS 3.53	12	82

GnuTLS is excluded from these measurements for two reasons. The first one being that on the current (fixed) version, the measurements yielded nearly 100 % false positives rate. This could mean two things, either the vulnerability is still present, or the implementation gives out noisy results, tricking the framework's analysis into pointing out the side-channel. The fact that GnuTLS is noisier than other implementations has been pointed out in the Lucky 13 paper [4]. The second reason is that the vulnerable version (GnuTLS 3.1.6) relies on a now-removed API in `glibc`, which made it unable to compile the library on any system that was available for the measurements. Given the high noise or the possibility of the vulnerability still present in the current version, it was decided to omit testing it for the Lucky 13 vulnerabilities.

ROBOT

Results for the Bleichenbacher-related vulnerabilities test script are displayed in Tables 7.3 and 7.4. For OpenSSL the side-channel is most likely less distinguishable, with the rate of false negatives reaching 31 % on the vulnerable version of the library, however, it is still clearly present among the 100 trials. The 100 % success rate on the fixed version of the library might be another indicator that the side-channel was small enough that the fixed implementation is approaching constant-time response latency even over the network. For the GnuTLS implementation the results yield near-perfect result, where, in the vulnerable version, the side-channel is detected in 100 % of the cases, and in the current version of the library the false positives rate is just 1 %. In the case of NSS, the situation is a polar opposite. The 100 % error rate in the current 3.53 version suggests that the implementation is not constant-time for all kinds of padding errors and should be examined for verification of the side-channel on the server side.

Table 7.3: Evaluation results for Bleichenbacher test script against vulnerable versions of TLS libraries.

TLS library	False Negatives (%)	Correct results (%)
OpenSSL 1.0.2f	31	69
GnuTLS 3.3.21	0	100
NSS 3.13.6	0	100

Table 7.4: Evaluation results for the Bleichenbacher test script against current versions of TLS libraries.

TLS library	False Positives (%)	Correct results (%)
OpenSSL 1.1.1g	0	100
GnuTLS 3.6.14	1	99
NSS 3.53	100	0

Summary

To sum up the results for both of the test scripts that were trialed, the extension is sensitive to even a small possibility of a timing side-channel. However, this comes with a tendency to show false positives when the side-channel is not present. However, over many trials, vulnerable and non-vulnerable implementations are clearly distinct.

Possibilities of further development might include moving the target server from the same host as the framework to another host on LAN to evaluate how the extension performs over the local network. Another possibility of improvement is related to the analysis module. The current classifier in the analysis module of the extension consists of two statistical tests. Two classes of samples are considered different when both of the tests agree that they differ in a significant way. The classifier could be extended in the future to include another test that would balance the classifier even more. The target system for these test scripts is a Continuous Integration (CI) pipeline, and the analysis is not deterministic, so this suggests a possible future extension, that will execute the timing variant of the tests multiple times and decide on the result based on the outputs.

Chapter 8

Conclusion

The main goal of this thesis was to improve testing of the TLS protocol by extending the `tlsfuzzer` tool, giving it an ability to test for timing side-channels. Furthermore, the functionality and the accuracy of the extension had to be demonstrated by adding testing scripts for known timing side-channels to the `tlsfuzzer` test suite, and evaluating them against popular TLS implementations.

These goals were met in their full extent, with the extension and test scripts for vulnerabilities based on the Lucky 13 attack and for vulnerabilities based on the Bleichenbacher attack, all merged into the library's master branch. The extension was also evaluated as per the thesis specification, with results suggesting a good ability to spot a side-channel when it is present and to distinguish a safe implementation from a vulnerable one over several trials. Furthermore, evaluating one of the test scripts hinted at a possible timing side-channel in the current version of a TLS library.

This thesis is different from the conventional approach to measuring timing differences over the network by offloading the capturing of that information to `tcpdump` that has the appropriate tools for that, instead of utilizing rather imprecise hardware counters provided by the CPU. An additional advantage of this approach is that a record of the communication is saved and can be later analyzed for other information than the timing of server responses.

Possibilities of further development include integrating the process of running the timing tests over many trials to the `tlsfuzzer` framework to allow for testing in CI and improve the accuracy of the extension as a whole. Further research and development are also possible in the analysis module, where more statistical tests can be added to contribute to the decision.

Bibliography

[1] *Transport Layer Security* [online]. Wikipedia [cit. 25. November 2019]. Available at: https://en.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=927737095#TLS_record.

[2] *Manpage of PCAP-TSTAMP* [online]. 2019 [cit. 7. May 2020]. Available at: https://www.tcpdump.org/manpages/pcap-tstamp.7.html.

[3] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M. et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In: *22nd ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, October 2015. ISBN 978-1-4503-3832-5. Available at: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf.

[4] Al Fardan, N. J. and Paterson, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In: *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, p. 526–540. ISBN 9781467361668.

[5] Bleichenbacher, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 1998, vol. 1462, p. 1–12. ISBN 3540648925.

[6] Böck, H., Somorovsky, J. and Young, C. Return Of Bleichenbacher's Oracle Threat (ROBOT). In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, August 2018, p. 817–849. ISBN 978-1-939133-04-5.

[7] Bright, P. *Apple, Google, Microsoft, and Mozilla come together to end TLS 1.0* [online]. 2018 [cit. 24. November 2019]. Available at: https://web.archive.org/web/20190728205529/https://arstechnica.com/gadgets/2018/10/browser-vendors-unite-to-end-support-for-20-year-old-tls-1-0/.

[8] Clarke, T. *Fuzzing for Software Vulnerability Discovery* [online]. Technical report. Egham, Surrey TW20 0EX, England: Department of Mathematics, Royal Holloway, University of London, february 2009 [cit. 26. May 2020].

[9] Corder, G. *Nonparametric statistics : a step-by-step approach*. Hoboken, New Jersey: John Wiley & Sons, 2014. ISBN 978-1118840313.

[10] Crosby, S. A., Wallach, D. S. and Riedi, R. H. Opportunities and Limits of Remote Timing Attacks. *ACM Trans. Inf. Syst. Secur.* New York, NY, USA: Association for Computing Machinery. 2009, vol. 12, no. 3. ISSN 1094-9224.

[11] Daniel Mayer, J. S. Time Trial: Racing Towards Practical Remote Timing Attacks. 2014. Available at: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf.

[12] Dierks, T. and Rescorla, E. *RFC 5246 – The Transport Layer Security (TLS) Protocol Version 1.2* [online]. 2008 [cit. 23. November 2019]. Available at: https://tools.ietf.org/html/rfc5246.

[13] Giraud, C. DFA on AES. In: *Advanced Encryption Standard – AES: 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*. Berlin, Heidelberg: Springer, 2005, vol. 3373, p. 27–41. Lecture Notes in Computer Science. ISBN 9783540265573.

[14] Herbert, T. F. *Linux TCP/IP stack*. Vyd. 1.th ed. Massachusetts: Charles River, 2004. ISBN 1-58450-284-3.

[15] Illowsky, B., Dean, S., Chiappetta, L., Desilets, L., Markus, L. et al. *Introductory Statistics*. 1st ed. Openstax, 2013. ISBN 9781947172050.

[16] K. Koç Çetin. *Cryptographic Engineering*. Springer, 2009. ISBN 978-0-387-71816-3.

[17] Kario, H. *TLS Test Framework*. [cit. 23. January 2020]. Available at: https://github.com/tomato42/tlsfuzzer/blob/master/docs/ruxcon2015-kario-slides.pdf.

[18] Kocher, P., Jaffe, J. and Jun, B. Differential power analysis. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 1999, vol. 1666, p. 388–397. ISBN 3540663479.

[19] Kocher, P. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 1996, vol. 1109, p. 104–113. ISBN 3540615121.

[20] Koeune, F. *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011. ISBN 978-1-4419-5906-5.

[21] Miller, B. P., Fredriksen, L. and So, B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. december 1990, vol. 33, no. 12, p. 32–44. ISSN 0001-0782.

[22] Rescorla, E. *RFC 8446 – The Transport Layer Security (TLS) Protocol Version 1.3* [online]. 2018 [cit. 23. November 2019]. Available at: https://tools.ietf.org/html/rfc8446.

[23] Smyth, B. and Pironti, A. Truncating TLS Connections to Violate Beliefs in Web Applications. In: *WOOT'13 Proceedings of the 7th USENIX conference on Offensive Technologies*. Washigton, D.C.: USENIX Association, January 2013 [cit. 3. December 2019]. Available at: https://www.usenix.org/system/files/conference/woot13/woot13-smyth.pdf.

[24] Takanen, A., DeMott, J. and Miller, C. *Fuzzing for Software Security Testing and Quality Assurance*. 1st ed. USA: Artech House, Inc., 2008. ISBN 9781596932147.

[25] Vaudenay, S. Security flaws induced by cbc padding – Applications to SSL, IPSEC, WTLS. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 2002, vol. 2332, p. 534–545. ISBN 9783540435532.

Appendix A

Contents of the attached CD

The attached CD contains the following directories and files:

- `xkosci00.pdf` – this thesis in PDF format
- `tex/` – LaTeX source codes of this thesis
- `tlsfuzzer/` – the `tlsfuzzer` tool with the side-channel detection extension integrated
- `README.md` – instructions on how to run `tlsfuzzer` with the extension