

École polytechnique de Louvain

A modular implementation of TLS

Author: **Antoine RIME**

Supervisor: **Olivier BONAVENTURE**

Readers: **Florentin ROCHET, Olivier PEREIRA**

Academic year 2019–2020

Master [120] in Computer Science and Engineering

Acknowledgements

First of all, I would like to thank my supervisor, Olivier Bonaventure, for his consistent guidance, availability and judicious remarks.

I would also like to thanks Florentin Rochet and Maxime Piraux for answering my questions and helping me throughout this work.

Finally, I would like to thanks my family and friend for their helps and support during the redaction of this thesis.

Contents

1	Introduction	3
2	TLS	6
2.1	Handshake	7
2.1.1	Key exchange	11
2.2	TLS Record	11
2.2.1	Padding	13
2.3	Encryption and AEAD	13
2.4	Key derivation process	15
2.5	PicoTLS	16
3	eBPF	18
3.1	What's eBPF?	18
3.2	uBPF	19
3.2.1	Helper function	19
3.2.2	Loading eBPF byte code	20
3.2.3	Executing eBPF byte code	20
3.2.4	Writing uBPF code	21
3.2.5	JIT	22
4	Implementation of plugins with uBPF	23
4.1	Insertion Point	24
4.1.1	Passing arguments	25
4.2	Registering protocol operation and plugins	26
4.3	Memory management	27
4.3.1	Interacting with the heap	28
4.3.2	Heap structure	28
4.3.3	Read and write on the heap	30
4.3.4	Pointer management	30
4.4	Interacting with picoTLS	31
4.5	Helper functions and Protocol operations	32

4.6	Conclusion	34
4.7	Future works - Exchanging plugins	34
5	Traffic shaping plugin	36
5.1	Website fingerprinting	37
5.1.1	WF attack	38
5.1.2	WF defenses	41
5.2	Padding plugin	44
5.2.1	Padding packets	45
5.2.2	Send dummy packets	46
5.2.3	Shape the connection	47
5.3	Evaluation of the defense	49
5.3.1	Gathering the dataset	50
5.3.2	DNS classifiers	52
5.3.3	Results	53
5.3.4	Conclusion and future works	56
6	Conclusion	57

Chapter 1

Introduction

Many of today's protocols use TLS to secure their connection; HTTPS [1], DoT [2], SMTPS [3]. TLS is used to encrypt data over the connection with the help of a master secret negotiated during the handshake. It provides privacy, authenticity and integrity to the connection. As security is a constantly evolving world, new attacks and defenses are proposed regularly. However, proposing a new feature to the standardized version of TLS takes some times. A new defense can be deployed as a new extension if the protocol permits it. If not, the defense is deployed in the next standardized version which can take a lot of times. For example, TLS version 2 was published in 2008 while version 3 was published in 2018. Furthermore, it often happens that for an exchange, the server or the client does not implement the most up to date version of the protocol. Thus forcing the other peers to use a deprecated version.

In this thesis, we explore a new way to implement network protocols making them more flexible and less dependent to the publication of standards proposed by organization such as the IETF.

We propose a new framework that speeds up the adoption of new features on TLS and gives the ability for developer to implement plugins that improve the behavior of the protocol. To do so, we add a virtual machine named uBPF to the protocol to execute plugins during the execution of the main process. Using a virtual machine to make protocols more flexible has been experimented on other protocols such as QUIC [4] or BPG/OSPF [5].

We combine uBPF and PicoTLS to provide a powerful API to develop and execute plugins on TLS.

PicoTLS [6] is a lightweight implementation of TLS 1.3 written in C. Whereas, uBPF is a virtual machine running on the user space that is able to execute eBPF [7] byte code. Hence, plugins are small programs written in eBPF that replace or

add new functions to the calling sequence of the main protocol. We create what we called insertion point in the code, an insertion point is a function in the code that serves as an anchor to inject plugins. Moreover, using a virtual machine gives us the ability to execute plugins on any platforms that implements the protocols. Therefore, it is possible to easily exchange plugins between users of the protocols.

To illustrate the possibilities of our flexible protocol, we propose a plugin that counter DNS fingerprinting attacks. Fingerprinting is a type of traffic analysis attack in which a passive adversary is able to retrieve the encrypted traffic of a user (e.g. ISP, Government...) and infer the visited website. Using machine or deep learning, the adversary is able to extract the signature of the visited website based on unique features such as the length or the time interval between packets. In the case of DNS fingerprinting attacks, the adversary gather DNS over TLS [2] or DNS over HTTPS [8] traces.

We propose a plugin that adds padding and shapes the TLS connection to hide features, thus reducing the accuracy of fingerprinting attacks. To evaluate our plugin, we confront our plugin against state of the art DNS fingerprinting attacks.

Structure of the thesis:

This thesis is organized as follows: ¹

- **Chapter 2:** In this chapter, we explain the handshake protocol used to negotiate extensions, cryptographic parameters and authenticate the server. Then, we describe the structure of the TLS record and how encryption is performed to protect records. Finally, we present PicoTLS, an implementation of TLS we used as a base to implement our flexible protocol.
- **Chapter 3:** The uBPF virtual machine gives us the ability to execute eBPF byte code. In this chapter, we explain the advantages of using uBPF to execute plugins. Then, to understand how to use the virtual machine we explain the core features of the VM and how to write eBPF program in C.
- **Chapter 4:** The main goal of this thesis is to implement a modular version of TLS with the help of the uBPF virtual machine. This chapter describes the implementation of the protocol and what issues we had to resolve. The uBPF virtual machine is very limited in the sense that it only allow to execute small independent programs. Therefore, we had to extend the virtual machine to provide additional features. Mainly, we had to provide way for plugins to interact with the main process and each other.
- **Chapter 5:** In order to demonstrate the ability to develop meaningful plugins for TLS, we have implemented a defense plugin to counter fingerprinting attacks. In this chapter, we explain the reasoning behind fingerprinting attacks and establish a state of the art of existing defenses. Then, we describe how we designed a plugin based on an already existing defense named BuFLO to add padding and shape to a TLS connection. Finally, we describe the experimental setup we used to gather a dataset of defended traffic, that we confront against a state of the art DNS fingerprinting attack to evaluate the effectiveness of the defense.

¹All the code used for this thesis can be found on a repository at <https://github.com/antoinerime/Thesis-tls>

Chapter 2

TLS

Formerly named SSL, TLS for Transport Layer Security is a widely known protocol used to secure a connection between two peers through encryption. The first version of SSL was published in 1995, since then 6 versions were published. The most up to date being TLS 1.3. As the security is a quickly evolving world, four of the six versions are now deprecated because they contain too many vulnerabilities. The goal of this thesis is to study the benefit of adding the eBPF virtual machine on `tls` to execute plugins. This way we are more flexible and can correct vulnerabilities much quicker. Security for a client is no more dependent of the `tls` implementation of the server (i.e. some server implements deprecated version of `tls` or use vulnerable cipher suite) as the client could propose plugin to the server to fix the vulnerabilities and vice versa.

TLS is independent of the overlying protocol. As such, many application protocol like *HTTP* for web browsing, *SMTP* for mail exchange or *SSL-VPN* use TLS to add a layer of security on the connection. Most of the time TLS is used over TCP connection to transport packets, however the protocol is compatible with any transport protocol as long as it is a reliable, in order data-stream.

TLS ensures the security of a connection by respecting the following properties:

- **Privacy** TLS uses symmetric encryption to ensure the privacy of the exchanged data. The shared secret used to encrypt the data is determined by both peers during the handshake protocol. Therefore, the data sent over the channel is only readable by the two peers. However, TLS does not hide the length of the data sent unless padding is configured. Thus, it is possible to perform traffic analysis on the traffic between two peers to recover information.
- **Authenticity** The server is always authenticated in a TLS connection, ensuring the identity of at least one peer. It is possible for the server to ask for the authentication of the client but it rarely happens in reality. Authentication

can be done using asymmetric cryptography algorithms like RSA or by using a symmetric pre-shared key.

- **Integrity** Data integrity mechanism are used in a TLS record to ensure that no alteration are made by an attacker without detection.

As TLS follows the client-server model, a connection is always initiated by the client to start the handshake. The purpose of the handshake is to establish shared keys for the symmetric encryption, negotiate cipher suites, modes and parameters. Furthermore, TLS authenticates one or both side with the help of certificates. Once the handshake is over, data can be safely sent with TLS record using the cipher suites defined previously.

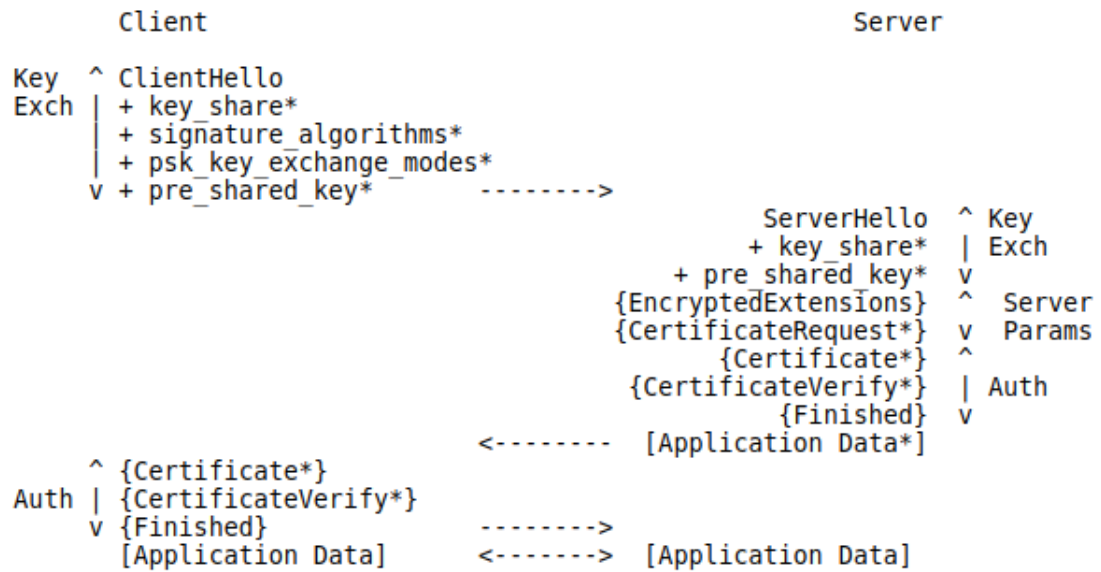
2.1 Handshake

The handshake is the first phase of a TLS connection. It is used to negotiate the security parameters of a connection. It can be divided into 3 or 4 steps depending on the version. In this section, we will present the process of a TLS handshake and point out the main differences between version 2 and 3. If an error or failure happen during the handshake phase, the connection must be terminated

Figure 2.1 from rfc8446 [9] presents the overview of the handshake for TLS 1.3.

During the KeyExch step, the connection will establish the shared secret and negotiate the cryptographic parameters (i.e. Key exchange, encryption and hash algorithms). It begins with a client sending a *ClientHello* message containing:

- **Legacy_version** This field is used to negotiate the TLS version. The client must indicate the highest version of TLS it supports. However, some implementations don't implement correctly version negotiation inducing "version intolerance" in which a server rejects a ClientHello indicating a version number higher than it supports. In TLS 1.3 to avoid this error the version is set to TLS 1.2 and an extension named "supported_versions" is used to negotiate the version of the protocol.
- **random** A random number (nonce) is generated by the client and servers. This number will be used as a seed in the generation of the cryptographic key. It will also be used for the authentication phase since it avoids replay attacks as the nonce can only be used once by definition. The nonce is the combination of the current timestamp and a pseudo random number.



- **+**: Indicates extensions
- *****: Indicates optional messages
- **{ }** & **[]**: Indicates encrypted messages

Figure 2.1: TLS 1.3 Handshake process.

- **cipher_suites** A list of supported cipher suites in order of preference. A cipher suite is a set of algorithms containing one key exchange, one encryption and one hash computation algorithm (e.g. TLS_AES_256_GCM_SHA384). These suites are defined in the rfc8446 [9]. It is not possible to choose a combination of algorithms that is not proposed in the rfc. This way, we ensure that a user choose algorithms for the connection that are compatible with each other.
- **extensions** In TLS 1.2 this field was meant to add extensions to eventually improve the protocol. In version 3, several extensions are mandatory and were put in this field for backward compatibility. For example "supported_versions" is indicated to tell the server we support TLS 1.3. In the extensions field is also indicated the key_share used to determine the symmetric key, allowing the server to compute the symmetric key and sending back data upon receiving the ClientHello.

The server will process the ClientHello and choose a cipher suite among those

proposed. What's proceeding next will depend on the version used.

In version 3, the server sends the key share in the ServerHello's extensions field. As the key share, necessary to compute the symmetric key, has been sent in the ClientHello, the server knows the key and can start encrypting data. Otherwise, the fields in the ServerHello are similar to the one in ClientHello. This ends the first step of the handshake meaning that all the following packets will thus be encrypted.

The next step consists of establishing additional parameters and extensions for the connection. A message named encrypted extensions is sent by the server containing all the additional extensions for the connection, eventually the server can ask a certificate for the client. Many extensions exist in TLS, but they will not be described here, they can be found in details on the rfc[9].

The last step is meant to authenticate the server and conclude the handshake by providing key confirmation and handshake integrity. In this last step the server sends three messages, Certificate, CertificateVerify and Finished.

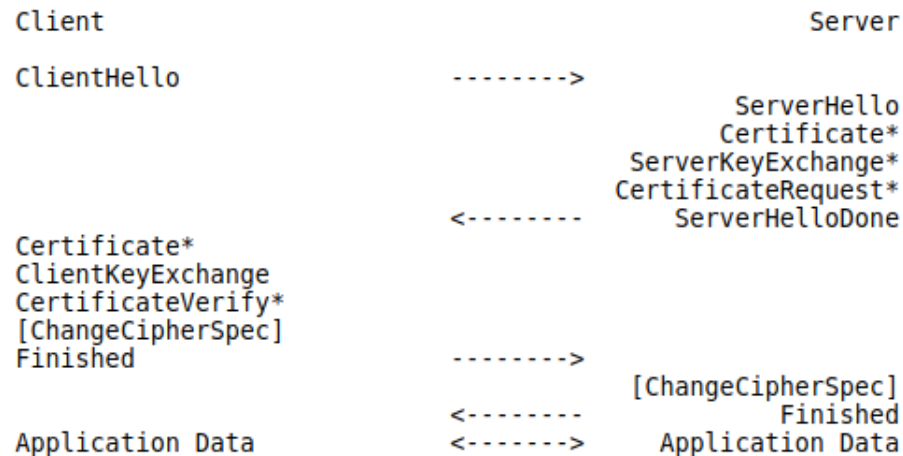
- The certificate message contains a chain of certificate to validate the public key of the server.
- The CertificateVerify which gives the proof that the server owns the private key associated with the public key in the certificate sent previously. The hash that is signed by the server is called the transcript hash. It is a concatenation of the hash of all the previous message exchanged during the handshake. By signing this hash the server guarantees his identity and the integrity of the handshake as the hash contains all the information that were previously exchanged between the two peers.
- The Finished message concludes the handshake for the server, this message contains a HMAC of the finished key which is itself derived from base key using key derivation function (HKDF) and transcript hash of messages sent during the authentication step (Handshake context, Certificate and CertificateVerify). This last message ends the handshake for the client. After the finished message it is possible to send early-data to the client however there is no guarantee that it will be accepted by the client as the handshake has not been validated on the client side.

On the client side, all the extensions and parameters are processed. Then if the server asked for a certificate the client responds with his chain of certificate and a CertificateVerify the same way the server does. Then the client validates the handshake by sending a Finished message.

In version 2, the main difference is that the key share used to compute the symmetric key is not present in the ClientHello. It means that the server has to initiate the

key exchange process once the suite has been established. Therefore, extensions and the certificate are sent in clear as the symmetric key is not known at this step. Once the server has sent the ServerHello, the Certificate, the ServerKeyExchange and an eventual CertificateRequest, it notifies the client that this step is done by sending a ServerHelloDone.

Figure 2.2 from rfc5246 [10] presents an overview of the handshake in TLS 1.2



- *: Indicate optional messages
- []: Indicate encrypted messages

Figure 2.2: TLS 1.2 Handshake process[9].

Upon receiving the ServerKeyExchange the client generates and sends the ClientKeyExchange and is able to compute the base key for symmetric encryption. Once the base key has been established, the client must send a ChangeCipherSpec to the server to notify that the base key has been generated on his side. Finally, the client can conclude the handshake by sending a Finished Message.

When the server receives the ClientKeyExchange it can compute the base key and then send the ChangeCipherSpec and the Finished message to end the handshake phase.

From this point the server can start sending data, the client needs to receive the Finished message before being able to send data as well.

2.1.1 Key exchange

In TLS there are two different mechanisms to compute the base shared key for symmetric encryption.

- Static RSA; in this key exchange technique, once the client has confirmed the identity of the server with the certificate, he can generate a 48 bytes long secret. Then, he encrypt it with the public key of the server and send it. The server can then decrypt the secret using his private key. With this technique the server does not need to send a ServerKeyExchange however there is a big flaw with static RSA. It does not provide Perfect Forward Secrecy, this is why it has been dropped in TLS 1.3. Perfect Forward Secrecy is a property that guarantees the secrecy of previous message if the private key of the server has been discovered. Let's assume that an attacker is able to store all the messages exchanged between the client and the server. If at one point he obtains the private key. Then, if the symmetric key has been obtained using Static RSA, the attacker will be able to decrypt the message containing the 48 bytes secret and derive the secret key to decrypt the rest of the exchange [11].
- Authenticated Diffie Hellman is a key exchange that guarantees PFS because a part of the information needed to compute the secret is not sent during the exchange. Making it impossible for an attacker to derive the shared secret even with the private key. This technique uses the Diffie Hellman algorithm or some variants like elliptic curve Diffie Hellman. With this technique both the client and the server have to send their Diffie Hellman parameters with the KeyExchange or KeyShare message depending on the version. To avoid man in the middle attack the parameters sent by the server are signed with his private key. Then the secret is computed for each side using the Diffie Hellman algorithm.

2.2 TLS Record

The TLS record is the main structure to convey data between two peers during a TLS connection. There are four types of records: handshake, application_data, alert and change_cipher_spec. The record protocol takes the message to be transmitted from TLS or higher-layer protocols, eventually fragments it if the size is higher than the max size defined (2^{14} bytes), encrypts it using the cipher suite and the shared secret negotiated during the handshake then computes the MAC to ensure the integrity of the data being sent.

Once the cryptographic suite and the shared secret have been defined, handshake,

application_data and alert message are encrypted in the record. The structure of the record is the following: 2.3. The TLS record is made of two parts, an encrypted

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

Figure 2.3: TLS Record.

part (i.e. TLSInnerPlainText) and a header in clear (i.e. TLSCipherText).

The TLSInnerPlainText is composed of three fields:

- **content**, is the data that needs to be transmitted by the higher-layer protocol. It is possible to send record with empty content to counter traffic analysis.
- **type**, a record can have three different types (i.e. handshake, application_data and alert) unlike the opaque_type that is always set to 23 to lure middle box. This type field defines the type of data expected in the content field
- **zeros**, in order to inflate the size of the packet to counter traffic analysis it is possible to add padding to the encrypted part of the record. Padding is added by appending an array of zero character after the type.

The TLSCipherText is composed of four fields:

- **opaque_type** This field is a constant field set to 23, corresponding to the application_data type. To avoid unintended behavior induced by middle boxes all the encrypted record are defined as application_data, the real information about the content type is stored in the encrypted part and is thus unreadable for middle boxes.
- **legacy_record_version** in the same way as the ClientHello and ServerHello to avoid version_intolerance errors the field indicating the TLS version is set to the value of version 2 even in version 3.

- length is the length of the encrypted_record, it is the sum of the content, the type, the eventual padding and additional data added by the AEAD algorithm like the MAC to ensure integrity.
- encrypted_record is the encrypted TLSInnerPlaintext.

2.2.1 Padding

Implementation of TLS can decide to add arbitrary padding to a record or send empty record during a connection between two peers. The goal is to obfuscate the traffic to prevent an observer from gathering information about the connection. By either hiding the real length of data being sent or by sending empty packet to hide the absence of activity during a connection. More details about padding models can be found in the plugin chapter[5] as we studied the attacks and defense known for website fingerprinting in order to create a plugin to counter website fingerprinting.

The distinction between data and padding is done with the type field. When a peer receives a record he first decrypts it and verifies the integrity. Then he can scan bytes from the end to the beginning of the TLSInnerPlaintext until he finds a non-zero value. This value corresponds to the type field, everything above the type field is therefore data for the higher layer protocol.

2.3 Encryption and AEAD

In TLS several methods exists to encrypt data depending on the cipher suite chosen during the handshake. Choosing the right suite for a given situation can be a difficult task as they have various properties: Capabilities, difficulty of implementation, cost of the encryption, known vulnerabilities [12]... Studying all the suites is outside the scope of this thesis and some recommendation for the best suites to use exists [13].

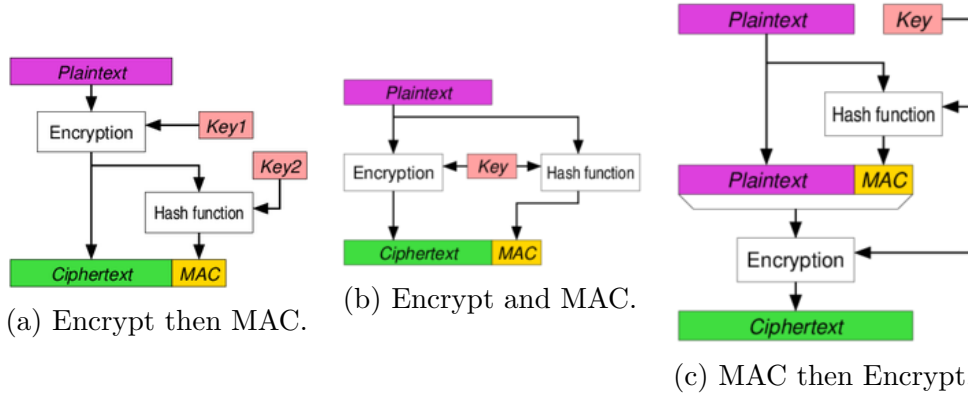
Three different schemes of encryption exist among the existing suites in TLS

- Encryption mode: This is the simplest and oldest mode, with encryption we guarantee the confidentiality of the data transmitted. It means that an attacker that intercepts the cipher text cannot get any information about the original plain text.
- Authenticated encryption mode (AE): With this mode we add an additional property that guarantees the integrity of the function. It means that if the cipher text is being altered by an attacker, it will be detected once received by the algorithm. Authentication of the data is done by producing a MAC of the data with the help of a cryptographic hash function.

- Authenticated encryption mode with additional data (AEAD): With this mode we add the ability to authenticate additional data that is generally not encrypted (e.g. the header of the record in TLS). This prevents an attacker from altering the header of a TLS record.

AE and AEAD combine already existing encryption algorithms with cryptographic hash to provide additional properties to a suite, however there exist various way to combine them:

- Encrypt then MAC: With this composition the plain text is encrypted first then the MAC is computed over the cipher text and appended to the cipher text.
- Encrypt and MAC: With this composition the MAC is computed over the plain text, the plain text is encrypted then both are appended
- Mac then Encrypt: With this composition the MAC is computed over the plain text then appended to it. Finally, the encryption is done over the concatenation of the plain text and the MAC.



The most common combination in TLS 1.2 is Mac then Encrypt, however it is not the best combination performance wise. Indeed, when a packet is received it needs to be decrypted first before the integrity can be checked which can lead to a waste of processing power if the packet has been altered. Security wise some vulnerabilities exist [14] [15].

Encrypt and Mac can lead to some security issue if the cryptographic hash algorithm used is not secured as it can directly give information about the plain text.

Therefore all the standardized cryptographic suites that are doing AEAD use Encrypt then MAC to provide the confidentiality and authenticity of the TLS record. Furthermore, in TLS 1.3 it has been decided that only AEAD suites are authorized.

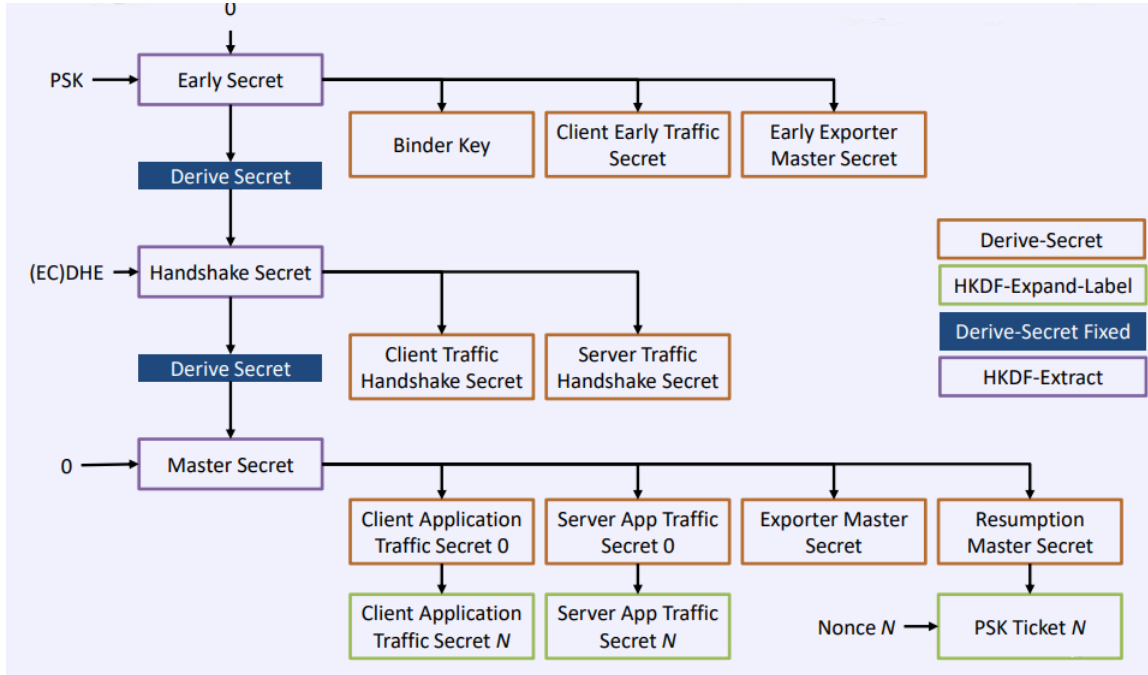


Figure 2.5: TLS 1.3 Symmetric key derivation[16].

2.4 Key derivation process

In this section, we explain how TLS 1.3 derives the keying material from the master secret to perform the encryption of packets. To ensure the security of transferred data, each record is encrypted/decrypted with a unique traffic secrets that is disposed once the record has been encrypted/decrypted. To obtain those traffic secrets TLS uses an algorithm named HKDF (i.e. HMAC-based key derivation function). HKDF implements two methods, Extract and Expand.

The HKDF-extract method creates a new pseudo random key from the input of previous input secrets and the handshake transcript. The input secret can be a pre shared keys, the shared secret derived from the (EC)DHE algorithm or the output of a previously called HKDF-extract. The handshake transcript contains nonce exchanged by the client and the server, thus ensuring that any handshake will have different traffic secrets.

The HKDF-expand creates additional keys from a previously generated key.

Figure 2.5 resumes the use of HKDF in a typical TLS connection. We begin by the early secret which can be use by the client to send early encrypted data when a pre-shared key has been established in advance.

If there is no early data to be sent we directly go to the second phase to compute

handshake secrets. The HKDF-extract take as input the shared secret derived from the (EC)DHE or the results of the first extract if the connection uses PSK instead of (EC)DHE to establish the shared secret between peers. From the handshake secret extract we get one server and one client traffic handshake secret. They are used to encrypt/decrypt, respectively, the second and third message of the handshake.

The last phase is to derive the traffic secret to encrypt/decrypt records for the remaining of the connection. We use the secret from the handshake secret as input to extract new keys. As for the handshake, one traffic secret is used for records from the server and the other one for records from the client. For every new record we use the HKDF-expand to derive a new secret.

In TLS version 2 we can only encrypt records. Therefore, the derivation of traffic secrets is much simpler, and we only need to compute the master secret from the shared secret, which is established during the handshake. From that master secret we derive new keys for each record similarly as TLS 1.3.

2.5 PicoTLS

The main goal of this thesis is to study the opportunity of adding an eBPF virtual machine to TLS to make it more modular. As such, we had to choose an implementation of TLS to add the virtual machine. PicoTLS [6] is a lightweight implementation of TLS 1.3. We chose this implementation because it implements TLS 1.3 which is the most up to date version of TLS and the code is light making it easier to add the eBPF virtual machine to the implementation, this way we were able to focus more on the creation of interesting plugin than the adding of the virtual machine.

PicoTLS take any data stream and sends it over a tls connection, it supports some TLS 1.3 functionality like PSK, PSK-DHE and connection resumption using 0-RTT. The main operations of PicoTLS are the following:

- `ptls_handshake` This function is responsible for performing the handshake for the client and the server, it consumes any input and eventually push some data to complete the handshake, as the connection must be initialized by the client in TLS, the first call to these functions must be done by the client with a zero sized input. Upon receiving data if the connection is still in handshake state (i.e. Finished message hasn't been received) the function is called. It first checks the integrity of the data and decrypt it if the cipher suite has been established. Then, it handles the message and answer with the appropriate message.
- `ptls_send` Every time new data need to be sent by the upper-layer applications

this function is called, the function first checks if the sequence number is lower than the max value. If not, as stated in the rfc, the encryption key must be renegotiated. Then, the function fragments the data if the size is higher than the max allowed size for a TLS record. Finally, based on the size it creates one or more record, encrypts it and computes the MAC for the all record.

- `ptls_receive` Every time new data is received this function is called, it starts by checking the integrity of the data and decrypting it. Then it removes the eventual padding by deleting all the zero bytes value present in the record from the end to the beginning until it encounters the type field.

PicoTLS maintains a context structure to communicate to all the functions the present state of the connection. We will see later that this structure will be helpful to manage plugins and communicate between eBPF plugins and the main implementation.

Chapter 3

eBPF

The execution of plugins inside the TLS protocol is made possible with the help of a virtual machine. In our case, the eBPF virtual machine. Executing a plugin on a virtual machine instead of the native machine provides several advantages. First of all, the generated byte code for the plugin is platform independent. Hence, the byte code can be executed on any machine as long as it implements the virtual machine. The most well-known example to illustrate this propriety is the Java Virtual Machine. Indeed, one of the main goal of java when it was designed was to be able to execute compiled code on any machine. Java code is compiled into Java byte code. The byte code is then executed by the JVM. The byte code generated by the compiler is the same regardless of the underlying architecture. Hence, making it platform independent.

Plugins may harm the main TLS protocol by doing unauthorized access to memory or by implementing infinite loop. Using a virtual machine gives us the ability to check the behavior of the plugin before executing it. Thus, we can avoid harmful plugin to be executed and to crash the main protocol.

In this chapter, we explain what are eBPF and uBPF. We also describe the main features of uBPF and how to use them to write and execute eBPF byte code.

3.1 What's eBPF?

eBPF [7] is a virtual machine implemented in the Linux kernel with a RISC instruction set. Historically the Berkeley Packet Filter (BPF) was used to capture and filter packets on a network interface. Filters were programs run by the virtual machine at the kernel level. Later, the machine was extended to become eBPF. The extension changed registers from 32-bit to 64-bit. It also added registers, from two to ten. Finally, it introduced the instruction *BPF_CALL* allowing the virtual

machine to call external functions.

3.2 uBPF

uBPF [17] is simply an implementation of eBPF virtual machine running in the user space. The TLS protocol run on the user space of a machine. Therefore, we need to be able to execute eBPF byte code from and in the user space.

The uBPF virtual machine implements several functions:

- **ubpf_create/ubpf_destroy**: instantiate/delete a new instance of a virtual machine.
- **ubpf_register**: register external functions that can be call with the *BPF_CALL* instruction.
- **ubpf_load**: load eBPF byte code into the VM.
- **ubpf_compile**: compile eBPF byte code into native machine code.
- **ubpf_exec**: execute the byte code.

3.2.1 Helper function

In order to call external libraries and other useful functions. It is possible to register external functions into the virtual machine with *ubpf_register*. Calling an external function in the virtual machine is done with the *CALL* instruction. External functions are executed by the host machine. Therefore, external functions are not restricted by the virtual machine and can access memory of the main process.

Registering a new function is done by calling the *ubpf_register* function. This function takes as argument the virtual machine in which we register the function, an id, the name and the address of the function to register. Each VM maintains an array with all the registered function.

When an uBPF program wants to call an external function, it uses the *CALL* instruction. That instruction takes as immediate the index of the function to call. During the compilation of eBPF byte code, the compiler cannot know in advance the index of the registered function. Therefore, the immediate field for the call instruction is set to -1. Then, during the load phase the virtual machine replaces the immediate field with the correct index for the external function.

Helper functions add more flexibility when developing uBPF programs. First of all, it gives the ability to execute functions from libraries without having to

re-implement them in eBPF. Furthermore, it allows uBPF programs to execute function with higher privilege. Indeed, as external function are executed by the host machine they are not limited by the environment of the virtual machine that prevents access to memory out of the uBPF stacks.

3.2.2 Loading eBPF byte code

A program executed by the virtual machine may be harmful for the host machine. This is especially true for eBPF that runs at the kernel level, but still applies to uBPF. Therefore, to reduce security and stability risks the `ubpf_load` functions performs several checks before executing the program. If any of the test fails, the `ubpf_load` function will return an error.

The virtual machine verifies that the program ends and that there is no infinite loop. Furthermore, uBPF checks that for instructions that may crash the program. For example, for every call to an external function, uBPF check that the function has been previously registered. It also checks that instruction are valid eBPF instruction and that there is no division by zero. For a complete list of all the checks you can refer to the `validate` function in the uBPF code [17].

3.2.3 Executing eBPF byte code

The uBPF virtual machine is a vm with a 64-bit RISC instruction set. To execute a program we provide to uBPF a file containing eBPF byte code. Which is simply a list of eBPF instructions.

eBPF provides the following type of instruction:

- ALU instructions to perform arithmetic operations.
- Byteswap instructions to perform operations on bits.
- Branch operations to manage loops and conditions.
- Memory instructions to interact with memory.

An exhaustive list of eBPF instruction can be found on the documentation [18]. To execute a new program the virtual machine allocates a stack to store variable during the executions. A pointer to a list of arguments is stored in the first register of the machine.

During the execution the virtual machine performs runtime checks on the program. Runtime checks are useful to monitor memory access done by the VM. To prevent corruption of the memory of the main process. An uBPF virtual machine is only allowed to access memory from its own stack. Thus, we improve the security of the

main process by preventing the VM from doing illegal write on the memory which may cause crash. Instead, when the VM try to access unauthorized memory, the access is catch by the runtime checks which immediately stop the VM.

3.2.4 Writing uBPF code

The first step to use the uBPF virtual machine is to write eBPF byte code to be executed. However, writing machine code is a bit cumbersome. Hopefully for us, the Clang compiler [19] provides an option to compile C code into eBPF byte code. Therefore, plugins are written in C then compiled into byte code for the virtual machine.

The written C code must follow a number of rules to be accepted by the virtual machine.

First of all, there can only be one function that defines the entire code to execute. Introducing the concept of function calls into the virtual machine would add complexity that is not desired as the purpose of the VM is to execute small code and not complex programs. Indeed, calling a new function requires to extend the stack to execute the function. It also means that we need additional memory space to store the code of functions. Therefore, there is no such concept as function calls for the virtual machine.

Hopefully, to improve readability of the code, some tricks exists to define additional functions.

The first one is to define macros as they are not additional functions and will be replaced in the code by the preprocessor.

The second trick is to implement functions with the following signature:

```
1 static __attribute__((always_inline)) void example(int a);
```

Always inline force the compiler to inline [20] the function even if optimizations are disabled. If a function is declared inline, the compiler makes calls to that function faster. To do so, it integrates the code of the function into the code of its callers. Thus avoiding the call operation and the preparation of the stack that are usually done when calling functions.

Additionally, when a function is defined as both static and inline, if the function's address is never used. The compiler will not output the code for that function.

In other words, using this signature produce the same output as directly writing the core of the example function into the code of the main function. Thus reproducing the behavior of a macro.

Furthermore, the function can only have one argument. This argument can be a

pointer to an array to pass multiple arguments to the function.

As there is no function calls it is not possible to directly call usual libc functions like `printf` or string functions. As this is a bit restrictive, uBPF implements a way to call external functions (see Section 3.2.1). Therefore, to use functions from the libc or any other libraries, it is first required to register them as external function into the virtual machine.

Finally, it is not possible to deference pointers outside of the memory area of the machine (i.e. the stack), nor it is possible to implement infinite loops as the VM will not execute them (see Section 3.2.3).

3.2.5 JIT

Executing code into the virtual machine induce some overhead. Therefore, to increase the performance of executed code and limit overhead, uBPF implements a feature called JIT. It uses the *ubpf_compile* function to compile eBPF byte code into native code. In other words, uBPF translate each eBPF instruction into its equivalent for the host machine instruction set. For now, only **x86_64** is supported.

Compiled code is no more executed by the virtual machine as it is not eBPF byte code. Instead, it is the host machine that will execute the code. Hence, reducing the overhead inherent with the usage of virtual machine.

Users need to be more careful when using JIT to execute programs. Indeed, as the compiled code is run natively. Runtime checks that are usually done when executing eBPF byte code with the virtual machine are no more possible. Hence, the program has access to the memory of the main process as any other function. Therefore, it is necessary to be more cautious as programs run in JIT mode are able to corrupt memory.

Chapter 4

Implementation of plugins with uBPF

In this chapter we describe how we combined PicoTLS and the uBPF virtual machine to implement a more flexible version of TLS 1.3. Indeed, we used the uBPF virtual machine to execute eBPF byte code during the execution of the main process. By doing so, we increase the ability for developers to add features to a TLS connection according to their need.

We begin by identifying the main functions of the TLS implementation which we called protocol operations. A protocol operation is a key element in the sequence of events in a TLS connection that we may want to alter with a plugin. For example, the operation *push_encrypted_records* that take some data and encrypt it in a new record. Each of those operations is an anchor to insert plugins into the main process. It means that we want to be able to execute uBPF code just before or after the execution of an operation or even replace the execution with uBPF code. Once again with the example of the operation *push_encrypted_records*, we may want to benchmark the time needed to encrypt a packet. To do so, we would implement a plugin that gets the time just before and after the operation in order to compute the time needed to encrypt data.

As we've seen in the chapter 3, the uBPF virtual machine gives us the ability to execute code written in eBPF. To give more flexibility on the implementation and ease the usage of plugins we need to implement more features that will enhance the virtual machine.

To begin with, the execution of eBPF code by the virtual machine asks many steps. It must first load the eBPF code, create a new instance of the virtual machine, register helper functions, insert the byte code into the machine and finally execute the code. In the case of multiple executions of the same byte code, reiterate all those

steps before executing the byte code is a redundant and costly process. Therefore, we do them once when the plugin is registered and then, store the instance of the virtual machine for future executions.

Furthermore, plugins need to be able to interact with the main process of TLS to change its behavior. To do so, we had to implement an API to manipulate the main structures of PicoTLS. This API gives us the opportunity to easily modify the variables of PicoTLS in a secure manner.

Finally, a plugin is a combination of one or more piece of uBPF code that we insert in the main code and that can be executed multiple times. Those pieces of eBPF code, which we refer to as pluglets for the rest of this chapter, need to be able to communicate in between multiple calls. Indeed, the uBPF machine, when executing a pluglet, only allocates a stack to perform its operations. As for any other functions, the stack is used to store all the local variables of the function. Therefore, the memory of the stack is non-persistent between calls of the same or different pluglets. Thus, we implemented a heap shared among all pluglets of the same plugin to store plugin's data in between calls. Furthermore, we implemented a set of custom lightweight functions to manage and interact with the plugin's heap.

4.1 Insertion Point

The first step to add plugins to PicoTLS is to create insertion points. An insertion point is a function in the code that serves as an anchor to inject plugins. To do so, we create an overlay that will handle the execution of the native function and eventual pluglet to be executed.

That overlay is what we called protocol operation 4.1, the protocol operation is a structure containing the following elements:

- **id**: The identifier of the protocol operation, it will be used to find the structure in the hash table when calling a specific protocol operation.
- **core function**: It is the native PicoTLS function that is executed when there is no plugin to execute.
- **replace**: If this field is not empty, it means that a pluglet as been loaded to be executed instead of the core function.

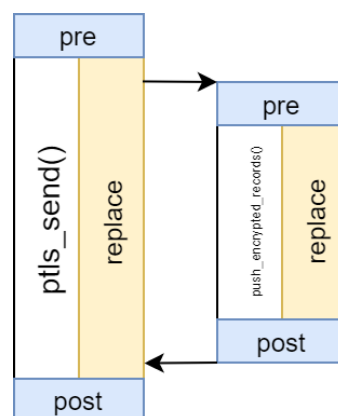


Figure 4.1: Protocol operation.

- **pre:** If this field is not empty, it means that one or more pluglets have been loaded to be executed before the core function or replace pluglet. Pre is a list of pluglet, they are executed sequentially.
- **post:** As for the pre field, post is a list of pluglet to be executed after the core function or replace pluglet.

Every call to a core function is replaced in the main code by a call to the *run_plugin_proto_op_internal* function, this function will search for the corresponding protocol operation structure. Based on the information provided it will execute the correct code.

4.1.1 Passing arguments

Two different protocol operations may have a different number of arguments, also pluglets may not need all the arguments of the core function. To make the *run_plugin_proto_op_internal* more flexible and to avoid confusion when writing plugin. We decided to store all the arguments of a function in the *ptls_t* data structure. This object is used in the main implementation of PicoTLS to keep track of the state of the TLS connection between function calls. We add a new field to this context that contains additional arguments for protocol operation. Arguments are stored in an array of pointer to be easily retrieved by the core function or the pluglet.

Therefore, the signature of a core function called by the protocol operation overlay will look like this:

```
1 proto_op_arg_t ptls_send(ptls_t *tls);
```

Only the context argument is passed to functions. Additional arguments can be retrieved at the beginning of the function as they are stored in the TLS structure. This feature also provides more flexibility when coding new plugins. As we see in Section 4.5, pluglets are able to call other protocol operation. Therefore, it is possible for a pluglet to directly call another pluglet via the uses of protocol operations. In this situation, we can imagine a pluglet A calling another pluglet B of the same plugin. Only this time, with additional arguments than what is usually provided when pluglet B is called by the core function of the protocol operation associated with A.

4.2 Registering protocol operation and plugins

To be able to execute core functions or plugins we first need to register them into the protocol operation structure. The registering of core functions is done at the beginning of the program and is quite straightforward.

We associate the protocol operation's id with the core function by creating a new protocol operation structure, then we register the operation by adding it into a hash table that store all the protocol operations.

Registering plugins is done once all the core functions have been registered.

In order to give information about the structure of a plugin to the PicoTLS implementation, we provide with each plugin a manifest file. This manifest is structured the following way;

First line always indicates the name of the plugin. Then each following lines indicate a pluglet. A pluglet is characterized by the following elements:

- **protocol operation ID:** ID of the protocol operation on which we plug the pluglet.
- **plugin operation:** Can either be pre, post or replace depending on the behavior defined for the pluglet.
- **eBPF byte code path:** Path of the eBPF byte code for the pluglet.

When given the manifest, our implementation start the registration process for each pluglet.

To begin with, we search and get the protocol operation from the hash table. If it does not exist, it means that the plugin defines a new operation. In this case we simply create a new operation that will not have a core function from the PicoTLS implementation associated.

Creating a new operation is only available for replace pluglet. The goal of pre and post pluglets is to work in conjunction with the core function or replace pluglet. For example by logging the state of the connection before or after a protocol operation. Therefore, it is pointless to create a new operation with a pre or post pluglet without any core function associated. Adding a pluglet as a new operation serves the same purpose as adding new function when designing a code (i.e. re-usability and abstraction).

The following step is to load the code into memory and create the new virtual machine instance to execute the pluglet.

We begin by creating a new VM. The uBPF can only store one piece of byte code at a time. In order to avoid reloading the code each time, we decided to associate for each pluglet his own VM. Thus, we save time when executing a pluglet by not

having to redo the loading steps.

Then we load the helper functions 4.5 into the VM and finally we load the byte code into the VM. As explained in the Section 3, the VM performs various checks during the loading of the code, like the presence of an exit code or the correctness of the calls to external functions. This ensures that when executed, the pluglet will not crash our main program.

If any error happens during the registering process of a pluglet we cancel the registration and delete all the elements of the plugin. Thus, we avoid any unexpected behavior with the executions of plugins.

In the end, we provide a very simple way to create and use new plugin. To do so, a user need to write the code in C, compile it in eBPF byte code with the help of the Clang compiler, then write in the manifest how the pluglets behave and where they need to be inserted in the main code. Once it's done, the user specifies when executing the main code the location of the manifest.

4.3 Memory management

As we've seen in previous sections pluglets are independent of one another as they all are executed in their own instance of the virtual machine which itself does not have persistent memory. It raises two major issues when designing plugin.

First of all, we cannot store and re uses variable between multiple calls of the same pluglet. Indeed, to properly execute a pluglet we provide a stack to the virtual machine to store the local variables. By definition a stack is non persistent and therefore, does not keep the states of local variables between multiple calls.

Furthermore, two pluglets of the same plugin may need to communicate between them. The uBPF virtual machine was designed to simply allow the execution of eBPF byte code on a machine in the user space. Therefore, there are no features that allow multiples instance of a VM to communicate with each other.

In that regard, we extended the virtual machine by adding persistent memory in the form of a heap.

The heap we provide is a fixed size heap that is shared among all the pluglet of the same plugin. It means that we create as many heaps as they are plugins loaded in PicoTLS process. The plugin's heap is a data structure that is itself stored in the core process's heap, which makes it persistent among executions of pluglets.

By providing a heap to plugins, we are able to store and reuses variables between multiple calls and also between calls of different pluglets of a plugin.

4.3.1 Interacting with the heap

Now that we have a heap for plugins, we must implement a way for pluglets to interact with the heap memory. Indeed, as stated in the previous section, the plugin's heap is stored into the main heap of the process. Therefore, the virtual machine cannot directly access the plugin's heap as the VM is not authorized to access the main memory, see Chapter 5. Therefore, we need to implement helper functions for the pluglets to interact with the heap.

There are a few issues we need to resolve for the pluglet to be able to interact with the plugin's heap.

First of all, the plugin's heap in its current state is just raw memory. To use it as a heap, we need to implement functions to allocate and free memory into the plugin's heap (i.e. malloc and free).

Furthermore, as the pluglet can not directly write or read data stored in the plugin's heap. We need to implement functions to copy data from the plugin's heap to the stack of the virtual machine. So that the pluglet is able to interact with data from the plugin's heap.

Finally, even though data can be stored in the plugin's heap. Pluglets still need the address in memory to access the data. As explained above, there is no way for virtual machines to communicate between each other. It means that when a pluglet allocate memory in the plugin's heap we need to store the returned address and the size of the allocated memory into the main process in order to be able to retrieve and uses the address for future execution of the pluglet.

4.3.2 Heap structure

In order to limit the overhead of using the plugin's heap we decided to implement a lightweight version of malloc and free. The implementation is inspired from the C++ implementation of B.Kenwright[21].

In this section we begin by describing the structure of the heap, how memory blocks are allocated. Then we describe the implementation of the malloc and free functions.

To reduce the complexity and to simplify the implementation of the malloc and free function, the plugin's heap memory is split into a fix number of constant sized blocks. A LIFO linked list is maintained to keep track of the available free blocks. We store the following metadata for each heap:

```
1 typedef struct memory_pool {  
2     uint64_t num_of_blocks;  
3     uint64_t size_of_each_block;  
4     uint64_t num_free_blocks;  
5     uint64_t num_initialized;
```

```

6     uint8_t *mem_start;
7     uint8_t *next;
8 }memory_pool_t;

```

At the beginning, the number of initialized block is zero. When a block is allocated, it becomes and stays initialized for the remaining time of the execution.

The first bytes of an initialized block can either be a magic number, indicating that the block is allocated or an index number, indicating that it's free.

You may notice that there are actually no elements that correspond to a linked list in the metadata. Indeed, the next field is a simple pointer that indicates the address of the first available blocks. To avoid the overhead of dealing with a real linked list that requires the frequent uses of malloc and free to push and pop elements. This implementation uses the first byte of data from memory blocks 4.2 to store the index of the next element of our linked list, in this case a free block. Therefore, the index number simply indicates the index of the next free block of the heap.

Figure 4.2 provides an example of a heap with four blocks, on this example

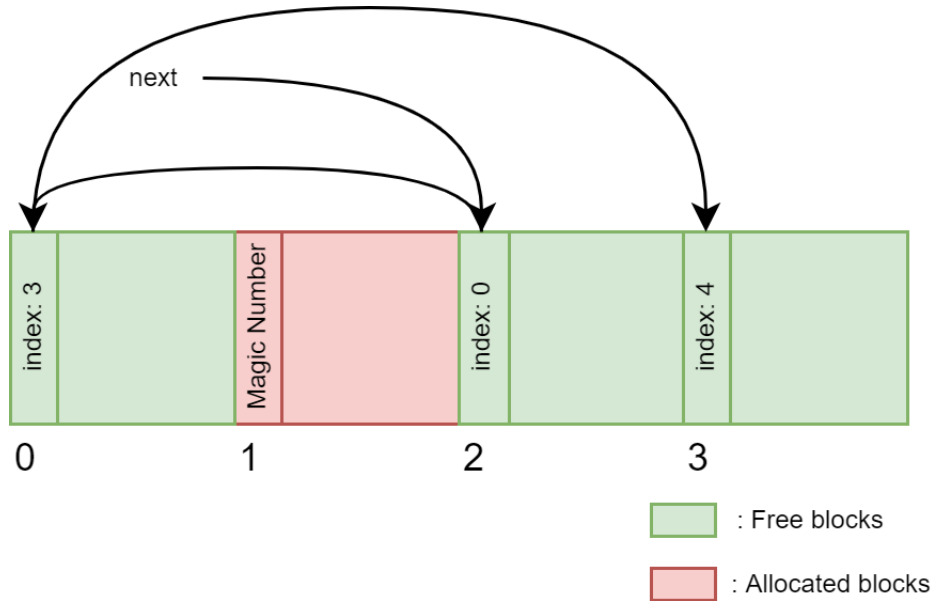


Figure 4.2: Structure of a 4 blocks heap.

only the second block is allocated and next is pointing towards the third block. This heap structure gives us the ability to implement malloc and free functions with a complexity of $\mathcal{O}(1)$. Indeed, when called on our example heap, the malloc function first checks if there is any free block remaining in the heap. As it is the

case in our example, malloc simply allocates the block pointed by next. Then it modifies the value of next to point to the next element in the list. As explained earlier, the next element in the list is provided by the index at the beginning of the block, in this example, the first block.

When a block is freed, the function checks if the value of the magic number is correct to detect any potential corruption of the memory. Then it replaces the magic number by the index of the block pointed by next, index 2 in our example. Finally, it modifies next to point to the newly freed block.

4.3.3 Read and write on the heap

The uBPF virtual machine is not allowed to read or write in the memory of the main process. As the plugin's heap is stored in the memory of the main process, the only way for a pluglet to access the plugin's heap is to use helper functions. Therefore, we implemented memory functions (i.e. memcpy and memset) for the pluglet to read and write on the plugin's heap.

With the help of memory functions, pluglets can now copy a local version of the data stored in the heap on their stack, modify it and then write it back on the heap.

However, as memory functions are executed by the main process, as any other helper function, the use of those functions raise security issues. Indeed, the intended use of those functions is for a pluglet to write and read from the plugin's heap, but nothing prevents the pluglet to use the memcpy to access other parts of the main process memory. This defeat the purpose of using a VM with no access to the main memory. Thus, we need to limit a pluglet memory access to its associated plugin's heap.

By providing a context to the memory functions, they can know what is the plugins's heap associated with the calling pluglet. By knowing the plugin's heap, memory functions check whether or not, the memory area that need to be read or written is allowed for the pluglet.

This way we ensure that a pluglet can only use memory functions to read or write data in its own plugin's heap, and does not corrupt the memory of the main process in any unintended way.

4.3.4 Pointer management

The last step to be able to use the plugin's heap is to provide a way to store addresses of allocated data in between calls of pluglets. Indeed, we provided with the plugin's heap persistent memory to store data. Even though data are stored, when a pluglet is executed it needs the address of the data to access it. Therefore,

we need to store and manage pointers of allocated blocks from the plugin's heap in between pluglet calls.

To do so, we once again use context objects to store pointer for plugins. Our goal is to store, in an array, pointers returned by our custom malloc. When a pluglet needs to store or retrieve data from the plugin's heap, it looks into this array to get the correct pointer associated with the data it is looking for.

To ease the management of pointers on the plugin side, we implemented the following helper function.

```
1 void *get_opaque_data(ptls_context_t *cnx, opaque_id_t op_id, size_t
    size, int *allocate);
```

To allocate a new block of memory or to retrieve the pointer of an already allocated block, a pluglet can call the *get_opaque_data* function. When called, the function will check if the array is empty at the index pointed by the *op_id* parameter. If it is the case, it calls malloc to allocate a new block into the plugin's heap, then store the pointer into the array. Finally, it returns the pointer for the pluglet. If the array is not empty, the function simply returns the pointer.

On the pluglet side, it means that we don't actually have to worry about the allocation on the heap. For each data stored into the plugin's heap, we associate an id. That id is used to retrieve the correct pointer in between calls.

4.4 Interacting with picoTLS

The plugin's heap described in the previous section gives the ability for pluglets of the same plugin to interact with each other. However, we still don't have a way to interact with the main process itself. The main goal of adding plugin to TLS is to be able to observe or modify the behavior of the main implementation. It is therefore mandatory for pluglets to interact with the main PicoTLS process. In this section we describe how a plugin can interact with the main process of picoTLS. Changing the behavior of the TLS connection can be done in two different ways. The first one is to modify the functions calling sequence of the main process by adding new calls in between the original sequence. Thus changing the results of the overall sequence.

For the second one, PicoTLS maintains a data structure (i.e. *ptls_t*) that stores information about the connection (e.g. Current state, Cryptographic suite, Cryptographic key, etc. ...). By modifying elements stored in the *ptls_t* structure we can alter the overall behavior of the TLS process.

It is possible for a pluglet to execute functions of the main process with the help of helper functions (see Chapter 3). Adding PicoTLS functions to the list of

helper functions is quite straightforward. To do so, we need to register them when creating a new virtual machine for the pluglet as for any other helper functions. Additionally to call basic functions, a pluglet may also need to call protocol operations. Calling protocol operation can be done by calling a specific function that will prepare all the arguments and call the correct function (see Section 4.1). Therefore, in addition to basic function, we also register a helper function to prepare and run protocol operations.

The *ptls_t* data structure is passed as an argument to every function of PicoTLS, including protocol operation. Therefore, pluglet also have access to the structure. However, as the structure is stored into the main memory pluglets are not able to read or write data on it. As we did for the plugin's heap we must allow pluglets to interact with the structure. One way to do it would be to extend memory functions described in Section 4.3.3 to allow read and write on the *ptls_t* data structure. To achieve this, we would need to know the area of memory that covers the structure. This is not trivial as the structure contains itself pointers towards other area of the memory. Furthermore, allowing pluglets to directly writes on the structure may lead to a crash of the main system if the pluglet writes incorrect data. Thus, we decided to use another method to access the *ptls_t* data structure, which is via the use of a get/set API.

With an API, we can control with more precision which fields in the structure a plugin is allowed to access. It also prevents crashes of the main process in case of incorrect data as the get/set API can sanitize input before writing them into the structure.

4.5 Helper functions and Protocol operations

In this section, we make a list of protocol operations and helper functions that are available to create new plugins. We begin with a non exhaustive list of helpers functions that describe the different categories of helper functions we have access to when implementing plugins.

- **get/set API:** List of functions to write and read fields of data structures that are maintained by the main process of PicoTLS.
- **get_opaque_data:** Function used to retrieve pointers of data allocated in the plugin's heap (see Section 4.3.4).
- **Memory functions:** Set of memory functions to read and write data from the plugin's heap.

- **run_plugin_proto_op_internal:** Function used to call other protocol operation
- **PicoTLS functions:** A set of useful functions from the main implementation of PicoTLS. In this set we find function to encrypt or decrypt data, send TLS alert, manipulate buffers...
- **Library function:** To ease the implementation of plugins, we added functions from various C libraries (e.g. time, ifaddrs, stdlib).

We follow by a list of protocol operations and a description of their usage.

- **Handle connection:** This operation is called right after the TCP connection has been established. It is responsible for managing the TLS connection from the beginning to the end.
- **Ptls send:** This operation is called each time new data need to be sent.
- **Ptls receive:** This operation is called when new data is received on the socket.
- **Push encrypted record:** This operation creates a new TLS record and encrypts it from the input's data.
- **Select file descriptor:** This operation selects if a new message is available on the receive socket or the input file.

The list of helper functions and protocol operations was mainly built for the need of the padding plugin described in Section 5. Therefore, it is not complete, and we may need to add in the future new elements to improve the capabilities of implementing plugin, thus making the protocol even more modular.

Ideally, on a larger scale we should define a standardized list of helper and protocol operation that would give enough flexibility to implement any plugins we could think of. This way, we would ensure that plugins can be developed independently of the underlying TLS implementation and that plugins are compatible with all the implementations of the defined standard.

4.6 Conclusion

We were able to extend PicoTLS by implementing every feature described in this chapter.¹

We began by developing the protocol operation framework. It adds an overlay on the execution of key functions in the protocol. This framework is the base that gives us the ability to create insertion point and choose between executing core function or pluglets.

To handle the registration of new plugins, we implemented a function that reads a manifest that contains all the information about a new plugin. Then, this function prepare a new virtual machine by reading the byte code indicated by the manifest. As helper functions and the byte code has already been loaded, the resulting virtual machine is ready to be executed by the protocol operation frameworks.

To enable pluglets to store data into a persistent memory. We implemented a heap for each plugin. As this heap is stored into the main process area, pluglets can store and retrieve data in between calls. To handle the heap, we implemented malloc and free function to respectively allocate and free new blocks of memory. Additionally, we implement memory functions to write or read data from the heap.

Finally we implemented a get/set API to easily read variable from the core process. This API gives the ability to control with a fine granularity what variables of the protocol are available for pluglets. This also gives the ability to implement sensitization of input when setting new variables.

With this extension of PicoTLS, developer are able to create complex plugins to customize and improve the behavior of the TLS protocol.

4.7 Future works - Exchanging plugins

As for now, the only way to register a plugin into the main process is to ask PicoTLS to load and register the plugin at the beginning of the connection. TLS being a Client/Server protocol, it would be interesting to allow the protocol to exchange plugins with peers.

In this section we discuss how we could modify the protocol to allow the exchange of plugins between a client and a server. Implementing such a mechanism requires thinking about how the two peers negotiate plugins and how do they exchange them. The latter part would be quite straightforward.

¹Code is available at <https://github.com/antoinerime/Thesis-tls>

To exchange plugins, we could imagine a new record that would be sent just after the handshake phase. In this record, we would find the manifest that indicates the structure of the plugins, then we would send the byte code for each pluglet. Once the peer has received the manifest and the byte code it can register the plugin the same way it would do it for a local plugin.

However the negotiation is more complex and requires a trade off between security and flexibility. TLS already possess a mechanism to negotiate extensions between the client and the server. A first lead would be to use this mechanism to also negotiate the usage of plugins. As for any other extensions, the client would propose to the server a list of plugins, the server will then respond with a truncated list containing the accepted plugins. This solution may be a good start as it works quite well to negotiate and exchange plugins known by the server and the client. However, it means that the server must know the plugins to be able to accept it. Therefore, defeating the purpose of using plugins to add new features to a protocol before having to wait for a new standard. For obvious security reasons, we cannot make the server accept any new plugin proposed by the client. A trade off could be for the server to ask a third party if the plugin is safe or not. In this case we could imagine one or many third party organisms that would maintain lists of plugins where:

- The server would be subscribed to those lists and would only accept new plugins validated by the third party.
- The client would download new plugins from the list.
- Developers would propose new plugins to add to the list.

Other negotiation mechanisms may be thought of, but in the end, it will always be a trade off between security and flexibility.

Chapter 5

Traffic shaping plugin

By using TLS to ensure the privacy, authenticity and integrity of exchanged data users can safely navigate to websites and exchange confidential data with servers. However, protocols evolve through time. It has been shown on several occasions that the need and expectations for a protocol can deeply change a few years after it has been standardized. Implementing a new standard can take times. For TLS, new features may be wanted by users to improve their privacy. Indeed, even though data is safely exchanged between a user and a website, a passive observer may link the user to the website visited.

Usually, a user directly accesses a website using his browser, in this case the visited website can be easily deduced by an observer on the network. By looking at the destination address of the IP header, the observer can perform a reverse DNS query with the address to obtain the visited website. In some situations, a user may want to hide from observers websites he is trying to access. An observer is any entity that is able to capture traffic on the path between a user and a website. It can be someone on the same wifi network, an internet provider, government...

Hopefully, several methods exist for a user to hide the website he is trying to access. He can use the Tor network, a vpn or contact any other proxy. Those three methods have in common that the user will no more directly send a packet to a website, but instead contact an intermediary node that will then forward traffic to other nodes or directly to the website. Hence, the observer can no longer deduce the visited website by checking the destination ip address as this address will always be that of the intermediary node. Furthermore, such mechanism must make sure that an observer cannot associate the outgoing traffic from the intermediate node with a user. Most of the time as many users use the intermediate node at the same time it is very hard to deduce what outgoing traffic is linked with which user.

In recent years, new attacks have emerged using the processing power of machine learning and deep learning algorithm to deduce the visited website without the help of the IP destination address. Those attacks known as website fingerprinting

[22][23][24] extract the signature of a website by looking at the size and timing features of packets exchanged while navigating on a website. Therefore, making the methods described above inefficient to hide visited websites.

In addition to website fingerprinting, a similar attack focused on DNS over TLS[2] and DNS over HTTPS[8] named DNS fingerprinting[25][26] has also emerged. The principle stays the same as for website fingerprinting, the only difference is that we extract information from DNS traffic instead of web traffic.

In order to evaluate the usage of our modular TLS implementation and at the same time contribute to the improvement of security and privacy on the internet, we implemented a plugin that adds padding and shape a TLS connection based on a padding scheme named BuFLO[23] (Buffer Fix Length ObfuscaTor) that was proposed to counter fingerprinting attacks. Then, to evaluate the efficiency of our plugin we used our implementation of TLS to do DoT. We gathered a dataset of protected DoT trace using our plugin that we evaluated against known DNS fingerprinting attacks [25].

In this chapter, we begin by explaining how it is possible with machine learning or deep learning to extract the signature of a website from a set of packet captures. Then, we establish a state of the art of the defense scheme that exists to counter website fingerprinting attacks. We briefly describe how they work, their advantages and flaws.

In the second part we describe how we implemented the BuFLO defense on a TLS connection using the uBPF virtual machine.

In the later part, we explain the architecture implemented to do DoT and how we gathered our dataset. We also, describe what classifier we used to evaluate our defense. Finally, we present the results of our defense against known attacks and discuss the strengths and flaws with using such a defense.

5.1 Website fingerprinting

Website fingerprinting is an attack that allows an adversary to identify the visited website of a user, even if he uses a defending mechanism such as an encrypting proxy. Indeed, even though proxy hides the final destination, it does not hide the size, direction and timing of received packets nor does it hide the duration of the connection. Those features, unique for each website, give the ability for an adversary to accurately deduce the visited website. Dyer, et al.[23] showed, in their paper published in 2012, an attack that could infer the visited website with an accuracy of 80%. To counter WF attacks, defenses try to hide features described above by adding fake packet or padding onto the connections. Another strategy consists of luring the attacker by imitating the signature of other websites.

5.1.1 WF attack

To perform a website fingerprinting attack an adversary must be able to retrieve all packets exchanged during the loading of a web page between the server and the client. The adversary is passive, therefore, he cannot alter the sequence or contents of packets in any way. This also means that the adversary is hard to detect.

The adversary uses a classifier build upon machine learning or deep learning algorithm to infer which trace corresponds to which website. To begin with, he must gather a training set for the classifier. To do so, the adversary gathers packet sequences from websites of interest and feed them to the classifier. This way, the classifier is able to extract a signature from each website and will be able to recognize them in the future.

To work efficiently a classifier must be able to find similarity between traces of the same site and differences between different sites. Therefore, to characterize a trace the classifier computes a set of features. Features vary from one classifier to another. Depending on the defense the classifier try to defeat, it could adapt to focus on the information the defense does not hide. Despite this, Cai, et al.[27] propose four major features we can extract from a trace that are independent of one another; unique packet length, occurrence of unique packet length, inter-packets timing and ordering.

Cai, et al. represent a packet sequence as:

$$P = \langle (t_1, l_1), (t_2, l_2), \dots, (t_n, l_n) \rangle$$

where t_i represent the timing interval between packets i and $i - 1$, with $t_0 = 0$. The size of packet i is represented by l_i . The direction of a packet is indicated with the sign of l_i , positive for outgoing and negative for incoming packets.

The first feature to consider is the **unique packet length**. Packet length is a quite simple way to differentiate websites. Indeed, resources sent by a website server vary from one site to another. In the other hand, the size of a resource stay constant in between access to the same website. Hence, if a website A send a resource in which the size B is unique compare to other websites. Then any packet sequence that contains a packet of size B must be from A. Servers always try to send data at the maximum transmission unit. Therefore, the length of a resource is always given by the size of the last packet.

Two sequences of packets P and P' have unique packet length iff:

$$(\exists L \in P | L \notin P') \vee (\exists L \in P' | L \notin P)$$

However, it is very easy to implement defenses that hide this feature by simply adding padding to always send constant size packets (e.g. Tor cells).

The second feature is the **Packet length frequency**. This feature highlights the duration of an exchange between a user and a website. As we see in Section 5.1.2, most of the defenses are not able to hide this feature. Indeed, to hide the duration of the exchange we need to send dummy padded packet to reach a fix loading duration for every website. In order to cover all website, even the slower to load, we must choose a duration long enough. Therefore, this would induce a large overhead for fast loading site as they will continue to send packet to reach the loading time of the slowest site.

We define the packet length frequencies as the number of packets of the same size in the sequence that are not unique. In other words, we count the number of MTU sized packet in the sequence.

Let's suppose $n_L(P)$ the number of packets of size L in P , we say that P and P' have a different packet length frequencies iff:

$$\exists L | n_L(P) \neq n_L(P') \wedge n_L(P) > 0 \wedge n_L(P') > 0$$

The third feature is the **Packet ordering**. The loading of a web page follows a client/server models in which the client sends multiples GET to the server to retrieve all the resources needed to load the page. Every web page follows a logic, after the initial GET the client receive the index of the page. Based on the reference on the index page, the client will then send GET to get all the remaining resources. Hence, it is possible to identify the logical order of each website in order to differentiate them.

Let's suppose M_l as the multiset of packet lengths in P_l , without ordering. We say that P and P' have different packet ordering iff:

$$M_l = M'_l \wedge P_l \neq P'_l$$

However, packet ordering is not always accurate as ordering may vary between two calls of the same sites. Indeed, networking conditions such as delay, bandwidth or jitter may unpredictably change the order of packets.

Finally, the last feature is the **Interpacket Timing**. This feature measures the timing between a packet and his successor. With this, a classifier can link a GET packet with its answers. Indeed, the timing between two linked packets should always be of one RTT. Therefore, if two packets have a timing less than the RTT measured it must mean that there are independent.

Let's suppose two sequence P and P' with length $|P|$ and $|P'|$. We say that P and P' have different interpacket timing iff:

$$\exists i, 1 \leq i \leq \min(|P|, |P'|) : (t_i) \neq (t'_i)$$

The same way as for packet ordering, network conditions may perturb this feature as the RTT is not constant across multiple access to the website.

To evaluate the effectiveness of a classifier, two experiments named closed world and open world experiments can be found in the literature.

In the closed world experiment, the victim visits n sites a constant number of times. The attacker knows in advance the n sites. Therefore, he can train his classifier with all the sites. The attacker must guess to which site correspond which packet sequence.

The Closed world experiment has been criticized for being unrealistic as it is quite possible for a victim to visit sites unknown to the attacker. Therefore, they propose a new experiment called open world. In this experiment, the attacker only knows a subset of sites visited by the victim. Thus, the decision for the classifier is harder because it must be able to tell that a website is unknown, if the site is not present in its training set.

Even with the open world experiment, the majority of assumptions that are made for the gathering of the dataset are in favor of the attacker [28].

To begin with, the victim follows a specific behavior when browsing websites. Most of the time, the victim access website sequentially, one at a time. Thus, it means that the victim only has a single tab in its browser and can not load multiple page at the same. This is a bit unrealistic as it is common to have multiple tabs open on a browser.

Furthermore, there is no background traffic during the packet capture that may prevent the attacker from correctly identifying websites. It is common to have background traffic from other applications than the web browser. Being able to separate the web traffic from the background traffic may be a non-trivial task for the attacker.

We assume during the capture of packets that the attacker knows exactly when the traffic begins and when it ends. In the case of background traffic or multi-tab browsing. Packets can continuously go through the network interface. Making it much harder for an attacker to know when to start and end the capture.

Finally, we assume that the attackers can gather his training set in the exact same condition as the victim. Thus, having the same operating system, network connection, browser parameters, etc... Having the exact same conditions is almost impossible and should lower the accuracy of the attacks, even though some recent classifier are able to adapt.

5.1.2 WF defenses

To counter website fingerprinting attacks, some defenses have been designed. The goal of a defense is to attempt to hide features described in the previous section[5.1.1]. To do so, we can add padding to packets to hide the real size of exchanged data, send dummy empty packets or delay packets. Unfortunately, every defense comes with a price and can induce overhead and/or delay on the connection. Therefore, a good defense must be efficient against known attacks while taking into account the cost on the connection.

In this section, we survey the state of the art of the various defenses that have been proposed to counter fingerprinting attacks.

A defense can be deterministic or probabilistic. In a deterministic defense, if the defense is applied on the same input, it will always produce the same output. On the other hand, probabilistic defenses rely on a pseudo random generator number to alter the initial input. Hence, making it harder for a classifier to recognize specific features on a packet sequence as they are all different.

To this day, three defenses have shown to be effective against the more recent classifiers; **BuFLO**, **WTF-PAD** and **Walkie-Talkie**

BuFLO, for Buffer Fixed Length Obfuscator is a deterministic defense proposed by Dyer, et al. [23]. The idea of BuFLO is quite simple, packets are sent at a fixed rate with a fixed size for a fixed amount of time. When there is no data to send, dummy padded packets are sent instead. Hence, we can hide unique packet length, packet ordering, interpacket timing and in some situations, packet length frequency and prevent side channel analysis. However, the use of BuFLO comes with a high price as it introduces a large overhead and some delay. The defense is characterized by three parameters; d, p and τ .

- d : is the size of a fixed-length packet.
- p : is the rate at which we send packet.
- τ : is the amount of time for which we send dummy packet.

Based on the parameters, the defense will produce more or less overhead at the cost of delay and efficiency of the solution. Indeed, one feature that can be leaked by BuFLO is the duration of the connection. If the client/server still exchange data after τ seconds the defense is no longer effective. This situation can occur if the loading of the web page is longer than τ without defense or, if the bandwidth allowed by the combination of p and d is too low for the website. A low bandwidth will introduce delay, thus making the loading longer than τ .

Two improvements have been since proposed, CS-BUFLO [29] and Tamaraw [27]. BuFLO may introduce congestion if the connection of a user is not able to transmit $\frac{d}{p}$ Bytes of data. CS-BuFLO analyse the bandwidth between the client and the server and adapt periodically d and p to avoid congestion.

Tamraw attempts to fix the problem of the connection duration being leaked if it's longer than τ . In Tamaraw, instead of sending padded packet for a fixed amount of time, we send a fixed number of padded packets proportional to the parameter L . Let's suppose that a user should receive X packets from a server with BuFLO defense. If $AL < X \leq (A+1)L$, then the server will send additional padded packet to reach $(A+1)L$ packets. With L large enough, we suppose that enough sites will send the same number of packets to prevent the leak of the total duration of the loading of a site. Furthermore, it reduces the overhead as the parameter gives more flexibility regarding the overall duration of the exchange. Instead of always sending X or more packets with BuFLO. With Tamaraw, we send a multiple of Y packets, where Y is much smaller than X .

WTF-PAD, for Website Traffic Fingerprinting Protection with Adaptive Defense is a probabilistic defense proposed by Juarez, et al. [30]. Focus on hiding burst traffic during the loading of a web page. To do so, they fill gaps between burst with dummy padded packets following the distribution of generic web traffic. To begin with the defense must construct two histograms H_B and H_G . The first histograms contain the distribution of usual delay between the end of a burst and the beginning of another burst based on a large sample of website trace. In the other hand, H_G contain the distribution of usual delay between packets of the same burst.

The defense is implemented as a symmetric state machine for both peers. Figure 5.1 describes the state machine on the client side. The machine begins in the idle state. When the first message is sent, the machine switch to the **Burst** state. In the burst state the machine picks a timer t from H_B . If a new packet is pushed by the overlying protocol and sent during the timer, the machine stays in the burst state and pick a new timer from H_B . If the timer expires before a new packet is pushed and sent, the machine sends a dummy padded packet and switch to the gape state. If the machine picks an infinite time from the histograms it goes back to the idle state and stop sending padding.

In the **Gape** state, the machine picks a timer t from H_G . If the timer expires before a new packet is pushed then send by the browser, the machine sends a dummy packet, stays in the gap state and pick a new timer from H_G . If a new packet is pushed by the browser and send or if the machine pick an infinite timer from the histogram, it switches back to the burst state.

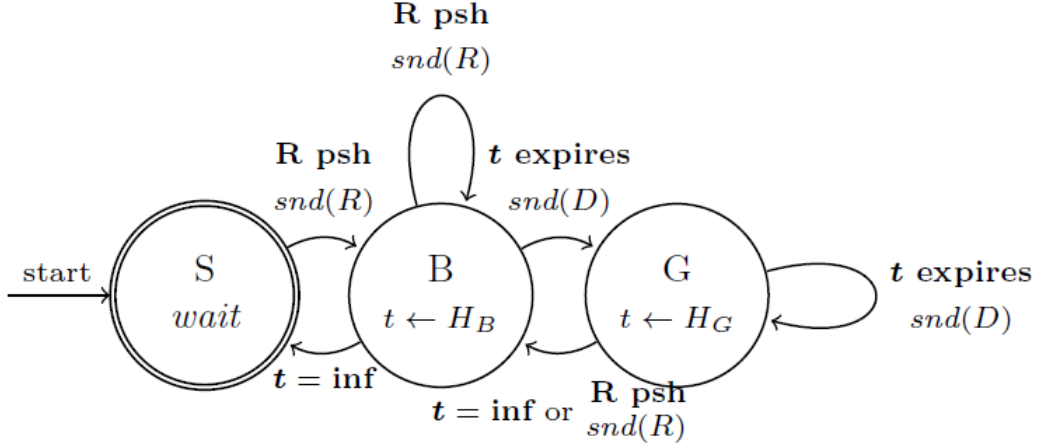


Figure 5.1: WTF-PAD state machine [30].

WTF-PAD generate much less overhead than BuFLO. Furthermore, it does not introduce delays as data pushed by the browser is immediately sent on the connection. However, it has been shown in recent classifiers that it does not perform well compared to other defenses. Rahman, et al. [31] were able to obtain an accuracy of 93.5% in a closed world experiments.

Walkie-Talkie is a deterministic defense proposed by Wang and Goldberg [32]. This defense is based on two mechanisms. The first one is that the browser communicates with the website in half-duplex mode. Usually browsers communicate in full-duplex mode, it means that they can send multiple requests to servers at the same time. In half-duplex mode, once the browser makes a request to get resources from the server, it must wait the answer of the server before being able to send a new request, hence the name walkie-talkie. The packet sequence can therefore be broken down into a sequence of bursts between the browser and the server.

The second mechanism consists of constructing a supersequence of the sequence of packets from the visited website and a second website. A sequence S' is a supersequence of S if $S \subseteq S'$. Let's suppose that the burst sequence for the visited site is $b_i = (b_{i+}, b_{i-})$ and for the second website is $b'_i = (b'_{i+}, b'_{i-})$ where b_{i+} is the number of sent packets and b_{i-} the number of received packets. Then, the supersequence is the maximum of packets between the two sequences: $B_i = (\max(b_{i+}, b'_{i+}), \max(b_{i-}, b'_{i-}))$. By constructing a supersequence of two sites, an attacker knows that the observed sequence belongs to one of the two sites, but is not able to deduce to which one exactly. Therefore, to use this attack, we first

need to identify sensitive and non sensitive websites. Then, when browsing a sensible site, we need to construct a supersequence with the burst sequence of a non sensitive site. On the other hand, when browsing a non-sensitive site we do the opposite a construct the supersequence with a sensitive site.

Walkie-talkie generates a low overhead, but may introduce some delay due to the half-duplex mode. Unlike other defenses, Walkie-talkie does not try to hide the visited website by making the trace indistinguishable from other sites. Instead of focusing on making packet sequences from all websites similar it focuses on making the sequence of only two similar websites. Therefore, this defense is designed to counter ISP or government that would block sensitive websites using website fingerprinting.

5.2 Padding plugin

In order to evaluate our ability to develop new plugins with the uBPF virtual machine, we decided to implement a plugin that pad and shape a TLS connection. Defenses presented in Section 5.1.2 are designed to protect against website fingerprinting. Therefore, they are implemented either at the Tor level for BuFLO and WTF-PAD or at the browser level for Walkie-talkie. By implementing a defense at the TLS level, we provide for multiple protocols that use TLS a defense against fingerprinting attacks. Indeed, it has been shown recently that DNS over TLS and DNS over HTTPS are also vulnerable against fingerprinting attacks [26][25]. Hence, the plugin we implemented can be used to protect access to a website or a DNS over a TLS connection against respectively Website and DNS fingerprinting.

The padding scheme used for our plugin takes inspiration from the already existing BuFLO defense. Indeed, BuFLO has proven to be an efficient defense against website fingerprinting attacks. Furthermore, the BuFLO plugin will serve as a working base for the development of other defenses.

The BuFLO defense sends packets of fixed size, at a constant rate during a fix amount of time. In order to implement the defense in PicoTLS we need to implement the following elements:

- Being able to pad packets at a given size using the padding feature in TLS records.
- Being able to send dummy padded packets at a given time.
- Being able to shape the connection to make sure that TLS record are sent at a constant rate.

Hence, our plugin is divided into four pluglets. The diagram in Figure 5.2 illustrates the calling order of pluglets during the execution. To begin with, *set_qdisc* and

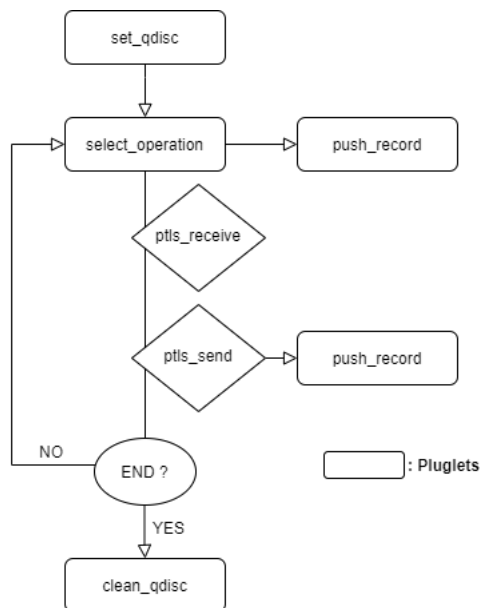


Figure 5.2: Execution diagram of pluglets.

clean_qdisc are respectively pre and post of the *handle_connection* protocol operation. Therefore, they are executed once, at the beginning, or at the end of the function. The purpose of those two pluglets is to set up and remove queuing discipline to shape the traffic (see Section 5.2.3). During the all duration of the connection, we loop over the *select_operation* pluglet. This pluglet is responsible for periodically sending dummy padded packets on the connection (see Section 5.2.2). The last pluglet replaces the *push_encrypted_records* to add padding to records.

5.2.1 Padding packets

The first element needed for the implementation of a defense is the ability to add padding to TLS record. Padding is inserted in TLS record in the form of an array of zero byte value following the type field in the record (see Section 2.2.1). To do so, we implement a pluglet that replaces the *push_encrypted_records* operation from PicoTLS. This operation takes as input a buffer of unencrypted data to send over the TLS connection. The operation encapsulates the data into a TLS record and encrypt it using the correct algorithm.

In addition to that, the pluglet add zero bytes in the record to add padding until it reaches the constant size that is defined in the parameters of the defense.

Hence, whenever a new TLS record is created the pluglet will be called and add padding in order to always send a record of constant size on the TLS connection.

5.2.2 Send dummy packets

In the BuFLO defense dummy packets must be sent periodically over the connection, as the implementation of PicoTLS is single threaded we can not create a task that waits and then send packets after p seconds as it would stall the entire process.

To deal with the sending and receiving of data on the TLS connection, `picotls` loops on a `select` that polls which file descriptors (i.e. sockets and input) are available for I/O operations. Based on the file descriptor, `picoTLS` calls the correct function to deal with retrieved data. For example, when a new record is available to read on the socket. `Select` returns the `fd` of the socket to notify the program that a new record is available, once the record has been read, `picoTLS` calls `ptls_receive` that will decrypt the record and extract data from it.

Therefore, to periodically send dummy padded packet we use the `select` operation that gives us the ability to wait for a certain amount of time with the `timeout` arguments without stalling the all process as `select` will return if there are any tasks that needs to be done.

If `select` returns before the end of the `timeout`, the function will update the value of `timeout` to reflect the amount of time not slept [33]. Thus, we implement a pluglet that replaces the `select` operation. The following pseudo-code illustrate the behavior of the pluglet:

```

1 defense_timer = get_defense_timer();
2 if state < PTLS_CLIENT_POST_HANDSHAKE
3 or defense_timer > PADDING_MAX_DURATION:
4     select(maxfd, readfds, writefds, timeout);
5 else:
6     timeout = get_timeout();
7     timeout -= get_time_of_previous_call();
8     select(maxfd, readfds, writefds, timeout);
9     if timeout == 0 && sendbuff.off == 0:
10         send_dummy_pkt();
11         timeout = BUFLO_RATE;
12         set_time_of_previous_call();
13         set_timeout(timeout);

```

We begin by getting the time elapsed since the beginning of the connection, in the BuFLO design we send dummy packets for a fixed amount of time. Therefore, if

the time elapsed is longer than the maximum duration we fall back to the normal behavior of the select operation. Likewise, during the handshake the defense is not active.

Otherwise, if the defense is active, we begin by getting the timeout value for select. If this is the first call and the value of timeout has not yet been initialized, the get function will set it by default to zero. Otherwise, the timeout is set to the value it was at the end of the previous call of the pluglet.

In between calls to the pluglet, time is running, but the timeout value is not updated accordingly. Therefore, we measure the time elapsed between two calls of the pluglet and subtract the value to the timeout. Hence, timeout accurately gives the elapsed time since the last dummy packet has been sent.

Select can return if the timeout reach zero or if one of the file descriptor is ready for I/O operations. In the first case, we check that the sending buffer is empty. Indeed, the goal of sending dummy packet is to fill the connection when there is no activity. A non-empty sending buffer means that there is some activity from the overlying application at the time. Therefore, it is not necessary to send a dummy packet as it would delay the sending of real data. If the sending buffer is empty, a dummy packet is sent over the connection and the timeout is reset to the rate value defined by BuFLO parameters for upcoming calls.

Otherwise, when select returns before the timeout reach zero, the new value of the timeout is registered for upcoming calls and the pluglet returns to let the main process handle file descriptors that have been flagged as ready by select.

Lack of precision may occur when sending dummy packets. For instance, let's imagine the situation in which select returns with a timeout value of $10\mu s$. As timeout is not zero, no dummy packet is sent and the pluglet returns. On the next call to the pluglet the elapsed time since the last call is $40\mu s$. In this situation, the dummy packet is late by $30\mu s$. However, we have to keep in mind that the order of magnitude of the BuFLO rate is about a hundreded of milliseconds and the average time between two calls of select operation is about ten microseconds. We think that this lack of precision will not lower the efficiency of the defense as it is not on the same scale and is therefore acceptable.

5.2.3 Shape the connection

The last part of the plugin is the ability to shape traffic. Indeed, with the help of the pluglet described in the subsection 5.2.2 we are able to send dummy padded packets at a constant rate. However, the overlying application may send bursts of data that exceed that constant rate. Therefore, to avoid leaking information,

we must shape the traffic. This way, we make sure that the outgoing traffic stays constant.

To shape the traffic, we use Linux traffic control to ensure that traffic goes out of the network interface at a constant rate. Traffic control implements queues, named queuing disciplines or qdisc, where network packets are stored before being transmitted. By default, there is only one qdisc named *pfifo_fast*. It's a first-in, first-out queue that does not limit traffic.

Otherwise, traffic control is composed of three components to handle network packets:

- **qdisc:** The queuing discipline defines the behavior of the queue, it can be classless (e.g. Stochastic Fairness Queuing, Token Bucket Filter) or classful (e.g. Hierarchical Token Bucket, Class Based Queuing). The difference is that classful qdiscs allow the use of filters to classify packets.
- **class:** A class is a sub-qdisc. Therefore, it must be linked to a queuing discipline. It defines attributes for the queue. For example the rate of an HTB queue. Packets are sent to a specific class based on the filter.
- **filter:** Filters are used to classify packet based on characteristics or metadata. A filter gives the ability to select to which qdisc/class a packet must be enqueued to.

Based on those components, a tree is built that defines the queues a packet must go through before being sent.

To send TLS traffic at a constant rate, we need to define an HTB queuing discipline. Then to make sure that the HTB is only consumed by the TLS connection, we create a class and a filter for each connection. Filters are based on the propriety of the connection (i.e. Src IP, Dst IP, Src Port, Dst Port) to send packets to the correct class.

Figure 5.3 illustrates the combination of queuing and classification modules produced by our pluglet. If the packet on the network stack match one of the filters it is queued in the associated class. If not, the packet goes to a default class with no limit on the sending rate.

As traffic control is implemented in the kernel we need a library to communicate from the user space to the kernel. To do so, we used the libnl libraries [34]. This library provides a framework to communicate with the kernel from the user space by using the Netlink protocol. The library provides similar commands to the TC utility program in Linux [35].

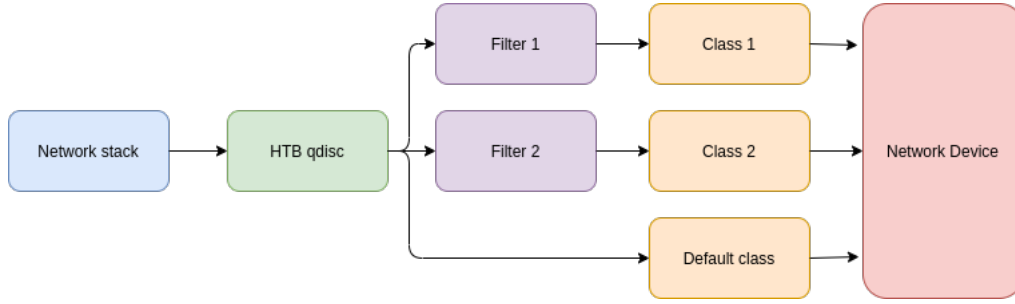


Figure 5.3: Traffic control tree for TLS.

5.3 Evaluation of the defense

The plugin we implemented in the previous sections gives us the ability to hide length and timing features by generating padded packets, and shaping the traffic to ensure a constant bandwidth. In this section, we begin by explaining the threat model and why we chose to focus on the defense of DoT with our plugin. Then, we describe how we gathered a dataset of DNS over TLS traces using our implementation of TLS. Next we describe classifiers we used to evaluate our defense. Finally, we present and discuss the results of our defense against known classifiers.

To evaluate our plugin we decided to focus on the defense of DNS over TLS. Even though BuFLO performs well to hide traffic features against an attacker, by using BuFLO on web connections we produce a large overhead compare to undefended traffic. This can produce congestion on the network and slow down the connection. Therefore, even though BuFLO has shown very good results it is considered to be too costly to be deployed on a large scale. On the other hand, using BuFLO on DoT may be worth considering. It has the same flaws as for web traffic. However, DNS traffic is much shorter in length and duration than web traffic. On average the size of a trace for DoT traffic is between 10 and 30 Kb. For a website the size of the trace may largely vary depending on the resources that are loaded by the client. As the size is short for DoT, the generated overhead is much smaller. Therefore, the cost of the defense on the connection of the client should be fairly low. On the other hand, the cost on the resolver may be an issue as the defense could congest the network on their side and provoke DDOS. To deploy this defense at a larger scale we would need to increase the resources of the resolver or coordinate the usage with the capacity of the resolver to only use the defense when the resolver is able to handle the load. Furthermore, as the packet size and the overall duration is much more predictable we can adapt the padding size and the duration of the BuFLO defense to be much closer to the real size and duration of the connection. Finally, compared to website fingerprinting, BuFLO defenses have

not been tested against DNS fingerprinting.

We consider a scenario in which a user uses DNS over TLS to safely contact a domain resolver. In this scenario an adversary is able to eavesdrop the traffic between the user and the DNS resolver and try to deduce which website the user is trying to access based on the DNS query.

5.3.1 Gathering the dataset

To evaluate the efficiency of our defense we had to gather a large sample of DoT trace. To do so, we need to access a set of predefined websites multiple times and capture packets for DoT traffic for each of the access. For our defense to work both the browser and the resolver must implement our defense. Even though firefox gives the ability to contact a resolver that supports DoT, the resolver does not implement our version of TLS with the BuFLO defense. Therefore, we must implement a DNS resolver that uses our TLS implementation and force the browser to use the resolver to resolve domain names. Hence, we designed two proxies to handle DNS queries over our TLS implementation.

The goal of those two proxies is to tunnel DNS queries over our version of TLS. At one hand, the entry proxy will receive DNS query from the browser. When this proxy receives a DNS query it opens a TLS connection with the second proxy and transmits the DNS query over that connection. The exit proxy role is to resolve queries it receives. To do so, this proxy sends queries in clear to a known resolver (e.g. Cloudflare). Figure 5.4 illustrate the architecture of the two proxies. To

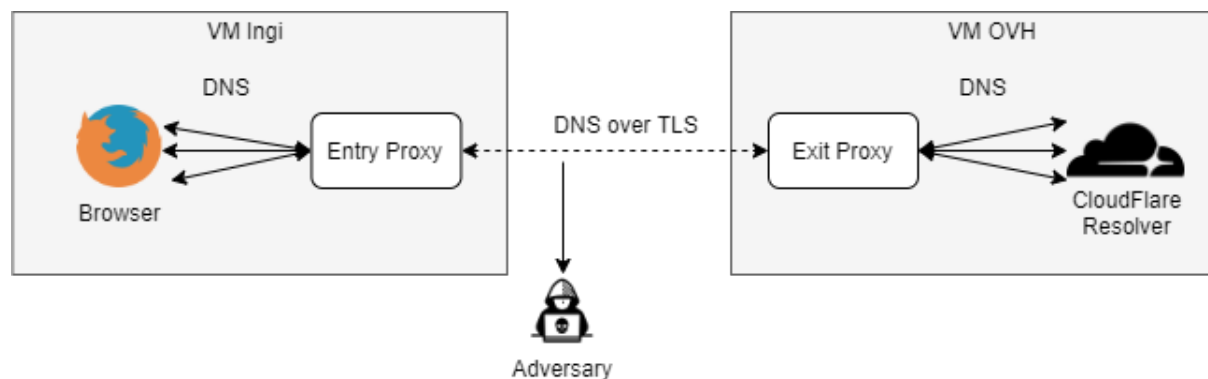


Figure 5.4: Proxies architecture.

make sure that the browser uses our DNS resolver we change the nameserver in *resolv.conf* to point to the entry proxy. In our setup the entry proxy runs on the same machine as the browser, therefore *resolv.conf* must indicate that the

resolver's address is 127.0.0.1.

The browser can send multiple DNS queries at the same time. Those requests are usually sent from different ports. However, in DoT DNS requests should all be pipeline over the same connection [2]. It means that when the entry proxy receives a DNS answer it must know to which ports it must send it back. To be able to associate each query with its answers we send additional metadata over the TLS connection.

When the entry proxy receives a DNS query from the server, in addition to the query itself, it sends the port from which it receives the query. Then, when the entry proxy receives the answer from the exit proxy, it contains the same port number. This way the exit proxy knows on which port he should send back the answer.

On the exit proxy side, each time a new port is received, the exit proxy will create a new connection with the DNS resolver. The proxy associates the port number with the connection. Therefore, when an answer is received from the resolver, the exit proxy sends this answer to the entry proxy with the associated port.

Hence, every DNS message that is sent over the TLS connection is encapsulated into an eight byte header with two fields; A four byte field indicating the port and a second four byte field indicating the length of the DNS message. Thanks to the length, if several messages are sent on the same TLS record. We are able to know when a message ends and when the second one begins.

With the help of the two proxies we can now simulate the behavior of DNS over TLS with and without our defense plugin. To gather the dataset we set up two virtual machines. The first one is located in the ingi cloud infrastructure and contains the browser and the entry proxy. To host the exit proxy, we rented a virtual machine on the OVH cloud. By having the two virtual machines on separate networks, we make sure that the DoT traffic is routed on the internet. Therefore, the gathered traces are subject to the variation of delay and bandwidth inherent to an internet connection. This way the gathered dataset is closer to a realistic scenario than if we did measurement on a local network.

To gather the dataset a python script run on the ingi virtual machine. With the help of selenium [36], the script visits a predefined list of websites based on Alexa top visited sites. While the site is loaded by selenium, the script capture the DNS over TLS traffic with the helps of tcpdump [37]. If any errors happens while the page is loaded or if we receive an HTTP response other than 200, the trace

is considered invalid and is deleted. We configure the Firefox profile to disable all caching mechanism. This way we make sure that the browser does the same operation for each sample of a website.

Websites are visited in a round-robin fashioned, for a total of 200 visited websites. Each website has been visited 40 times. The list is short compared to other study that takes samples of around 1000 sites [26][25][31]. The choice of sampling 200 sites was mainly due to the time constraint of presenting this thesis in June.

We gather one set of undefended TLS traffic, which is our control group to evaluate the efficiency of classifier and a second set of DoT trace defended with our implemented plugin.

1 2

5.3.2 DNS classifiers

The next step to evaluate our defense is to use classifiers on our dataset to test if the defense we implemented is efficient or not. Because of time constraint, we decided to search for already existing implementation of classifiers. In this section we briefly describe the features exploited by those classifiers. Indeed, we think that it is necessary to understand the features to be able understand and interpret the results of classifiers against our defense. We found two classifiers, one for DNS and the other for WEB fingerprinting. DNS fingerprinting is fairly recent, therefore it is difficult to find authors that published their implementation of the classifier that we could use to evaluate our defense. As you will see below, we found one implementation of a DNS classifier that gave us good results. However, this implementation only focuses on packet length and packet order which are an easy feature to hide in website fingerprinting. To provide a more complete evaluation, we also wanted to confront our defense against classifiers that exploit timing features. Unfortunately, we only found web classifiers that exploit timing features. Features that are evaluated between DNS and Web are fairly similar, therefore we hope that we can tweak the parameters of the web classifier to obtain good results against DNS.

The first classifier we used was proposed by Siby, et al. [26][38]. This classifier is designed to do DNS fingerprinting. Therefore, it uses a set of features to extract information from traces specifically designed for DNS traffic. The authors proposed features that consist of n-grams of packet length in the trace. A n-gram

¹ The complete code for the implementation of the proxies and the gathering of the dataset can be found on github: <https://github.com/antoinerime/Thesis-tls>

² The dataset is stored on a drive https://uclouvain-my.sharepoint.com/:f:/g/personal/arime_oasis_uclouvain_be/Ep-R15U5Z1JKtaxSU1v30vABoqA95IkoT2hrqH7QT-EZ8Q?e=lyVlVs.

is a tuple of n following packets. Let's suppose the following packet sequence: $(-64, 88, 33, -33)$. The corresponding unigrams are $(-64), (88), (33), (-33)$, bigrams are $(-64, 88), (88, 33), (33, -33)$ etc... Additionally, they represent consecutive packets with n -gram representation of traffic bursts. The idea is that bursts correlates with the number of resources load by the website. With our previous example, a burst length sequence would be: $(-64, 121, -33)$. The classification algorithms used is called Random Forests.

The second classifier we were able to use is designed for website fingerprinting, it was proposed by Rahman, et al. [31][39].

In this classifier, the authors proposed a set of timing features to extract information from packets. They propose eight timing features, three of them being focuses on the timing interval between packets of the same burst, while the other five focus on timing between bursts.

5.3.3 Results

We begin our evaluation with the DNS classifier, to feed the data to the classifier we need to create a json file with the following format:

```
N.pcap :
{
    sent: [],
    received: [],
    order: []
}
```

Where $N.pcap$ is the identifier of the website, *sent* is a list of packet sizes sent from the client, *received* is a list of packet sizes received by the client and *order* a list of $+1$ and -1 values indicating the order of packets.

We do a first classification with undefended DoT traffic and a second classification with defended DoT traffic. The parameters for the BuFLO defense are the following:

- $d = 1384$ bytes
- $p = 90$ ms
- $\tau = 20$ s

Results are summarized in Table 5.1.

First of all we can see that the classifier provides the expected behavior for undefended DoT. Siby, et al.[26] obtained an accuracy 0.904 for DNS traffic, this

DNS Classifier	Accuracy	Recall	Websites	Samples
Undefended DoT	0.9767 $\pm$ 0.009	0.976 $\pm$ 0.009	200	40
Defended DoT	0.1473 $\pm$ 0.014	0.16 $\pm$ 0.013	200	40

Table 5.1: DNS classifier accuracy.

difference can be easily explained by the number of evaluated websites. In their study, Siby, et al. used a dataset of 1500 websites. The larger the number of websites is, the harder it is for the classifier to extract unique features for each of them.

Our defense gives good results, the accuracy and the recall are similar. It means that for a given website, the classifier makes as many false positives as false negatives. However, the defense still leaks some information as the accuracy is not close to zero.

If we investigate the trace produced by our defense (Figure 5.5) we find that some TCP packets are fragmented. TCP fragments packets when it receives a PDU larger than what the IP layer is allowed to send in one fragment. This behavior seems to only happen when there is real DNS traffic that is sent. Indeed, at the end of the connection when the connection only exchange padded packets to hide the length of the connection, we don't see fragmented packet. Therefore, this is probably a side effect of the bucket filter implemented with traffic control to limit the bandwidth when we send real data. This side effect is problematic for the

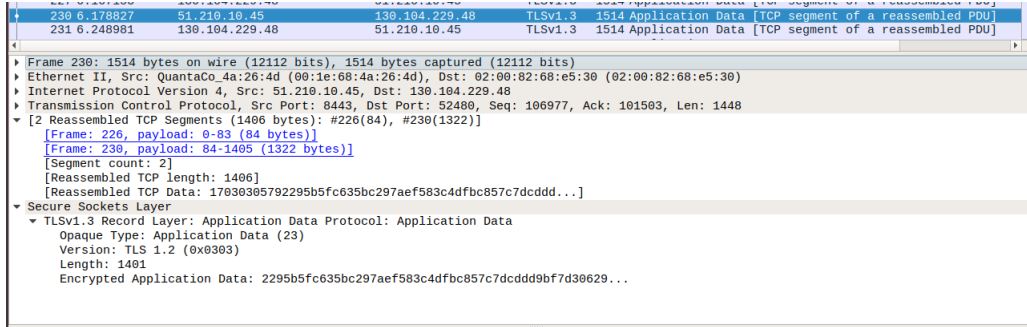


Figure 5.5: Capture of defended DoT.

performance of the defense, a classifier can correlate the presence of fragmented packets with the sending of DNS data. Thus, it means that the defense leaks information when real DNS data is sent over the connection. This can explain the 0.14 accuracy from the classifier.

One of the issues of our defense is the large overhead it produces, Figure 5.6 compares the average size of the traces for undefended and defended DoT in bytes. The majority of traces are around 620Kb which correspond to the constant rate the defense apply for a duration of 20 seconds. We can see that some traces are higher than that. For some reasons, the selenium driver that was accessing the web page took a lot of time to return for those samples (120 seconds for the larger one) during this time selenium was still doing DNS query which explains the larger size.

As we can see the overhead is not negligible. We decided to overestimate the value of the parameters as we were firstly interested in accuracy results. However, a more careful choice of parameters for the BuFLO defense may reduce that overhead. First of all, the value τ is largely overestimated. Future works may study the average duration of DoT connection to use a value closer to the reality. In the same way the size of the packet is overestimated. For undefended TLS record the size range from 100 to 800 bytes.

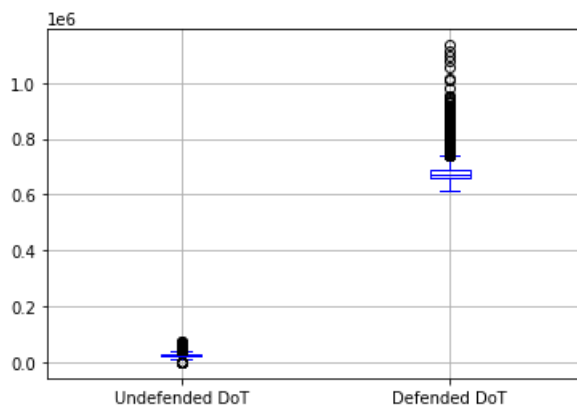


Figure 5.6: Size comparison between defended and undefended traffic.

The DNS classifier we used was mainly designed to classify undefended DoT traffic, thus they did not take into account timing features as length features were enough to give good results. However, we’ve seen in Section 5.1.2 that timing features are usually the hardest features to counter with defenses. To give a more accurate view of the performance of our defense, we wanted to evaluate it against a classifier that use timing features. As there is no DNS classifier at the moment that does that, we used the web classifier proposed by Rahmann, et al. We obtained the results in Table 5.2. Unfortunately, this classifier does not perform well on DNS traffic with an accuracy of only 0.49. Even though, the defense still gives similar results to the previous classifier, they are not admissible as the control group does

Web Classifier	Accuracy	Websites	Samples
Undefended DoT	0.4977	200	40
Defended DoT	0.1388	200	40

Table 5.2: Web classifier accuracy.

not perform as expected.

By changing the hyper parameters of the neural networks, it may be possible to obtain good performance on DNS. Hence, providing admissible results for our defense for this type of classifier. However, due to time constraint we won't be able to explore this idea in this thesis.

5.3.4 Conclusion and future works

In this chapter, we showed how we were able to implement a defense plugin against DNS fingerprinting on TLS using the uBPF virtual machine. Even though the plugin gives promising results, it can be improved on two axes. First of all, we could modify the plugin to prevent the fragmentation, moreover we've seen that the produced overhead is important. Future works may focus on optimizing the parameters to find a good compromise between the overhead and the efficiency of the solution.

In Section 5.1.2, we've talked about several other defenses. These defenses have not been tested to prevent DNS fingerprinting, it would be interesting to try to implement them on TLS using uBPF and see how they perform compared to the BuFLO defense.

Finally, we were not able to correctly test the efficiency of our solution against classifiers that uses timing features as they do not exist for DNS fingerprinting. It would be interesting to develop a DNS classifier that exploits all the possible features we described in Section 5.1.1. Compare to the classifier proposed by Siby, et al., this classifier would be designed to counter defenses.

Chapter 6

Conclusion

In this thesis, we showed how we were able to implement a modular version of the TLS protocol. With the help of the uBPF virtual machine, we are able to execute plugins that modify the behavior of TLS by adding or replacing functions in the calling sequence of the main implementation. We have extended the uBPF virtual machine to meet the requirements for developing complex plugins. Hence, we provided a powerful API that allow developers to customize the behavior of TLS.

We have illustrated the possibility of our implementation by developing a defense against DNS fingerprinting attacks, a type of traffic analysis attacks. Our defense adds padding to TLS record and shape the connection. By shaping the connection, we make sure that records are sent at a constant rate on the connection. To evaluate the effectiveness of our defense, we collected a dataset of DNS over TLS traces with and without our defense. A DNS trace consists of the exchange between a user and a DNS resolver when accessing a website. We visited a total of 200 websites to obtain a consequent pool of data.

Our dataset was feed to a state of the art classifier to perform DNS fingerprinting and gave significant results, showing that our defense was effective to lower the accuracy of DNS fingerprinting.

Discussion and future works

In this section we discuss the current state of our works and how it could be improved.

The list of protocol operation we provide does not cover all the function in PicotLS. Indeed, operations were added as the implementation of plugins progressed, when it was needed for the plugin. To allow developers to custom more features of

TLS, future works may focus on adding new protocol operations.

Even though the virtual machine adds a layer of security by performing various checks before and during the execution of plugins. It is possible to disrupt or crash the protocol with plugins by changing some of the state variable of the connection. The API we provide gives a lot of freedom for the developer. It would be interesting to study how to provide flexibility while ensuring that the protocol cannot be disrupted. To do so, we would need to work on the get/set API that can sanitize input and make sure that changes from plugin on the TLS state are safe for the connection.

The plugin we implemented to defend against fingerprinting attacks is a good starting point to a more in depth study about fingerprinting defense implementation. Even though the reasoning behind the defense is quite simple, it implements features that can be reused for more complex defenses. The plugin gives the ability to control the amount of padding to add to a packet. This amount can be easily modified and can even vary during the connection. We also implemented a way to send dummy packet at a given rate. As for padding, it is trivial to change this rate and make it variable during the connection. As we've seen in Chapter 5, the defense gives good results but induce a large overhead. Some other defenses like WTF-PAD are known to produces much less overhead. As we already have some tools to add padding and shape the traffic, implementing a defense inspired from WTF-PAD at the TLS layer would be possible. Other defense scheme could be thought of and easily implemented.

On the other hand we could work on improving the performance of DNS classifiers against padding defenses. As we've seen in Chapter 5, it is necessary to take into account all the possible features to build a performing classifier. However, this is not enough to obtain good performance. As we have seen for the web classifier, the accuracy against undefended DoT are bad. To obtain good results we would need to modify the hyper parameters of the neural networks. We may also need to modify the architecture of the different layers of the neural network until we obtain good results on undefended DoT.

Bibliography

- [1] E. Rescorla, “The Secure HyperText Transfer Protocol,” Internet Requests for Comments, RFC Editor, 2660 2660, August 1999. [Online]. Available: <https://tools.ietf.org/html/rfc2660>
- [2] Z. Hu, “Specification for DNS over Transport Layer Security (TLS),” Internet Requests for Comments, RFC Editor, RFC 7858, May 2016. [Online]. Available: <https://tools.ietf.org/html/rfc7858>
- [3] P. Hoffman, “SMTP Service Extension for Secure SMTP over Transport Layer Security,” Internet Requests for Comments, RFC Editor, 3207 3207, February 2002. [Online]. Available: <https://tools.ietf.org/html/rfc3207>
- [4] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, “Pluginizing quic,” in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 59–74.
- [5] T. Wirtgen, C. Dénos, Q. De Coninck, M. Jadin, and O. Bonaventure, “The case for pluginized routing protocols,” in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 2019, pp. 1–12.
- [6] “Picotls,” <https://github.com/h2o/picotls>.
- [7] M. Fleming, “A thorough introduction to ebpf.” [Online]. Available: <https://lwn.net/Articles/740157/>
- [8] P. Hoffman, “DNS queries over HTTPS (DoH),” Internet Requests for Comments, RFC Editor, RFC 8484, October 2018. [Online]. Available: <https://tools.ietf.org/html/rfc8484>
- [9] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” Internet Requests for Comments, RFC Editor, RFC 8446, August 2018. [Online]. Available: <https://tools.ietf.org/html/rfc8446>

- [10] T. Dierks, “The Transport Layer Security (TLS) Protocol Version 1.2,” Internet Requests for Comments, RFC Editor, RFC 5246, August 2008. [Online]. Available: <https://tools.ietf.org/html/rfc5246>
- [11] V. Bernat, “TLS and Perfect Forward Secrecy,” November 2011. [Online]. Available: <https://vincent.bernat.ch/en/blog/2011-ssl-perfect-forward-secrecy>
- [12] P. Rogaway, “Evaluation of some blockcipher modes of operation,” *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.
- [13] T. Polk, K. McKay, and S. Chokhani, “Guidelines for the selection, configuration, and use of transport layer security (tls) implementations,” *NIST special publication*, vol. 800, no. 52, p. 32, 2014.
- [14] H. Krawczyk, “The order of encryption and authentication for protecting communications (or: How secure is SSL?),” *Annual International Cryptology Conference*, 20001.
- [15] P. Gutmann, “Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS),” Internet Requests for Comments, RFC Editor, RFC 7366, August 2014. [Online]. Available: <https://tools.ietf.org/html/rfc8446>
- [16] OWASP, “Overview of tls v1.3.” [Online]. Available: https://owasp.org/www-pdf-archive/TLS_v1.3_Overview_OWASP_Final.pdf
- [17] “ubpf,” <https://github.com/p-quirk/ubpf>.
- [18] kernel.org, “ebpf instruction set.” [Online]. Available: <https://www.kernel.org/doc/Documentation/networking/filter.txt>
- [19] LLVM, “Clang compiler.” [Online]. Available: <https://clang.llvm.org/>
- [20] GCC, “Gcc inline.” [Online]. Available: <http://gcc.gnu.org/onlinedocs/gcc/Inline.html>
- [21] B. Kenwright, “Fast efficient fixed-size memory pool: No loops and no overhead,” in *The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, 2012.
- [22] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, “Touching from a distance: Website fingerprinting attacks and defenses,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 605–616.

- [23] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, “Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail,” in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 332–346.
- [24] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, “Effective attacks and provable defenses for website fingerprinting,” in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 143–157.
- [25] R. Houser, Z. Li, C. Cotton, and H. Wang, “An investigation on information leakage of dns over tls,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 123–137.
- [26] S. Siby, M. Juarez, C. Diaz, N. Vallina-Rodriguez, and C. Troncoso, “Encrypted dns→ privacy? a traffic analysis perspective,” *arXiv preprint arXiv:1906.09682*, 2019.
- [27] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg, “A systematic approach to developing and evaluating website fingerprinting defenses,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 227–238.
- [28] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt, “A critical evaluation of website fingerprinting attacks,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 263–274.
- [29] X. Cai, R. Nithyanand, and R. Johnson, “Cs-bufo: A congestion sensitive website fingerprinting defense,” in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, 2014, pp. 121–130.
- [30] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright, “Toward an efficient website fingerprinting defense,” in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 27–46.
- [31] M. S. Rahman, P. Sirinam, N. Matthews, K. G. Gangadhara, and M. Wright, “Tik-tok: The utility of packet timing in website fingerprinting attacks,” *arXiv preprint arXiv:1902.06421*, 2019.
- [32] T. Wang and I. Goldberg, “Walkie-talkie: An efficient defense against passive website fingerprinting attacks,” in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1375–1390.
- [33] M. Kerrisk, *Select*, Linux Programmer’s Manual. [Online]. Available: <https://www.man7.org/linux/man-pages/man2/select.2.html>

- [34] infradead, “Netlink protocol library suite.” [Online]. Available: <https://www.infradead.org/~tgr/libnl>
- [35] M. Kerrisk, *TC*, Linux Programmer’s Manual. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc.8.html>
- [36] “Selenium.” [Online]. Available: <https://www.selenium.dev/>
- [37] V. Jacobson, *Manpage of TCPDUMP*, TCPDUMP. [Online]. Available: <https://www.tcpdump.org/manpages/tcpdump.1.html>
- [38] “doh traffic analysis,” https://github.com/spring-epfl/doh_traffic_analysis/commits/master.
- [39] “Tik_tok,” https://github.com/msrocean/Tik_Tok.
- [40] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci, “Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows.” in *NDSS*, vol. 11, 2011.
- [41] M. Perry, “Experimental defense for website traffic fingerprinting,” *Tor project Blog*.” <https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting>, 2011.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl