Radbound University

Master's Thesis Computing Science
in Cyber Security

Fingerprinting TLS Implementations Using Model Learning

Author:
Erwin Janssen

Supervisor:
Frits Vaandrager

Daily supervisor:
Joeri de Ruiter

Second reader:
Erik Poll

Radboud University Nijmegen
Institute for Computing and Information Sciences
Digital Security

March 19, 2021

Abstract

We developed a new approach for generating and matching fingerprints for network protocol implementations and applied this to TLS server implementations. For generating the fingerprints we used model learning (a.k.a. active automata learning) to infer the state machine models of more than 200 different versions of two major TLS implementations. To perform the identification, we applied and compared two different methods. One method uses the adaptive distinguishing graph (ADG) algorithm, a direct generalization of Lee & Yannakakis algorithm for adaptive distinguishing sequences. The ADG pre-computes a decision tree with fixed inputs, and each model is identified through a unique input-output sequence. The other method is the *heuristic decision tree* (HDT), a new method that we present here. The HDT compares all models simultaneously during the identification and dynamically chooses which input to send based on heuristics. It is highly configurable and extendable, allowing multiple (and custom) input selection algorithms. We benchmarked and compared both methods using the models we learned. The performance of the two methods were comparable in our tests and we concluded that both the ADG and the HDT are capable of finding efficient input sequences to perform fingerprint matching.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Transport Layer Security . . . 3
2.1.1 Architecture . . . 4
2.1.2 Record protocol . . . 5
2.1.3 Handshake protocol . . . 7
2.2 State machines . . . 11
2.2.1 Mealy Machine . . . 11
2.2.2 Model learning . . . 11
2.2.3 State identification . . . 12

3 Related work **14**
3.1 Model learning . . . 14
3.2 Fingerprinting TLS . . . 14
3.2.1 Fingerprinting TLS usage of client and server applications . . . 15
3.2.2 Fingerprinting TLS server implementations . . . 15
3.3 Formal fingerprint matching . . . 15

4 Solution overview **18**
4.1 Target protocol versions . . . 18
4.2 Architecture . . . 18
4.3 Implementation details . . . 20

5 Building the implementations **21**
5.1 Build manager . . . 22
5.1.1 Overview . . . 23
5.1.2 Details . . . 23
5.2 Build components . . . 24
5.2.1 Overview . . . 25
5.2.2 Details . . . 25
5.3 Adding a new implementation . . . 27
5.3.1 Overview . . . 27
5.3.2 Details . . . 27
5.4 Discussion . . . 28
5.4.1 New versions . . . 28
5.4.2 Maintenance . . . 28
5.4.3 Future work . . . 29

6 Automated learning **30**
6.1 Learn manager . . . 30

6.1.1 Overview . . . 30
6.1.2 Details . . . 31
6.2 Learning setup . . . 32
6.2.1 Overview . . . 32
6.2.2 Details . . . 33
6.3 Learning alphabet . . . 35
6.4 Learning results . . . 36
6.5 Discussion . . . 37
6.5.1 Implementation details . . . 37
6.5.2 Models . . . 37

7 Identification 40
7.1 General process . . . 40
7.1.1 Removing duplicate models . . . 41
7.1.2 Construct model tree . . . 41
7.1.3 Identification . . . 42
7.2 Running example . . . 42
7.3 Distinguishing sequences . . . 42
7.3.1 Pairwise distinguishing sequences . . . 43
7.3.2 Lee & Yannakakis . . . 43
7.3.3 Adaptive distinguishing graph . . . 46
7.3.4 Integrating ADG . . . 48
7.4 Heuristic decision tree . . . 49
7.4.1 Normalize models . . . 49
7.4.2 Identification procedure . . . 51
7.4.3 Input selection . . . 52
7.4.4 Example . . . 55
7.4.5 Future work . . . 55
7.5 Comparison . . . 57
7.5.1 Benchmark setup . . . 58
7.5.2 Benchmark results . . . 60

8 Conclusion 66

References 69

A Details of learned models 72
A.1 TLS 1.0 . . . 72
A.2 TLS 1.1 . . . 73
A.3 TLS 1.2 . . . 76

B Model weights 78
B.1 TLS 1.0 . . . 78
B.2 TLS 1.1 . . . 79
B.3 TLS 1.2 . . . 80

C Benchmark results per model 81
C.1 TLS 1.0 . . . 81
C.1.1 Number of inputs . . . 81
C.1.2 Number of resets . . . 83
C.1.3 Computation time . . . 85
C.2 TLS 1.1 . . . 87
C.2.1 Number of inputs . . . 87
C.2.2 Number of resets . . . 89

C.2.3 Computation time 91
C.3 TLS 1.2 93
C.3.1 Number of inputs 93
C.3.2 Number of resets 95
C.3.3 Computation time 97

D Weighted benchmark statistics **99**
D.1 Weight function "equal" 99
D.1.1 Number of inputs 99
D.1.2 Number of resets 102
D.1.3 Computation time 104
D.2 Weight function "count" 106
D.2.1 Number of inputs 106
D.2.2 Number of resets 108
D.2.3 Computation time 110
D.3 Weight function "recent" 112
D.3.1 Number of inputs 112
D.3.2 Number of resets 114
D.3.3 Computation time 116

List of Tables

2.1 TLS versions 3
2.2 TLS terminology, taken from [25] 4

3.1 Taxonomy of network fingerprinting problems [32] 16

6.1 Learning alphabet 35
6.2 Models learned per implementation per TLS version 36
6.3 Number of unique models per TLS version 36
6.4 Number of unique models per implementation 37

7.1 Different weight functions applied to models learned for TLS 1.2 59
7.2 Number of nodes of each model tree 60
7.3 Number of inputs for each model of TLS 1.2 61
7.4 Number of inputs, distribution of values for each model of TLS 1.2 with HDT Random 61
7.5 Benchmark summary: Number of inputs with weight function 'equal' for TLS 1.2 63

A.1 Details of models learned for TLS 1.0 72
A.2 Implementation versions of models learned for TLS 1.0 72
A.3 Details of models learned for TLS 1.1 73
A.4 Implementation versions of models learned for TLS 1.1 75
A.5 Details of models learned for TLS 1.2 76
A.6 Implementation versions of models learned for TLS 1.2 77

C.1 Number of inputs for each model of TLS 1.0 81
C.2 Number of inputs, distribution of values for each model of TLS 1.0 with HDT Random 82
C.3 Number of resets for each model of TLS 1.0 83
C.4 Number of resets, distribution of values for each model of TLS 1.0 with HDT Random 84
C.5 Time in seconds for each model of TLS 1.0 85
C.6 Time in seconds, distribution of values for each model of TLS 1.0 with HDT Random 86
C.7 Number of inputs for each model of TLS 1.1 87
C.8 Number of inputs, distribution of values for each model of TLS 1.1 with HDT Random 88
C.9 Number of resets for each model of TLS 1.1 89
C.10 Number of resets, distribution of values for each model of TLS 1.1 with HDT Random 90
C.11 Time in seconds for each model of TLS 1.1 91
C.12 Time in seconds, distribution of values for each model of TLS 1.1 with HDT Random 92

C.13 Number of inputs for each model of TLS 1.2 93
C.14 Number of inputs, distribution of values for each model of TLS 1.2 with HDT Random 94
C.15 Number of resets for each model of TLS 1.2 95
C.16 Number of resets, distribution of values for each model of TLS 1.2 with HDT Random 96
C.17 Time in seconds for each model of TLS 1.2 97
C.18 Time in seconds, distribution of values for each model of TLS 1.2 with HDT Random 98

D.1 Benchmark summary: Number of inputs with weight function ‘equal’ for all TLS versions 99
D.2 Benchmark summary: Number of inputs with weight function ‘equal’ for TLS 1.0 99
D.3 Benchmark summary: Number of inputs with weight function ‘equal’ for TLS 1.1 99
D.4 Benchmark summary: Number of inputs with weight function ‘equal’ for TLS 1.2 100
D.5 Benchmark summary: Number of resets with weight function ‘equal’ for all TLS versions 102
D.6 Benchmark summary: Number of resets with weight function ‘equal’ for TLS 1.0 102
D.7 Benchmark summary: Number of resets with weight function ‘equal’ for TLS 1.1 102
D.8 Benchmark summary: Number of resets with weight function ‘equal’ for TLS 1.2 102
D.9 Benchmark summary: Time in seconds with weight function ‘equal’ for all TLS versions 104
D.10 Benchmark summary: Time in seconds with weight function ‘equal’ for TLS 1.0 104
D.11 Benchmark summary: Time in seconds with weight function ‘equal’ for TLS 1.1 104
D.12 Benchmark summary: Time in seconds with weight function ‘equal’ for TLS 1.2 104
D.13 Benchmark summary: Number of inputs with weight function ‘count’ for all TLS versions 106
D.14 Benchmark summary: Number of inputs with weight function ‘count’ for TLS 1.0 106
D.15 Benchmark summary: Number of inputs with weight function ‘count’ for TLS 1.1 106
D.16 Benchmark summary: Number of inputs with weight function ‘count’ for TLS 1.2 106
D.17 Benchmark summary: Number of resets with weight function ‘count’ for all TLS versions 108
D.18 Benchmark summary: Number of resets with weight function ‘count’ for TLS 1.0 108
D.19 Benchmark summary: Number of resets with weight function ‘count’ for TLS 1.1 108
D.20 Benchmark summary: Number of resets with weight function ‘count’ for TLS 1.2 108
D.21 Benchmark summary: Time in seconds with weight function ‘count’ for all TLS versions 110
D.22 Benchmark summary: Time in seconds with weight function ‘count’ for TLS 1.0 110
D.23 Benchmark summary: Time in seconds with weight function ‘count’ for TLS 1.1 110
D.24 Benchmark summary: Time in seconds with weight function ‘count’ for TLS 1.2 110
D.25 Benchmark summary: Number of inputs with weight function ‘recent’ for all TLS versions 112
D.26 Benchmark summary: Number of inputs with weight function ‘recent’ for TLS 1.0 112
D.27 Benchmark summary: Number of inputs with weight function ‘recent’ for TLS 1.1 112
D.28 Benchmark summary: Number of inputs with weight function ‘recent’ for TLS 1.2 112
D.29 Benchmark summary: Number of resets with weight function ‘recent’ for all TLS versions 114
D.30 Benchmark summary: Number of resets with weight function ‘recent’ for TLS 1.0 114
D.31 Benchmark summary: Number of resets with weight function ‘recent’ for TLS 1.1 114
D.32 Benchmark summary: Number of resets with weight function ‘recent’ for TLS 1.2 114
D.33 Benchmark summary: Time in seconds with weight function ‘recent’ for all TLS versions 116
D.34 Benchmark summary: Time in seconds with weight function ‘recent’ for TLS 1.0 116
D.35 Benchmark summary: Time in seconds with weight function ‘recent’ for TLS 1.1 116

D.36 Benchmark summary: Time in seconds with weight function 'recent' for TLS 1.2 116

List of Figures

2.1 TLS layers architecture . . . 6
2.2 Connection states in TLS 1.0. The dashed states are the pending states. . . . 6
2.3 TLS 1.0 handshake . . . 8
2.4 TLS 1.3 handshake . . . 9
2.5 A simple Mealy machine . . . 11
2.6 Example of an adaptive distinguishing sequence for a finite-state machine [21] 13

4.1 General flow of stages . . . 19
4.2 Overview of all components . . . 19

5.1 Pipeline design . . . 22
5.2 Components used for the build stage . . . 22
5.3 Overview of the periodic build . . . 23
5.4 Different repositories on Drone . . . 23
5.5 Pipelines in the Drone dashboard . . . 26

6.1 Components used for the learn stage . . . 30
6.2 Pipeline for learning . . . 31
6.3 Conceptual learning setup . . . 33
6.4 Learning setup in practice . . . 33
6.5 Learning pipeline on Drone . . . 35
6.6 Reduced model for OpenSSL 0.9.7 with TLS 1.0 . . . 38

7.1 Example models . . . 42
7.2 Conversion from separate to combined models . . . 44
7.3 Example of an adaptive distinguishing sequence for a finite-state machine (duplicate of Section 2.2.3) . . . 45
7.4 Example models converted to a labeled transition system . . . 47
7.5 Adaptive distinguishing graph of the example models . . . 47
7.6 Model tree of the adaptive distinguishing graph for the example models . . . 48
7.7 Model A with normalized tree result . . . 50
7.8 Heuristic decision tree for example models . . . 51
7.9 Node encoding example for heuristic decision tree . . . 52
7.10 Different weight functions applied to the example heuristic decision tree . . . 54
7.11 Gini impurities for the input selection of the first descent . . . 56
7.12 Identification for example models . . . 56
7.13 Benchmark results: Number of inputs with weight function ‘equal’ . . . 64

D.1 Benchmark results: Number of inputs with weight function ‘equal’ . . . 101
D.2 Benchmark results: Number of inputs with weight function ‘equal’ . . . 103
D.3 Benchmark results: Computation time with weight function ‘equal’ . . . 105
D.4 Benchmark results: Number of inputs with weight function ‘count’ . . . 107

D.5 Benchmark results: Number of inputs with weight function ‘count’ 109
D.6 Benchmark results: Computation time with weight function ‘count’ 111
D.7 Benchmark results: Number of inputs with weight function ‘recent’ 113
D.8 Benchmark results: Number of inputs with weight function ‘recent’ 115
D.9 Benchmark results: Computation time with weight function ‘recent’ 117

Chapter 1

Introduction

The Transport Layer Security (TLS) protocol is one of the cornerstones of secure communication on the Internet. It is widely used to secure communication in a variety of settings, including web browsing, email and virtual private networks. However, TLS is not without its flaws, and vulnerabilities have been found in the past: both in the protocol itself [1, 5, 23], and in specific software implementations [4, 6, 11, 12, 29].

It is beneficial to know if the TLS implementation used by a (web) server contains any known vulnerabilities. For users this might affect the trust they have in the security of the connection, and therefore the decision whether to share confidential information with this server or not. Network operators or cloud providers also want to ensure that no vulnerable services are present in their environment. If one would know the specific name and version of the TLS implementation running on a server, it is possible to look up vulnerabilities in the CVE database[1]. Unfortunately this information isn't always straight-forward to come by, as the TLS protocol does not define a standard way to include a version banner. As a result, an alternative method is required: a way to fingerprint and identify a TLS server implementation.

Fingerprinting TLS has been a topic of previous research and this is done from multiple angles. The focus is often on identifying a specific application, such as a piece of malware, through its use of TLS. This can be done by creating a fingerprint based on the contents of the TLS `ClientHello` or `ServerHello` message [16, 35, 36]. This method allows for passive fingerprint collection and identification, but it is susceptible to small configurations differences since it relies on the actual content of the message. A client or server with a different set of supported cipher suites or extensions will lead to a different fingerprint result, even though the underlying implementation might be the same. While this method is useful (and has been applied successfully) for identifying specific applications, it is not suitable to create a detailed fingerprint of the actual TLS implementation itself.

Fingerprints for the TLS implementations themselves can be created by looking at behavior differences between implementations. Instead of passively looking at the TLS handshake contents, this is an active approach: message sequences are sent to the target and the response is recorded. This method looks more at the type of messages and less at their content, this means configuration does not have as much impact on the identification process compared to the content based method. Current implementations of this method make use of hand-picked input sequences, and often send many sequences in order to perform an identification. The downsides of these hand-picked sequences is also that creating new sequences is a manual task that requires expert knowledge of the protocol.

In this thesis, we present a new approach for fingerprinting generation and matching for TLS

[1] https://cve.mitre.org/

server implementations leveraging multiple formal methods. The approach consists of two steps. The first step is to generate fingerprints using *model learning* [39], a black-box analysis technique where the internal protocol state machine of software is inferred by sending valid messages in an arbitrary order. The learned model of an implementation, which is a Mealy Machine, serves as its fingerprint. The second step is to use these fingerprints to determine efficient input sequences that match the behavior of a given TLS implementation to the correct model. We apply and compare two methods to automatically determine these inputs: *adaptive distinguishing graph* (ADG) and *heuristic decision tree* (HDT). The ADG [7] is a direct generalization of Lee & Yannakakis' algorithm for computing adaptive distinguishing sequence [21]. The HDT is a new method that we present in this thesis, it compares all available models simultaneously and selects the inputs based on heuristics. While we focus on the TLS protocol in this thesis, the approach we present is not specific to TLS and can be applied to other protocols.

For a proof of concept we focused on two major TLS implementations: OpenSSL[2] and mbed TLS[3]. Both implementations have a long history which gives us many versions to analyze. We applied our approach to 134 versions of OpenSSL and 114 versions of mbed TLS for several versions of the TLS protocol. To support our research and in attempt to future-proof our work, we automated as much as possible. This includes pipelines that periodically fetch, build and package new versions of each implementation and then learn their models. We also developed a command line tool with the name `tlsprint` to support various tasks and perform the actual identification.

The rest of this thesis is structured as follows. We start with some theoretical background in Chapter 2, followed by an overview of related work in Chapter 3. In Chapter 4 we provide an overview of how we implemented our solution, which is split up in independent stages. Each stage is discussed in detail in its own chapter, starting with the build process in Chapter 5, followed by the learning setup in Chapter 6 and lastly the comparison of the identification methods in Chapter 7. We wrap up with a conclusion in Chapter 8.

[2]https://www.openssl.org/
[3]https://tls.mbed.org/

Chapter 2

Preliminaries

This thesis builds upon different fields and theories. The two most significant are discussed in this chapter: the TLS protocol in Section 2.1 and state machines in Section 2.2.

2.1 Transport Layer Security

In this section we will give an overview of the Transport Layer Security (TLS) protocol. Readers who are familiar with TLS and its handshake protocol can safely skip this. For more details we refer the reader to the specifications listed in Table 2.1.

TLS is a security protocol that "*allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery*" [25]. In a typical protocol run, the client and server set up a stateful connection through a handshake procedure in which they (optionally) authenticate each other and negotiate a session key. After the handshake, the client and server can securely exchange data. The protocol is widely used to secure network communication, such web browsing (HTTPS), email (SMTPS, IMAPS) and virtual private networks (OpenVPN). As a result of this widespread use, TLS has been extensively studied, strengthened and extended over the years. It is generally regarded as one of the most mature security protocols available.

The first version of TLS was published in 1999 as a successor of Secure Sockets Layer (SSL), and it has received three updates since then. The most recent version, TLS 1.3, provides significant changes to reduce complexity and improve security and performance. Table 2.1 shows when each version of TLS was released, and which RFC defines the specification.

Table 2.1: TLS versions

Version	Year	RFC
1.0	1999	2246[1]
1.1	2006	4346[2]
1.2	2008	5246[3]
1.3	2018	8446[4]

The TLS standard defines some terminology, shown in Table 2.2 [25] which will also be used

[1] https://tools.ietf.org/html/rfc2246
[2] https://tools.ietf.org/html/rfc4346
[3] https://tools.ietf.org/html/rfc5246
[4] https://tools.ietf.org/html/rfc8446

here for consistency.

Table 2.2: TLS terminology, taken from [25]

Term	Definition
client	The endpoint initiating the TLS connection.
connection	A transport-layer connection between two endpoints.
endpoint	Either the client or server of the connection.
handshake	An initial negotiation between client and server that establishes the parameters of their subsequent interactions within TLS.
peer	An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is not the primary subject of discussion.
receiver	An endpoint that is receiving records.
sender	An endpoint that is transmitting records.
server	The endpoint that did not initiate the TLS connection.

In the following sections, the different versions of the TLS protocol will be described in more detail, starting with a high level view of the architecture and followed by the two primary components of the TLS protocol: the record protocol and the handshake protocol. The discussion will remain relatively high-level. For more details on each specific version, we refer the reader to the appropriate RFC.

2.1.1 Architecture

TLS is a security protocol which follows a client-server model. It is application protocol independent, which means that any application protocol (e.g. HTTP or DNS) can layer on top of it without any modifications. From the application layer, TLS can be viewed as a black box which transports data with certain security guarantees: application data goes in, encrypted data is sent, and application data comes out at the other side. How the data is secured and authenticated, and how the security parameters are established between the client and the server, is the responsibility of the TLS protocol.

For the actual transmission of bytes, TLS is dependent on the underlying transport layer. TLS requires a transport layer which provides a reliable, in-order data stream, which often means TCP is used. There is also an adapted version of TLS that does not have these requirements, called Datagram Transport Layer Security (DTLS) [27].

The TLS protocol aims to provide a secure channel with the following three properties [25, 26]:

- **Authentication**: Both endpoints can authenticate their peer using asymmetric cryptography or a symmetric pre-shared key. This authentication is not mandatory, and client authentication is often omitted. Since TLS 1.3, the server side of the channel is always authenticated.
- **Confidentiality**: After the handshake procedure, all data sent over the channel is only visible to the endpoints. TLS does not hide the length of the data sent, but endpoints are able to pad the data if necessary.
- **Integrity**: An adversary cannot modify the data sent over the channel without being detected by the endpoints.

These security properties should hold, even in the event that an adversary has complete control over the network [25], and is able to drop, modify, and inject packets in the network.

While TLS can be treated as a single logical layer, it is actually composed of two layers itself, where each layer uses its own sub-protocols. The upper layer protocols define the actual

communication (such as the *handshake protocol* and the *application data protocol*), and pass their messages to the lower layer, which is home to the *record protocol.* The record protocol uses the parameters established by the handshake protocol to protect traffic between the communicating peers [25].

The TLS standards specify four protocols for the upper layer:

- **Handshake**: The handshake protocol is responsible for establishing the connection, setting the proper security parameters in the record protocol, and optionally authenticating the peer. This is one of the most important components of the TLS protocol. There are two versions of the handshake protocol, the one used in TLS 1.0 to 1.2, and the one introduced in TLS 1.3. We shall refer to these two different handshakes as the *TLS 1.0 handshake* and the *TLS 1.3 handshake* respectively.
- **Change cipher spec**: This protocol consists of a single message (also called `ChangeCipherSpec`) and is used to "*signal transitions in ciphering strategies*" [26]. It is used during the TLS 1.0 handshake.
- **Alert**: To indicate a connection closure or error, each endpoint can send an alert at any time. Alert messages contain a description and a severity level (`warning` or `fatal`). The specification states that a fatal alert should always terminate the connection, but explicitly doesn't state how a warning should be handled, which leaves room for ambiguity. In TLS 1.3 this ambiguity is removed, as each alert should be treated as fatal.
- **Application data**: After the handshake, the application data protocol is used to transmit the actual application data. This is done by passing the data directly to the record protocol.

Compared with the handshake protocol, the other three upper layer protocols are quite simple. As such, most of the specification of TLS deals with the record protocol and the handshake protocol. Both will be discussed in more detail in the following sections.

2.1.2 Record protocol

The record protocol performs the heavy lifting when it comes to protecting the traffic between client and server. When an upper layer wants to send a message, it passes the contents to the record protocol. The record protocol then fragments the data into manageable blocks, compresses, encrypts and signs these blocks, and then passes them to the transport layer. Conversely, when the record layer receives data from the transport layer, this data is decrypted, verified, decompressed, reassembled, and then passed to the upper layer [26]. This flow is shown schematically in Figure 2.1.

The record protocol needs to keep track of the security parameters (the algorithms and their parameters, including keys) used for a connection, which makes TLS a stateful protocol. In the context of TLS the information about a connection is called the *connection state.* In the TLS 1.0 handshake there are always four connection states for every peer: the current read and write states, and the pending read and write states [26]. Each connection state has its own set of security parameters, including different keys. All records are always processed using the *current* read and write states: sending uses the write state, and receiving uses the read state. The write state of client communicates with the read state of the server, and vice versa, as can be seen in Figure 2.2a.

The current connection states cannot be mutated directly and instead, changes have to be made through the pending states. The values of the pending connection states can be set through the handshake protocol, and activated with the `ChangeCipherSpec` message. When this activation message is sent the peer activates its pending *write* connection state, when it is received the *read* connection state is switched. This is shown in Figure 2.2. This mechanism allows renegotiation of the security parameters during an active session.

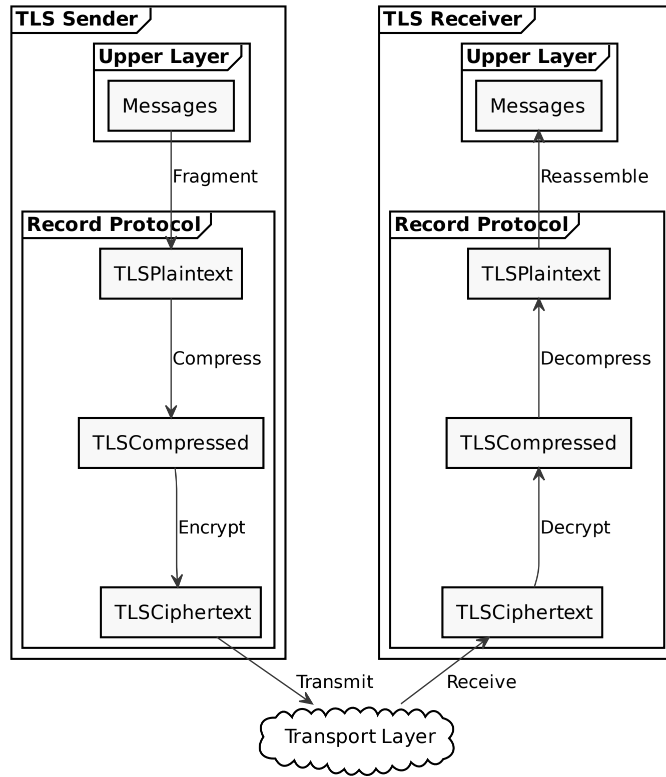
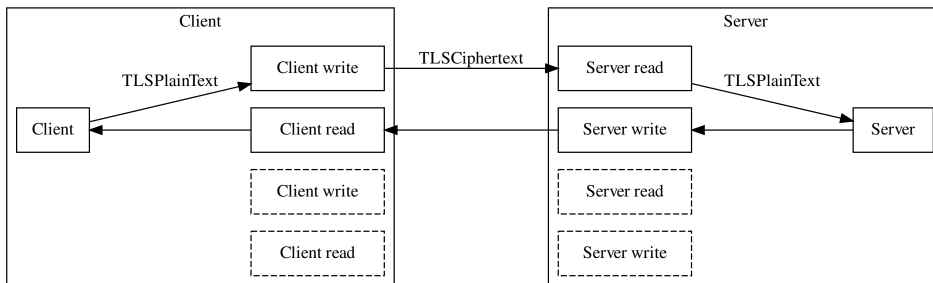
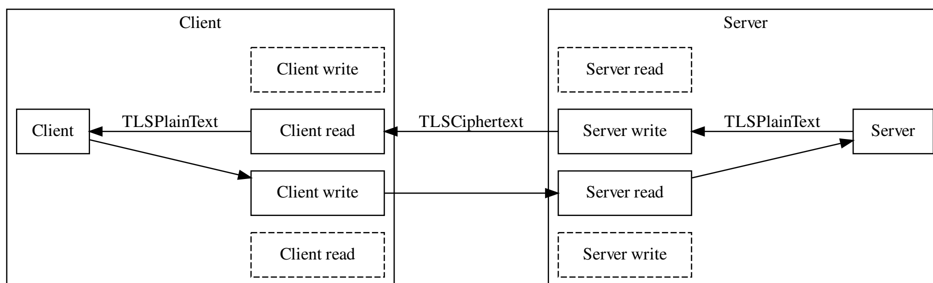


Figure 2.1: TLS layers architecture



(a) Initial connection states



(b) Connection states after `ChangeCipherSpec` from client

Figure 2.2: Connection states in TLS 1.0. The dashed states are the pending states.

The ability to renegotiate the security parameters of an active session has been a source of vulnerabilities in the past [42], as such TLS 1.3 forbids renegotiation [25]. This simplifies the connections states: TLS 1.3 only has current read and write states. The parameters of these states are set during the handshake, and cannot be modified after this point, making renegotiation impossible.

The record protocol relies on the handshake protocol to set the security parameters. How this is done, and how the handshake protocol works, is the topic of the next section.

2.1.3 Handshake protocol

The TLS Handshake Protocol allows the server and client to agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and generate shared secrets [25]. The handshake phase is critical, as most of the security for the rest of the connection dependents on security parameters negotiated in this phase. The handshake protocol aims to satisfy three properties (taken from [26]):

- "*The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA, DSA, etc.). This authentication can be made optional, but is generally required for at least one of the peers.*"
- "*The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.*"
- "*The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.*"

Because of these requirements, the handshake protocol is the most complicated part of the TLS protocol. As mentioned before, the TLS handshake has remained largely the same since the first version, but has received a major revision in version 1.3. Both versions will be discussed in more detail below.

2.1.3.1 TLS 1.0 handshake

The normal flow of the TLS 1.0 handshake is shown in Figure 2.3. This handshake has remained unmodified until the introduction of TLS 1.3 in 2018. Because it has remained the same for so long, this handshake is often referred to as "the TLS handshake" in other literature.

The client initiates the handshake by sending a `ClientHello`. In this message, the client specifies which version of the TLS protocol it would like to use and includes a list of cipher suites, compression methods and extensions it supports. The server then responds with a `ServerHello` in which the server notifies the client which parameters (TLS version, cipher suite, extensions, session ID etc.) the server has selected for this session. Optionally it then sends its own certificate, additional key exchange data and/or a request for the client's certificate. It ends this sequence with a `ServerHelloDone`, which does not contain any additional data.

After receiving the data from the server, the client first verifies that it is valid (e.g. the certificate is correct, it indeed supports the selected parameters, etc.). It then generates its own part of the key share and computes the session key. Depending on what the server requested, the client sends the required certificate or key exchange data, and then sends a `ChangeCipherSpec`. As mentioned before, the `ChangeCipherSpec` message indicates a change in ciphering strategies, which means the client will encrypt everything from this point forward with the computed session key, starting with the `Finished` message. The `Finished` message contains the hash of all handshake messages exchanged so far, so the peer can verify the integrity of the handshake. Interestingly, the `ChangeCipherSpec` is primarily used during the

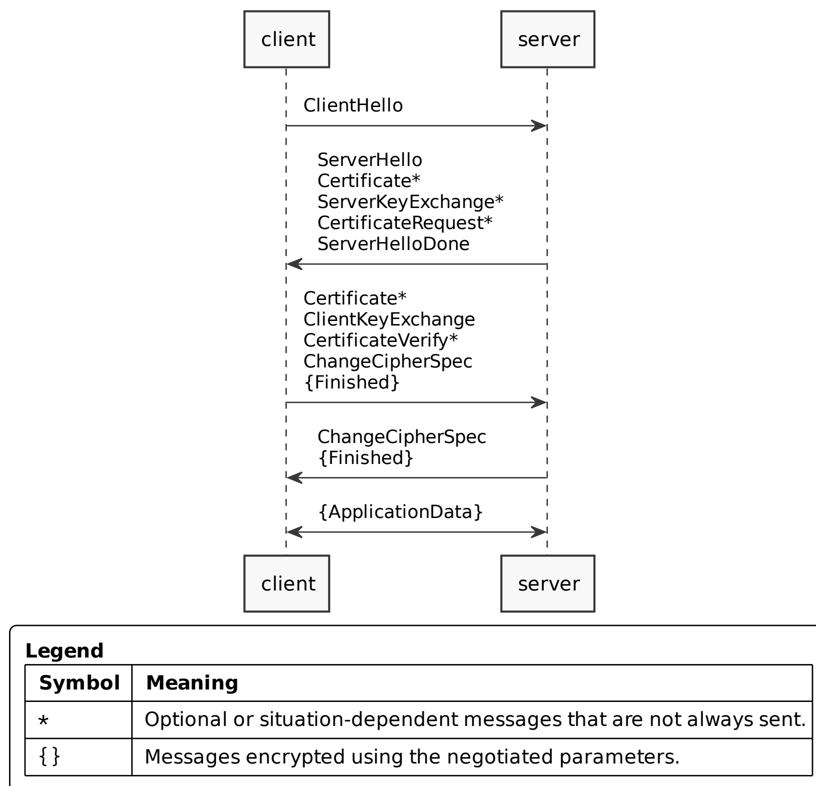


Figure 2.3: TLS 1.0 handshake

handshake, but is an independent TLS protocol type. According to the specification, this is "*to help avoid pipeline stalls*" [26].

When the server receives the messages from the client, it can also compute the session key. It then sends a `ChangeCipherSpec` to indicate that it will also start using encryption, and also ends with an encrypted `Finished`. After this point, the handshake is complete, and both client and server can start to exchange application data.

The handshake consists of quite some messages that must be sent back and forth between client and server and some expensive cryptographic operations, which introduces communication overhead. Since no application data can be transmitted before the handshake is complete, this results in a delay when setting up a connection. To mitigate this delay, the TLS handshake protocol also includes a session resumption mechanism, where the security parameters of a previous session are reused. When the client wants the resume a session, it can include the session ID of the previous session in the `ClientHello`. If the server agrees to resume the session, it immediately sends a `ServerHello` with the same session ID, followed by `ChangeCipherSepec` and `Finished`. The client also responds with `ChangeCipherSpec` and `Finished`, and the handshake is complete. Session resumption requires that both client and server keep track of past sessions and their parameters.

In the TLS 1.0 handshake, it is also possible to renegotiate the security parameters after the first handshake. At any time, both client and server can request renegotiation. The client can do this by sending a `ClientHello`, which will trigger the handshake. The server can send a `HelloRequest`, to notify the client that it should begin the negotiation process anew by sending a `ClientHello` [26]. As mentioned before, this renegotiation has been a source of vulnerabilities and is removed in TLS 1.3.

2.1.3.2 TLS 1.3 handshake

In TLS version 1.3, the handshake protocol has received a major update for the first time. The normal flow of this new handshake sequence is shown in Figure 2.4. The handshake has been significantly restructured and the standard now defines three separate logical phases (as indicated in Figure 2.4): key exchange, server parameters and authentication. The number of optional messages has increased, but superfluous messages (such as `ChangeCipherSpec` and `ServerHelloDone`) have been removed [25].

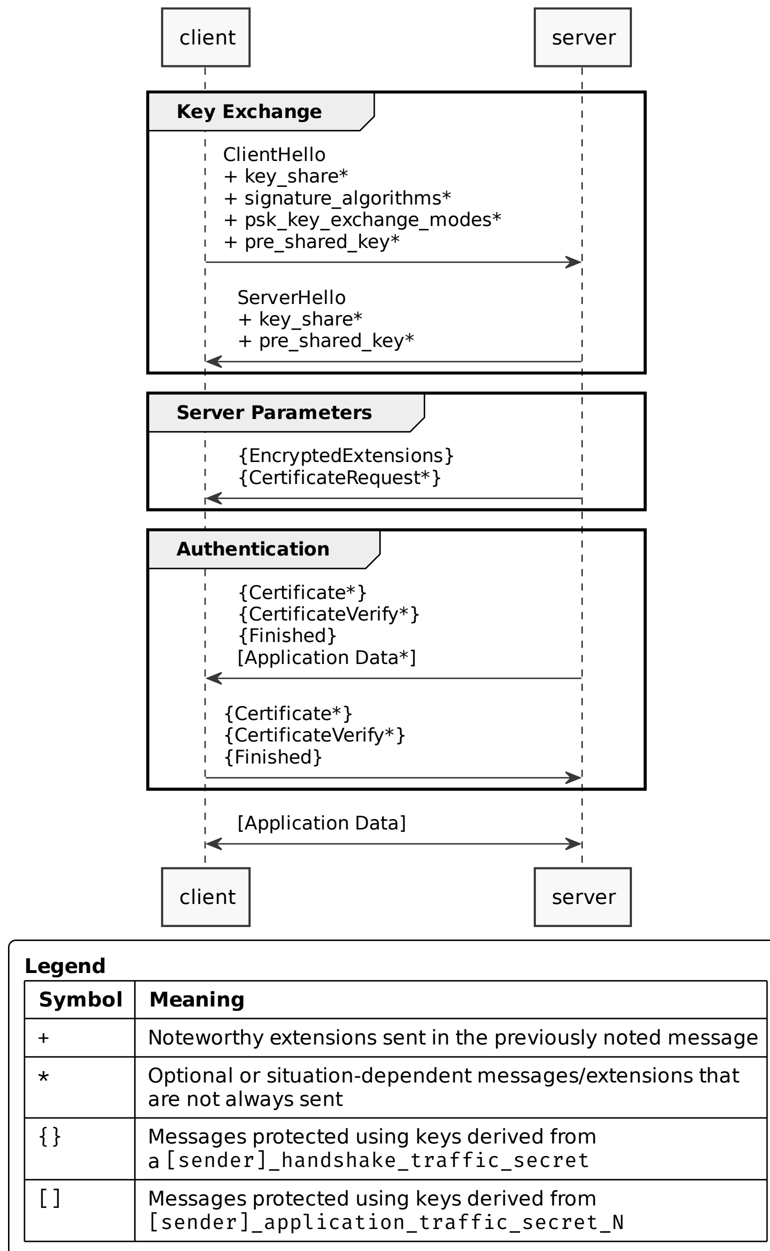


Figure 2.4: TLS 1.3 handshake

The TLS 1.3 handshake starts the same as before: with the client sending a `ClientHello` specifying supported cipher suites and extensions. In the earlier version of the handshake, the client would wait with generating its own key share until the server specified which parameters

to use. In this version however, the client can provide a list of key shares (one share for every key type supported by the client) directly after the `ClientHello` and the server can then select which one to use. If the client does not provide a list of key shares, the server will send a `HelloRetryRequest`, which specifies which key share the client should include, and the client should send a `ClientHello` again. After receiving the initial information from the client, the server responds with a `ServerHello` and its own key share, and the key exchange phase is complete. At this point, both client and server know both key shares, and they can compute the *handshake traffic secret*, which will be used to encrypt all following messages.

After the key exchange, the server sends its parameters to the client in the form of an *extension list* and whether it requests a client certificate. The server then sends its certificate if required. Unlike the TLS 1.0 handshake, these messages are encrypted, providing additional confidentiality.

At this point the server part of the handshake is already done, and the server notifies the client of this by sending a `Finished` (this is why the `ServerHelloDone` is no longer necessary). The server can then also compute the *application traffic secret*, and can already start sending application data.

After receiving all information of the server, the only thing the client has to do is send its certificate (if requested) and end with a `Finished`. The client can now also compute the *application traffic secret* and the handshake is complete.

TLS 1.3 defines a number of extensions to modify this handshake, including the so-called *zero round-trip time* (0-RTT) mode. This mode allows the client to include application data in its first message, at the cost of certain security properties. For more details on this extension and other, the reader is referred to RFC 8446 [25].

2.2 State machines

For the analysis and identification of the TLS implementations, we apply formal methodologies using state machines. In Section 2.2.1 we provide a definition for a *Mealy Machine*, the type of state machine we use to model the TLS implementations. A short introduction to *model learning* and the *state identification* problem are given in Section 2.2.2 and Section 2.2.3 respectively.

2.2.1 Mealy Machine

A *Mealy machines* is a finite-state machine (FSM), which produces its outputs based on the state transitions after receiving inputs [21]. A formal definition for a Mealy machine, quoted from [39], is:

> "*A* (deterministic) Mealy machine *is a tuple* $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$*, where* I *is a finite set of inputs,* O *is a finite set of outputs,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $\delta : Q \times I \to Q$ *is a transition function, and* $\lambda : Q \times I \to O$ *is an output function.*"

> "*Output function* λ *is extended to sequences of inputs by defining, for all* $q \in Q$*,* $i \in I$*, and* $\sigma \in I^*$*,* $\lambda(q, \epsilon) = \epsilon$*, and* $\lambda(q, i\sigma) = \lambda(q, i)\lambda(\delta(q, i), \sigma)$*. The behavior of Mealy machine* $\mathcal{M}$ *is defined by function* $A_{\mathcal{M}} : I^* \to O^*$ *with* $A_{\mathcal{M}}(\sigma) = \lambda(q_0, \sigma)$*, for* $\sigma \in I^*$*. Mealy machines* $\mathcal{M}$ *and* $\mathcal{N}$ *are equivalent, denoted* $\mathcal{M} \approx \mathcal{N}$*, iff* $A_{\mathcal{M}} = A_{\mathcal{N}}$*. Sequence* $\sigma \in I^*$ *distinguishes* $\mathcal{M}$ *and* $\mathcal{N}$ *if and only if* $A_{\mathcal{M}}(\sigma) \neq A_{\mathcal{N}}(\sigma)$*.*"

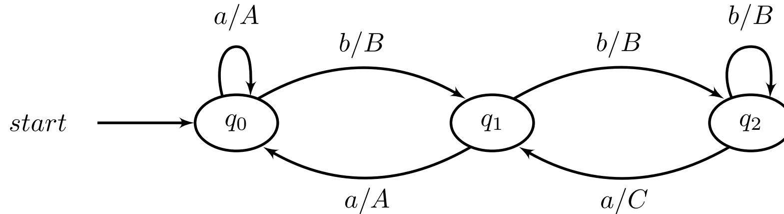


Figure 2.5: A simple Mealy machine

In Figure 2.5 a simple example of a Mealy machine (taken from [39]) is shown. This machine has input symbols $I = \{a, b\}$, output symbols $O = \{A, B, C\}$, states $Q = \{q_0, q_1, q_2\}$ and the initial state is q_0. In the diagram, the states are shown as nodes and the transitions as edges between them. The edges are annotated with *input/output*, to indicate which input causes the transition and what the resulting output is. As an example, when the machine in Figure 2.5 is in the initial state q_0, supplying input message b will cause a transition to state $\delta(q_0, b) = q_1$ and give output $\lambda(q_0, b) = B$.

2.2.2 Model learning

In this section we will provide an overview of the field of *Model Learning*. This overview is largely based on [39], and we refer the reader there for a more thorough review.

Model learning is a formal methodology for analyzing stateful systems. It is also known as *state machine inference* [10], *active automata learning* [17] and *protocol state fuzzing* [29]. The goal of model learning is to "*construct black-box state diagram models of software and hardware systems by providing inputs and observing outputs*" [39]. Model learning is a vast field of research in itself, and many types of models can be inferred, such as hidden Markov models, class diagrams and state machines [39]. In this thesis, we will infer Mealy machines from the TLS server implementations we analyze.

There are multiple algorithms available for model learning, but the most efficient all follow the framework of a *minimally adequate teacher* (MAT) [13]. This framework, published by Angluin in 1987 [3], views the learning process as a game between a *learner* and a *teacher* [13]. The teacher has full knowledge of a given state diagram $\mathcal{M}$, which in our case is a Mealy machine representing the behavior of a TLS server. The learner must then infer the behavior of this state machine by sending *membership* and *equivalence* queries. These queries are described in [39] as follows:

- "*With a* membership query *(MQ), the learner asks what the output is in response to an input sequence $\sigma \in I^*$. The teacher answers with output sequence $A_{\mathcal{M}}(\sigma)$.*"
- "*With an* equivalence query *(EQ), the learner asks if a hypothesized Mealy machine $\mathcal{H}$ with inputs I and outputs O is correct, that is, whether $\mathcal{H}$ and $\mathcal{M}$ are equivalent. The teacher answers* yes *if this is the case. Otherwise she answers* no *and supplies a counterexample $\sigma \in I^*$ that distinguishes $\mathcal{H}$ and $\mathcal{M}$.*"

Using Angluin's L^* algorithm [3], the learner can then incrementally construct the state diagram by repeatedly sending these membership and equivalence queries to the teacher [39]. As we only use the L^* algorithm as a tool in this work, we refer the reader to [39] for more details.

2.2.3 State identification

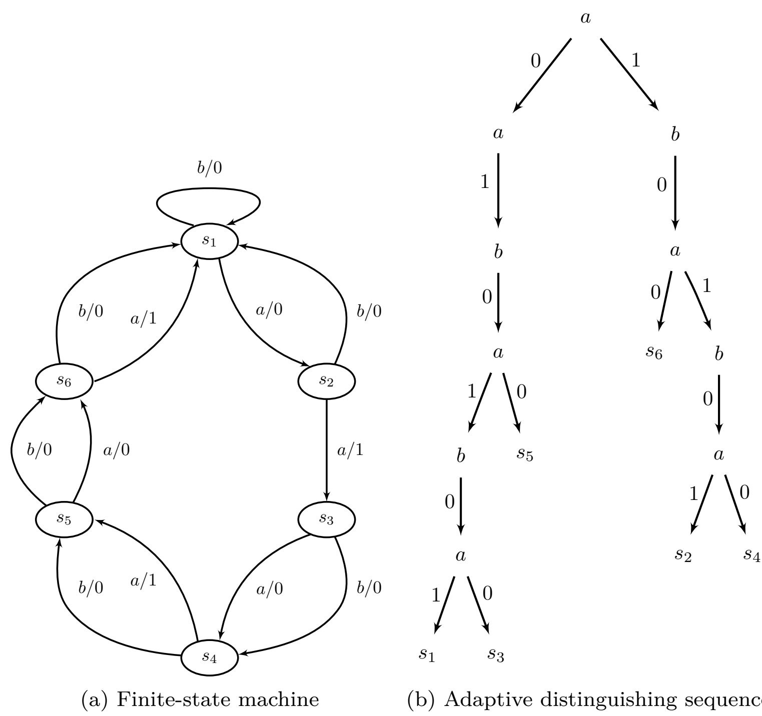
State machines are used to model systems in a wide array of fields, and there is an entire field of research focused on analyzing them and verifying their correctness. One of the fundamental problems in this field is that of *state identification*: given a finite state machine $\mathcal{M}$ that is fully known – except for its initial state – identify the unknown initial state q_0 (note that this is not always possible). An input sequence that solves this problem, if it exists, is called a *distinguishing sequence* [20, 24].

A distinguishing sequence can be *preset* (the input sequence is fixed ahead of time) or *adaptive* (the input symbols depend on the observed output) [20]. Adaptive distinguishing sequences (ADS) are more general than preset distinguishing sequences; a machine might have an adaptive distinguishing sequence, but not a preset one [21]. We will therefore only discuss adaptive distinguishing sequences. Note that since the inputs for an adaptive distinguishing sequence depend on the observed outputs, it is technically a decision tree and not a sequence.

There is a well known algorithm for computing adaptive distinguishing sequences for finite state machines published by Lee & Yannakakis in 1994 [21]. In their paper, they provide the following definition for an adaptive distinguishing sequence (notation modified to match Section 2.2.1):

> "*An* adaptive distinguishing sequence *is a rooted tree T with exactly n leaves; the internal nodes are labeled with input symbols, the edges are labeled with output symbols, and the leaves are labeled with states of the FSM such that: 1) edges emanating from a common node have distinct output symbols, and 2) for every leaf of T, if x, y are the input and output strings respectively formed by the node and edge labels on the path from the root to the leaf, and if the leaf is labeled by state q_i of the FSM then $y = \lambda(q_i, x)$. The length of the sequence is the depth of the tree.*"

Figure 2.6 shows a finite state machine alongside the adaptive distinguishing sequence constructed using the algorithm from [21]. The ADS can then be used to solve the state identification problem for this state machine. To find the initial state, start at the top of the ADS, which tells you to provide input a to the machine. If the output is 0, go down the left branch and input another a, if the output 1 then take the right branch instead and input b next. Repeating this process and moving further down, eventually brings you to a leaf node. This leaf node then tells you in which state you started.



(a) Finite-state machine (b) Adaptive distinguishing sequence

Figure 2.6: Example of an adaptive distinguishing sequence for a finite-state machine [21]

Chapter 3

Related work

In this section we discuss relevant related work conducted in the fields of model learning, fingerprinting TLS, and formal fingerprinting.

3.1 Model learning

Model learning has been applied to various security protocols in order to find vulnerabilities and standards violations, including OpenVPN [10], SSH [13] and also TLS [28, 29]. The results of these studies include finding standard violations in three major implementations of the SSH protocol [13] and multiple vulnerabilities in major implementations of the TLS protocol [28, 29].

One of the things that can be noted from these related works, is that two implementations of the same protocol (and even two versions of the same implementation) do not necessarily yield the exact same model when model learning is performed. There can be input sequences that result in different outputs for different implementation. This is especially likely when inputs are sent in an unusual order, one that deviates from a standard protocol run. This has been observed for implementations of different protocols, including TLS. During this thesis, we collaborated with one of the authors of [29] and built upon their tools in our learning pipeline. This is described in more detail in Section 6.2.

The fact that different implementations can exhibit different behavior when supplied with the same inputs is a key property exploited in this thesis. It implies that the learned model of an implementation can serve as a fingerprint. If two implementations have a different model, we could therefore provide an input sequence to distinguish these implementations from each other. Following this reasoning, given a set of implementations for which we have learned the models, then given a random implementation from this set we could use these distinguishing input sequences to match the implementation to one of the models, effectively identifying the implementation.

3.2 Fingerprinting TLS

Since TLS is one of the most widely used security protocols on the Internet, it has been subjected to a lot of analysis over the years, including different kinds of fingerprinting: traffic analysis, identification of client and server applications, identification of specific implementations, etc. The most common, and this is often referred to as simply “fingerprinting TLS,” is the identification of client (and in lesser extent server) applications by creating a fingerprint of

their TLS usage. This is not the kind of fingerprinting we perform here, but we will briefly discuss it in order to highlight the differences.

3.2.1 Fingerprinting TLS usage of client and server applications

The primary drive behind identifying applications using TLS is to detect malware and malicious connections in a (trusted) network. These threats make increasing use of TLS to encrypt their traffic, so deep packet inspection is no longer a viable method to detect them. The approach therefore emerged to identify these applications based on their usage of TLS, specifically the contents of the `ClientHello`. This initial message is not yet encrypted, and its values can be sufficiently unique to build a fingerprint for a given client application [16, 35].

When a fingerprint is confirmed to belong to an application, it can be stored in a fingerprint database. For each subsequently observed `ClientHello`, the fingerprint can then be matched with this database in order to identify the application that sent it. This method has been applied successfully to detect (among others) malware, censorship circumvention tools and web browsers [14, 16, 36]. The most well known tools that implement this method are JA3 and JA3S [36]. The latter extends this method to create a fingerprint of TLS servers based on the received `ClientHello` and the resulting `ServerHello`.

To summarize, this approach focuses on the client and server applications and their behavior, the desired outcome is to identify a client as "Malware version X," or "Web browser version Y." The specific underlying TLS implementation used isn't relevant here.

3.2.2 Fingerprinting TLS server implementations

In contrast to the previous approach to fingerprinting, our focus is figuring out which specific implementation is used to provide the TLS connection (e.g. OpenSSL 1.0.1, or mbed TLS 1.14.3). It is not our goal to identify the higher level applications themselves.

When it comes to identifying protocol implementations, the common approach is sending input sequences and observing the outputs [32]. For the TLS protocol there is `tls_prober` [37], which also aims to identify TLS server implementations based on a fingerprint of their behavior. `tls_prober` defines a list of about 295 different hand-picked test sequences, which are chosen as "*likely to find edge cases in the implementations*" [37]. A fingerprint is created by sending all sequences to the target server and storing the results. Identification has a "thorough" and a "quick" option. The thorough option sends all 295 test sequences to the target, the quick option sends less but is also less accurate according to the creators. The output is a list of possible implementations, sorted by the percentage of matching probes.

`tls_prober` makes use of hand-picked sequences to find differences between implementations, we take a more generalized approach. We will first apply model learning to map all possible behavior of an implementation, so we can then compute which input sequences yield the most distinguishing information. As a result we sent fewer inputs, but our identification will be a "yes or no" as it either matches one of the models we learned, or it doesn't. A partial match might be possible as a subject of future work.

In our current implementation, we only look at the message types and their order. `tls_prober` also does this, but in addition it also looks at how the messages are structured in the TLS record layer. This is something that could be applied to our implementation in future work.

3.3 Formal fingerprint matching

In the Section 3.1 we mentioned that the learned model of an implementation can serve as a fingerprint, and that certain input sequences might be used to perform fingerprint matching. The use of output differences to identify different implementations of the same protocol is not

new and has been applied successfully to other protocols. The most well known example is Nmap, a tool that can distinguishing between thousands of different operating systems using TCP/IP input sequences [9].

The shortcoming of existing methods is that these sequences are usually not automatically inferred but manually selected, which requires expert knowledge of both the protocol and the implementations. This is also the case for `tls_prober`, as mentioned in Section 3.2. In order to distinguish between more implementations and to find the shortest possible sequences to do so, it can be beneficial to leverage formal methods.

A formal methodology for network protocol fingerprinting is discussed in [32]. In this paper the authors[1] present an extension to the FSM model to better model communication protocols, called the Parameterized Extended Finite-State Machine (PEFSM). The PEFSM model extends the FSM model in a number of ways: including variables as part of the state, adding conditions and actions to transitions, and adding parameterized input and output symbols. The resulting model has much more descriptive power, but it also requires that a specification is available in order to perform fingerprint discovery.

Using the PEFSM model, the authors present a taxonomy of fingerprinting problems for both passive and active experiments [32]. For completeness, this taxonomy is shown in Table 3.1.

Table 3.1: Taxonomy of network fingerprinting problems [32]

	Active experiment	Passive experiment
Fingerprint matching	PEFSM conformance testing	PEFSM passive testing
Fingerprint group matching	Online machine enumeration	Concurrent passive testing
Fingerprint discovery with spec	Machine enumeration and separation	Back-tracking based passive testing
Fingerprint discovery without spec	FSM supervised learning	No efficient solution

In this taxonomy, our use case can be classified as an active experiment, specifically "fingerprint group matching" (matching an implementation with multiple fingerprint simultaneously) and "fingerprint discovery without spec" (as we will apply black-box testing). Unfortunately, the authors of [32] already noted that "*it is, in general, impossible to recover model constructs in PEFSM related to protocol design (i.e., predicates and actions) from black box implementation.*" They conclude that in the black-box scenario "FSM supervised learning" [33] is the best known approach, which is another name for model learning which we covered in Section 2.2.2. Since the algorithm presented for fingerprint group matching is specific to the PEFSM model, it cannot be directly applied here. We will discuss alternatives in Chapter 7.

Other related work focuses on applying a range of formal methods to improve performance of an existing fingerprinting approach. A popular subject is OS fingerprinting and identification, often with Nmap referenced as a baseline. In [15], the authors evaluate the "information gain" of each Nmap probe in order reduce the number of packets required for an identification, while maintaining the same accuracy. Another approach is presented by the authors of Hershel, a tool for single-packet OS fingerprinting [31], who note that the large number of packets required by Nmap is problematic for Internet-wide use. Their alternative approach uses only one TCP SYN packet to extract the input for its classification, at the cost of being less specific.

Since fingerprinting can be done with malicious intent, there is also work performed to counter these techniques. Two examples are *protocol scrubbers*, and what is often called the *moving*

[1] It's worth noting that one of the authors of this paper is David Lee, who is also an author of the algorithm for adaptive distinguishing sequences, which we discuss in Section 2.2.3 and Section 7.3.2.

target defense. Protocol scrubbers are software components designed to normalize the network traffic for a given protocol by removing superfluous values and reducing ambiguities, reducing the ability to identify the target [41]. The moving target defense aims to present an outside observer with an inaccurate view of the system in order to confuse potential attackers [22, 43]; in this context "system" can refer to a single application as well as an entire network. Presenting this different view can be achieved in multiple ways such as modifying packet headers to mimic other implementations [2], or continuously change IP addresses and routes [8, 19]. In this thesis we focus on developing our fingerprinting and identification techniques in an unobstructed setting, and consider evasion techniques to be out of scope.

Chapter 4

Solution overview

In the previous chapters, we have covered the preliminaries and the related previous work. In the rest of this thesis, we will describe the design and implementation of our solution for automated fingerprint extraction and identification of TLS server implementations in detail. This chapter will provide an overview of the entire solution architecture and some general implementation details. The following chapters will focus on the individual components.

4.1 Target protocol versions

As discussed in Section 2.1, there are currently four versions of the TLS protocol available. The most recent version, TLS 1.3, has modified the handshake sequence extensively. Because this new handshake is not compatible with earlier versions, research and tools tailored to TLS 1.2 and earlier cannot be used without modification. Since TLS 1.3 is relatively new, the number of implementations supporting it is small in comparison to all versions supporting TLS 1.2 and earlier. Our focus is on providing a proof of concept for fingerprinting and identifying TLS implementations leveraging existing formal methods, not analyzing the TLS 1.3 implementations in detail. Therefore we will not include TLS 1.3 in our learning setup, but only look at TLS versions 1.0, 1.1, and 1.2. We expect that, with updated tooling, our approach will also be applicable to TLS 1.3.

4.2 Architecture

We split up the process of automated fingerprint extraction and identification in multiple stages, which will be described in more detail in the following chapters:

- **Build** (Chapter 5): In order to analyze the TLS server implementations, they must first be compiled, packaged, and stored for later use. We set up an automated pipeline to build all versions of the target TLS implementations and publish the build artifacts to a public location.
- **Learn** (Chapter 6): After building the TLS server implementations, models can be learned, for which we set up a second pipeline. This pipeline uses the available build artifacts published by the build stage, learns a model for all TLS versions supported by each implementation, and also publishes these models to a public location. These models serve as fingerprints of the TLS implementations.
- **Identify** (Chapter 7): When all models are learned, they can be used to identify a TLS implementation by matching it with its fingerprint. We approached fingerprint matching as a state identification problem and applied two methods for computing distinguishing sequences: *adaptive distinguishing graph* (ADG) and *heuristic decision*

tree (HDT). The ADG is a direct generalization of Lee & Yannakakis' algorithm for adaptive distinguishing sequences and the HDT is a new method we present here. A Python package with the name `tlsprint` was developed to perform the identification with either method.

We designed these stages in such a way, that each stage only depends on the output of the previous stage (visualized in Figure 4.1), which creates a loose coupling between the different stages. As a result, different stages can be executed and developed independently of each other, which makes the entire process more flexible.

Figure 4.1: General flow of stages

To keep our solution modular, we designed an architecture consisting of multiple components, where each component is responsible for a (part of a) stage. An overview of all components and their relationships is given in Figure 4.2; the solid arrows indicate a dependency, the dashed arrows indicate that a component is derived from another component. Each component has an owner and a name (the format is `owner/name`), where the `tlsprint` owner refers to this project. In the following chapters, the relevant components for each stage will be discussed separately.

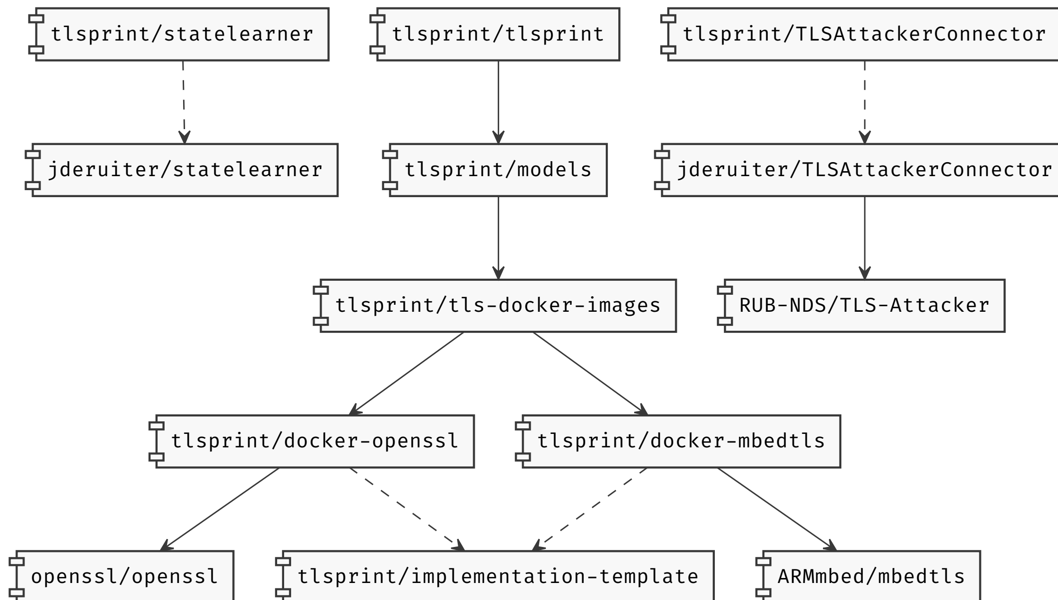


Figure 4.2: Overview of all components

The different components, just like the stages, are set up in loosely coupled way, with no dependencies on the internals of other components. Two components performing the same task can be therefore be used interchangeably, as long as they both have the same interface. We use this mechanism in the *build* stage to supply implementation specific build configuration, in the *learn* stage to create an implementation independent learning pipeline, and in the *identify* stage to allow two different methods for fingerprint matching.

4.3 Implementation details

In this section, we mention some concrete tools and technologies we have used across all stages. This is primarily relevant for those we want to gain a better of the understanding of the internals of the solution.

The components shown in Figure 4.2 are managed with Git[1] repositories, where each component is stored in a separate repository. All repositories are hosted publicly on GitHub[2] where the name corresponds directly with the GitHub namespace. Most of the `tlsprint` repositories are also hosted on a private GitLab[3] instance. The dependencies between the different repositories are handled using Git submodules.

The build artifacts from the build stage are Docker[4] images. Docker is a container technology, a type of OS-level vitalization, which can be seen as a light-weight virtual machine. Docker allows software, along with its dependencies and configuration files, to be packaged in a *Docker image*. These images can then be executed on any machine with Docker installed. To make the Docker images publicly available, we upload them to Docker Hub[5], a repository where public Docker images can be distributed freely.

For the tools and scripts we developed, we used Python[6], a flexible high-level general purpose language. For various tasks in our pipelines we created template files to automatically generate other files. These templates make use of Jinja[7], a widely used templating language for Python.

[1] https://git-scm.com/
[2] https://github.com/
[3] https://about.gitlab.com/
[4] https://www.docker.com/
[5] https://hub.docker.com/
[6] https://www.python.org/
[7] https://jinja.palletsprojects.com/

Chapter 5

Building the implementations

Before any model learning can take place, the target implementations must be build, configured and tested. This the *build stage* from Section 4.2. Given the large number of implementations we are dealing with, setting this up is a time-consuming process. These steps were also required in previous TLS research, but it turns out that the wheel is largely reinvented each time: the specific pitfalls of compiling old and unsupported software are usually not part of the research results, the scripts that are used are often specific to the machine of the researcher, and the build artifacts (the compiled and packaged binaries) are not published.

As a first step in automating the fingerprint procedure, we set up the necessary infrastructure to continuously build as many TLS implementation versions as possible in a clear, portable, and automated manner. The build artifacts are published to a public registry and newly released versions of implementations are automatically included in this process. While these build artifacts are primarily made for the purpose of fingerprint extraction within this project, future research can also directly use the artifacts, and not be bothered by the plumbing work that is required to compile the software.

When building this many implementations and testing if they all function as expected, it quickly becomes hard to keep track of what is happening. To solve this, we designed a pipeline with the following properties:

- Perform every build in an isolated environment. This is to improve portability and prevent the "works on my machine" problem.
- Provide a clear overview of the status of every build: was the build successful or did it fail? What was the output generated during compiling and verification? Multiple stages (build, verify, publish) should be easily distinguishable.
- Process multiple versions in parallel to speed up the process.
- Failure processing one version, should not halt the entire pipeline. These failures must be clearly marked for troubleshooting at a later point, but all versions should be processed.
- Results should be cached to speed up subsequent pipeline runs, as there is no need to recompile a working implementation. It must also be possible to force a recompilation by removing the cache.
- It should be easy to add a new TLS implementation to the pipeline.

A schematic overview of this pipeline is shown in Figure 5.1. Because some desired properties overlap with those of continuous integration and delivery (CI/CD) tools used in software development, we used a CI/CD platform as the basis for the pipeline. We evaluated multiple

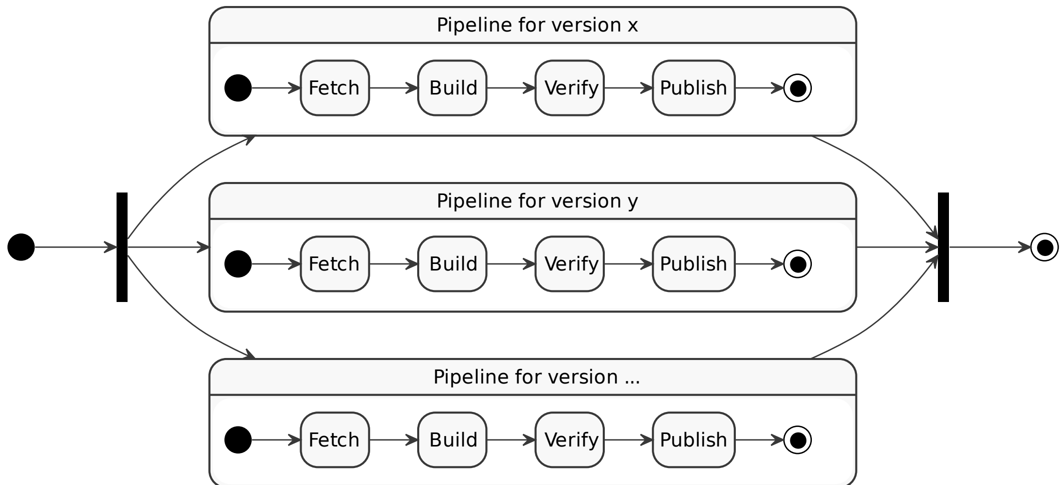


Figure 5.1: Pipeline design

of these tools in order to determine which one fitted best: Travis CI[1], Circle CI[2], Jenkins[3], GitLab[4], and Drone[5]. Eventually we went for a combination of GitLab and Drone. Both are very flexible, but Drone was the only one offering truly parallel builds with multiple stages each. GitLab was also used for source control management, which provides a convenient integration, and provides betters support for scheduled builds compared to Drone.

In the realized build stage, multiple components from Figure 4.2 are involved, those are shown again in Figure 5.2. In the following sections we will explain the role of each component in more detail.

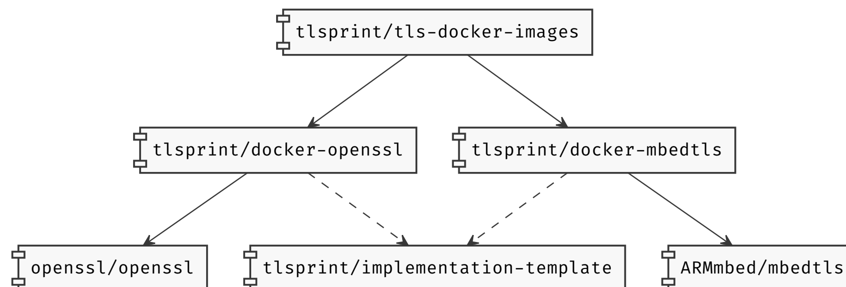


Figure 5.2: Components used for the build stage

5.1 Build manager

For each implementation we analyze, there are two components: the upstream code of the implementation developer, and our own build component. For each build component we set up a dedicated pipeline using Drone as designed in Figure 5.1 (we shall describe this in more detail in Section 5.2). These builds components are managed by the `tlsprint/tls-docker-images` component, which we call the *build manager*. The build manager configures the pipeline for

[1]https://travis-ci.com/
[2]https://circleci.com/
[3]https://jenkins.io/
[4]https://about.gitlab.com/product/continuous-integration/
[5]https://drone.io/

these implementation specific build components, in order to maintain consistency across the different implementations. Specific build settings (compiler flags, etc) are implementation specific, and are delegated to the individual build components themselves.

5.1.1 Overview

Setting up pipelines to build, verify and publish many TLS server implementations is useful, but if it is a one-off event, it will quickly be outdated as new versions of TLS implementations are released. The build manager therefore ensures that the pipeline configuration of each build component always includes the latest versions, by performing a daily update. For each TLS implementation, this daily update queries the upstream component for new versions and triggers the build pipeline with the updated configuration. This pipeline will then publish the build artifacts of these new versions in the form of Docker images. This process is shown in Figure 5.3.

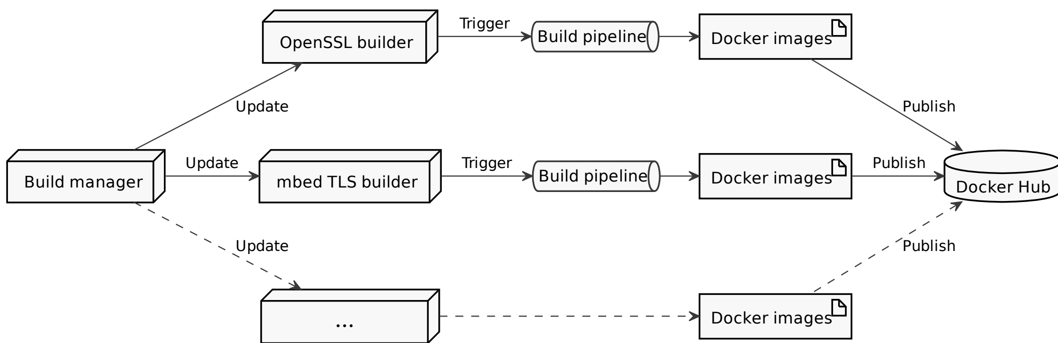


Figure 5.3: Overview of the periodic build

Because each build component is running its own pipeline independently of the others, they are displayed separately in the Drone dashboard. This results in an overview of the general build status of the individual implementations, as seen in the screenshot in Figure 5.4.

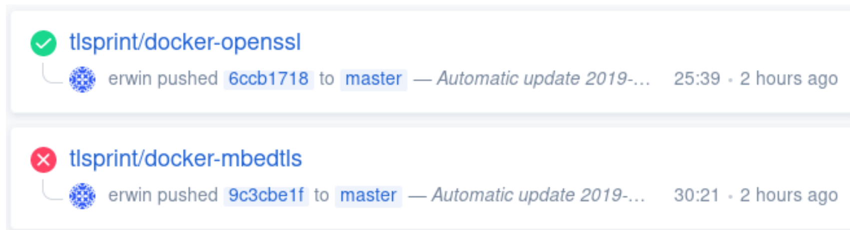


Figure 5.4: Different repositories on Drone

5.1.2 Details

The build manager is hosted as a Git repository called `tlsprint/tls-docker-images` on a private GitLab instance and mirrored to GitHub[6]. This repository contains the implementation specific repositories as submodules (more details in Section 5.2), and an update script written in Python. Each build component can be imported as a Python module with two hook functions (`extract_versions` and `get_supported_tls`), which are used by the update script. A GitLab CI pipeline is scheduled to run the update script on a daily basis. For each implementation specific repository referenced in this manager repository, the update script performs the following actions:

[6]https://github.com/tlsprint/tls-docker-images

- **Retrieve a list of all release tags**. For the implementations we analyzed so far, information about releases can be retrieved from the upstream Git repository in the form of *tags*. As such, these build repositories contain a reference to the upstream code as a submodule. The update script pulls the most recent version of the `master` branch of the upstream, and reads a list of all tags pushed to this repository. In case Git tags are not available when adding support for a new implementation, this can be replaced by crawling a release page, parsing the change log, or something else. As long as the result is a list of tags, the rest of the script can continue.

- **Filter tags**. Not all tags published by the upstream are suitable for inclusion in the build and publish pipeline. Examples are pre-releases and releases which cannot be build because they rely on tooling which is no longer available. These tags are filtered from the list.

- **Extract version numbers**. The tags published in Git often include more than just the version number, which makes them difficult to use in comparisons. For example, mbed TLS prefixes their tags with `mbedtls-` or `polarssl-` (the old name of this library); this should be removed to get the actual version number. Both this step and the previous filter step are handled by the build component through the `extract_versions` handler function.

- **Query supported TLS versions for each implementation**. Not all implementations support all versions of the TLS protocol. Each build component therefore exposes a `get_supported_tls` function, which will return a list of supported TLS versions based on the version number of the implementation.

- **Generate dockerfiles**. For each version of an implementation a separate `Dockerfile` is generated, from which the Docker image can be build. This file contains all commands and configuration required to build and run that version. Because these build steps are specific to a particular implementation, each build repository has a template `Dockerfile.j2`, which the update script uses to generate the dockerfiles for that implementation.

- **Generate `.drone.yml` pipeline configuration**. Drone is configured through a file called `.drone.yml`. The build manager contains a template file, `.drone.yml.j2`, which is used to generate the pipeline configuration of each build repository. A single template is used to keep the build, verification and publish steps consistent across the different implementations. This template takes as input the name of the implementation, the raw tags, the cleaned version numbers, and the supported TLS versions of each implementation version. The resulting pipeline configuration is then given an entry for each implementation version, including a verification step (in the form of a standard handshake) for each supported TLS version.

- **Push changes to the build repository**. If the update script changes any files (updated submodule, new dockerfiles, changed pipeline configuration), these are collected in a Git commit, and pushed to the build repository. This push then triggers the Drone pipeline for this implementation, which will result in new versions being published to Docker Hub.

After all build repositories are updated, the submodule references in this manager repository are no longer up to date. As a final step, these updated references are also committed and pushed back to the build manager.

5.2 Build components

There are two components for every TLS implementation we include in our build pipeline. The first component, the upstream code from the developer of this implementation, is not

under our control, but we rely on it for the build and update process. The second component is our own build component, specifically configured for a given TLS implementation, which will be the topic of this section.

5.2.1 Overview

The build component is responsible for building, verifying and publishing build artifacts of many versions of a given TLS implementation. It is self-contained, in the sense that all the files and configurations required to build this implementation, are stored in this component; this includes the pipeline configuration. Any changes to either the build files or the pipeline configuration will trigger the pipeline to run.

The pipeline configuration, while generated by the build manager as mentioned in Section 5.1, is specific to the individual build component. The pipeline consists of multiple sub-pipelines (one for every version of the implementation) that can be executed in parallel. The pipeline is executed by Drone and follows the design from Figure 5.1. Figure 5.5 shows a segment of the pipeline dashboard for successful and failed pipelines, expanded on a single version. As can be seen, all pipeline steps are planned for every version, and errors do not prevent other versions from running (in Figure 5.5b, version 1.3.4 successfully completes, even though the earlier 1.3.3 fails). The pipeline currently performs the following actions for every implementation version.

- **Clone**: In this step the code of this component is fetched by Drone, and the configuration for this sub-pipeline is loaded.
- **Build**: The Docker image for the specific version is build. This involves fetching the correct version of the source code, installing dependencies and invoking the right compiler commands. The resulting binaries are placed in a second, clean Docker image (using a multistage Docker build), so the build tools won't pollute the test environment later.
- **SUT**: Start the implementation that has just been build (called SUT, short for *system under test*) in the background, with the TLS server listening, in preparation of the verification stage.
- **Verify**: The implementation is verified by checking if a standard TLS handshake is possible. Most implementations support multiple versions of the TLS protocol; these will all be verified separately. This step is necessary as it can happen that the build is successful, but the SUT cannot complete a normal handshake. This can be caused by configuration errors, or faults in the implementation itself.
- **Publish**: If the implementation has been successfully verified, the Docker image will be uploaded to Docker Hub, to be publicly available.

After every version of the TLS implementation has been processed, the pipeline ends and the final status is displayed on the Drone dashboard as shown in Figure 5.4.

5.2.2 Details

Each implementation specific build component has a Git repository (`tlsprint/docker-openssl`[7] for OpenSSL, `tlsprint/docker-mbedtls`[8] for mbed TLS, etc.) in which all the build and configuration files are stored. Most of the files in these repositories are generated by the build manager, using implementation specific templates. These components have a largely identical directory structure, they contain at least the following files and directories:

- `upstream`: This directory is a Git submodule referencing the upstream repository, the source code of the actual implementation. For example, for OpenSSL this points

[7] https://github.com/tlsprint/docker-openssl
[8] https://github.com/tlsprint/docker-mbedtls

(a) Sucessful pipeline (b) Pipeline with failures

Figure 5.5: Pipelines in the Drone dashboard

to https://github.com/openssl/openssl.git and for mbed TLS it points to https://github.com/ARMmbed/mbedtls.git.

- `dockerfiles`: For each version of this implementation, this directory contains a `Dockerfile`. This `Dockerfile` can be used to easily build the Docker image for that specific version by running `docker build dockerfiles/$VERSION`. Each `Dockerfile` is self-contained, which allows the inspection of the specific build steps, and allows building either a single specific version, or multiple versions simultaneously.

- `Dockerfile.j2`: From this template, all dockerfiles in this repository are generated. The template takes some values (such as upstream URL, version, tag, etc.), and contains the logic which results in the eventual `Dockerfile`. This logic largely includes running certain commands, or setting different values, based on the target version.

  For example, pulling the source code of a specific version:

```
RUN git clone --branch {{ tag }} --depth 1 {{ url }} .
```

  Or including an extra flag based on the version to build:

```
{% if version < "0.9.7" %}
    RUN ./config -fPIC no-asm
{% else %}
    RUN ./config -fPIC no-asm zlib
{% endif %}
```

  By placing all the logic in this single template file, it creates a single source of truth about how the TLS implementation should be compiled for different versions.

- `__init__.py`: This file turns the directory into a Python package with the handler functions described in Section 5.1.2. These handler functions are called by the update scripts of the build manager and learn manager (Section 6.1).

- `.drone.yml`: This file defines the configuration for Drone CI, it is generated using the `.drone.yml.j2` template file from the build manager.

To store the artifacts, each build component is linked to a repository on Docker Hub under the `tlsprint` namespace: images for OpenSSL are stored at `/r/tlsprint/openssl`[9] and for mbed TLS at `/r/tlsprint/mbedtls`[10]. Specific versions are marked with *image tags*, which makes it easy to download any given version. For example, OpenSSL version 1.1.1c can be downloaded by running `docker pull tlsprint/openssl:1.1.1c`.

5.3 Adding a new implementation

To add a new implementation to the build cycle, the necessary components must be created and added to the build manager. The build manager then includes this implementation in the periodic update.

5.3.1 Overview

When adding the components for a new TLS implementation, the directory structure as discussed in Section 5.2.2 is very important. It is however tedious to create it from scratch every time. To ease this process, we have created a project template from which these build components can easily be created. The template takes some variables, such as the implementation name, to create a scaffold project, which includes the necessary boilerplate and a link with the upstream component. The user can then add additional implementation specific code and configuration. Once completed, this new component must be linked to Docker Hub and Drone, and can then be added to the build manager.

5.3.2 Details

The template is stored on GitHub as `tlsprint/implementation-template`[11]. It leverages Cookiecutter[12], a project template framework. After installing Cookiecutter (for installing Cookiecutter we refer the reader to the documentation of the project[13]), a new build component can be added by running `cookiecutter https://github.com/tlsprint/implementation-template` and filling in the prompted values. This will create the necessary files with some stub content, initialize a Git repository, add the upstream repository as a submodule and create an initial commit of it all. An example run is shown below:

```
~> cookiecutter https://github.com/tlsprint/implementation-template

implementation_name [some-tls]: OpenSSL
implementation_slug [openssl]:
upstream_url [https://github.com/openssl/openssl.git]:
Initialized empty Git repository in /home/tlsprint/docker-openssl/.git/
Cloning into '/home/tlsprint/docker-openssl/upstream'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 337806 (delta 0), reused 1 (delta 0), pack-reused 337803
Receiving objects: 100% (337806/337806), 169.90 MiB | 17.24 MiB/s, done.
Resolving deltas: 100% (231462/231462), done.
```

[9]https://hub.docker.com/r/tlsprint/openssl
[10]https://hub.docker.com/r/tlsprint/mbedtls
[11]https://github.com/tlsprint/implementation-template
[12]https://cookiecutter.readthedocs.io/
[13]https://cookiecutter.readthedocs.io/en/latest/installation.html

```
[master (root-commit) 7d5693d] Initialize repository for OpenSSL
 4 files changed, 58 insertions(+)
 create mode 100644 .gitmodules
 create mode 100644 Dockerfile.j2
 create mode 100644 __init__.py
 create mode 160000 upstream
```

In this example run, the user wants to add a repository for `OpenSSL`. The template infers that `openssl` is a good short name to use in directories, and guesses that the source can be found on GitHub where the user and the repository are both called `openssl`. In this case this is correct and the default can be accepted, but a different URL can be passed by the user. The result is an initialized Git repository in a new `docker-openssl` directory. The user can then modify the stub files where needed.

5.4 Discussion

The build pipeline is designed and implemented to operate autonomously. The durability of such a solution depends on the different components and tools, and how they are connected. It is therefore worthwhile to discuss how future-proof this setup is in terms of required manual intervention and maintenance, and how the setup can be improved.

5.4.1 New versions

Most of the time, newly released implementation versions can automatically be build and published by the build pipeline. Sometimes however, the build process is changed by the upstream maintainers in such a way that human intervention is required. Examples are changes to dependencies, build commands, compiler flags, project structure, etc. For these changes, new checks must be added to the `Dockerfile.j2` of the matching build component. It depends on the maintainers of the individual implementation how often this is required, as some maintainers care more about a stable build interface then others.

5.4.2 Maintenance

For the build pipeline, Docker and Drone are the most important tools. Drone provides two deployment options: cloud and self-hosted; both are free for open source projects. Unfortunately, we could not use the Drone Cloud for our build pipeline. In particular, we need to build a Docker image and, at a later stage, start this same Docker image in order to verify a particular TLS implementation. This means that the Docker image must be stored somewhere, which is, at the time of writing, not supported by the cloud hosted version of Drone. The self-hosted version allows this, by binding the Docker process running in Drone to Docker running on the host system. This does breaks the isolation of individual build pipelines, as the Docker images will now be persistently stored on the host.

Using the self-hosted version of Drone means that we also have to manage this instance. Not a lot of maintenance is necessary, except that logs and unused intermediate images are not removed automatically. Drone keeps track of the logs of all jobs it ever executed, and stores these in an SQLite database. At some point, this will consume quite some storage. Unfortunately, Drone does not provide a convenient way to cleanup the database, so this must be done manually. For more information, check the following GitHub issues:

- https://github.com/drone/drone/issues/1694
- https://github.com/drone/docs/issues/238
- https://github.com/drone/drone-cli/pull/83

A side effect of storing the images on the host, is that this caches intermediate layers when building the images, which speeds up the build process considerably when building the same image a second time. This also means that a lot of Docker images and intermediate layers are stored on the host machine. This will have to be regularly cleaned if storage is limited. During our tests we used a 200GB partition to store the Docker layers, which filled up from time to time. We then had to clean this by running `docker images prune`, but this was only a temporary solution. Expanding storage could resolve this issue, but it might also be beneficial to automatically call the `prune` command periodically.

5.4.3 Future work

A first target for future work is to add more TLS implementations the pipeline. These could be standalone libraries such as Botan, BoringSSL or LibreSSL, but also programming language specific implementations such as the `crypto/tls` package in Go's standard library. In some cases, sample applications may not be provided, or sufficient, in which case these should also be added.

Cases in which the pipeline fail (such as a new version which cannot be processed) are not yet reported to the maintainer. This means the pipeline cannot be left completely unattended, because checking for errors should still happen. Ideally, the maintainer of the pipeline will receive a periodic (daily) overview of which implementations are processed successfully, and for which errors have occurred.

The build pipeline relies on various tools, technologies, and platforms: Git, GitLab, GitHub, Docker, Docker Hub, Drone, Python. The assumptions is that these will be available for the foreseeable future, but this might change. In case of new technologies or disappearing platforms, the build pipeline might have to be modified to remain relevant.

Chapter 6

Automated learning

After all implementation versions have been build, and their artifacts published, model learning can begin. This is the *learn stage* from Section 4.2. Just as for the build stage, we set up a pipeline for this task. This pipeline learns the models of all TLS server implementations published by the build stage, and stores the results. The setup for the learning pipeline is similar to the build pipeline: there is a learn manager, parallel sub-pipelines, and a periodic trigger to make sure models of new implementations are learned. The components involved in this pipeline are shown in Figure 6.1. In the following sections the role of each component will be explained in more detail, starting with the learn manager.

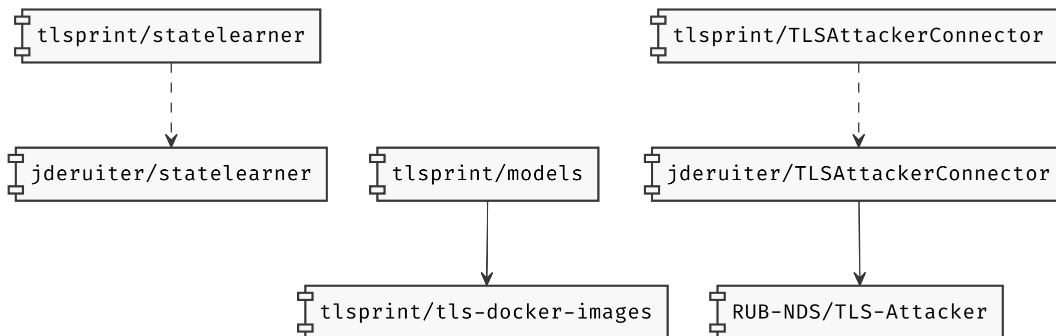


Figure 6.1: Components used for the learn stage

6.1 Learn manager

The learning pipeline is centered around the `tlsprint/models`[1] component, which we call the *learn manager*. It is responsible for configuring and running the pipeline on a regular basis, and storing the learned models.

6.1.1 Overview

The learn manager configures and executes the entire learning pipeline. This is in contrast to the build manager, which delegates pipeline execution to other components. The reason for this, is that the learning pipeline is simpler than the pipeline for building; whereas the build procedure is different for every TLS implementation, the learning setup (described in

[1] https://github.com/tlsprint/models

Section 6.2) is identical. This is necessary, because we use a black box learning algorithm, which is implementation agnostic. The different build artifacts therefore all conform to the same interface, they expose a TLS server, so there are no implementation specific variables or settings required for learning.

When learning the model of an implementation, a TLS protocol version must be picked. Instead of focusing only on the most recent version of the TLS protocol supported by an implementation, we look multiple at TLS versions. In our experiments we noticed that specifying different TLS versions can lead to different models for the same implementation version. This results in additional distinguishing information, which can potentially be used during the identification phase.

A daily update ensures that the models for new TLS implementations will be learned and stored, in order to stay up-to-date. The update script starts by querying the available implementation versions published by the build stage, and the TLS versions supported by each implementation version. Because model learning can be a time consuming process, we do not want to relearn the models of implementation for which a model is already stored. Only the configurations (implementation and TLS version combinations) for which a model is missing will be scheduled for learning. The learning pipeline will then be triggered to initiate learning the scheduled configurations. This process is visualized in Figure 6.2.

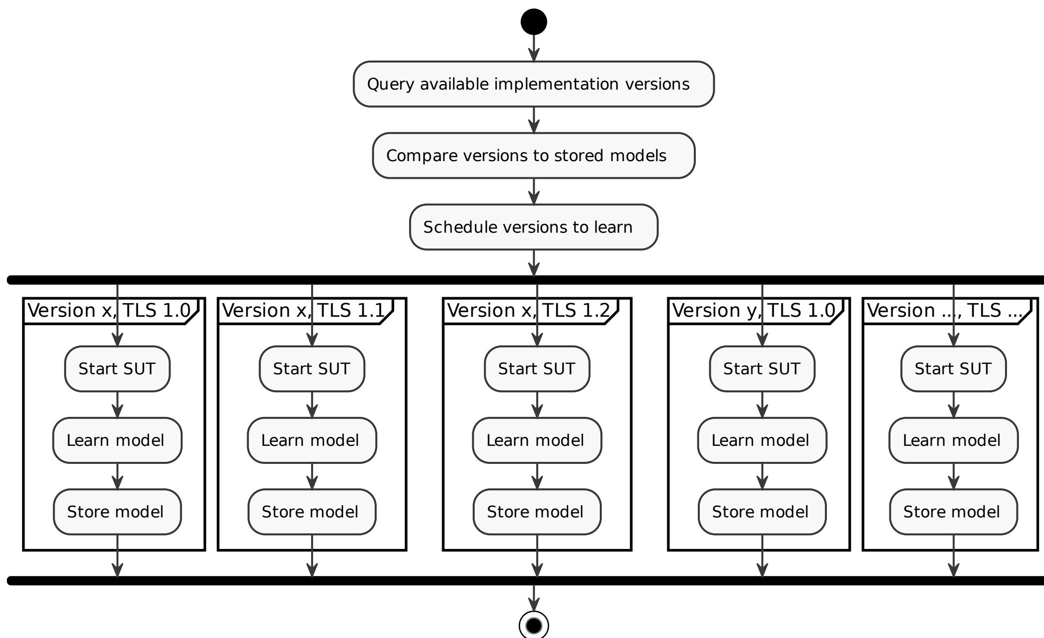


Figure 6.2: Pipeline for learning

6.1.2 Details

The Git repository for the learn manager can be found on GitHub[2]. It contains the configuration of the learning pipeline, the update scripts, and a reference to the build manager. Most importantly, it also stores all the models that are the result of learning the different implementations.

This component periodically updates and triggers the learning pipeline. This is done through a GitLab CI pipeline, which is scheduled to run the update script on a daily basis. This

[2]https://github.com/tlsprint/models

update scripts performs the following actions:

- **Query available TLS implementation versions**. The learn manager is linked to the build manager (using a Git submodule), so it can construct a list of supported TLS implementations. For every TLS implementation, the update script then queries the Docker registry for the available image tags, which in our case are the version numbers of the different implementations as published by the build pipeline.
- **Query supported TLS versions for each implementation**. As the learn manager is linked to the build manager, it can also call the handler functions in the individual build components. By using the same `get_supported_tls` handler as used in the build setup, the learn manager can query the supported TLS versions for every image available on the Docker registry.
- **List stored models**. All learned models are stored in the `models` directory in the learn manager. The implementation version and TLS version used to learn each model is encoded in the path where the model is stored, with the following format:

  ```
  models/$IMPLEMENTATION/$VERSION/$TLS_VERSION/learnedModel.dot
  ```

  By reading the contents of the models directory, the update script can create a list of the configurations (implementation and TLS version combinations) for which a model is already learned.
- **Compare available versions to stored models**. The update scripts now knows which configuration are available, and for which a learned model already exists. Because we only want to learn new configurations, we subtract the learned configurations from the available configurations to get the those for which a model is not yet learned.
- **Generate `.drone.yml` pipeline configuration**. After determining for which configurations a model should be learned, the Drone configuration file is generated. This file will be generated based on the `.drone.yml.j2` template stored in the learn manager. The input for this template is a list of combinations of implementation name, version and TLS version. Based on this, the learning setup as described in Section 6.2 is configured for every combination.
- **Push changes**. Any changes made by the update script (updated configuration or submodule) are collected in a Git commit and pushed to the learn manager. This push triggers the learning pipeline to be executed on Drone.

6.2 Learning setup

The learning setup consists of multiple components and is responsible for inferring the state machine of a particular implementation version, using a specific version of the TLS protocol.

6.2.1 Overview

As discussed in Section 2.2.2, we follow Angluin's MAT framework for model learning. This framework however, does not map cleanly to an actual black-box learning setup, since we do not have a teacher who knows the full state machine (this is what we are trying to find after all). Learning setups therefore often approximate this framework by using a learner, a tester, a mapper and a system under test (SUT) [39]. Together, these components can conceptually represent the *learner-teacher* setup from the MAT framework.

In this representation, the role of the learner remains the same; it uses a learning algorithm such as as Angluin's L^* to incrementally construct the state machine by sending membership and equivalence queries. The teacher is modeled through a combination of the tester, the mapper, and the system under test.

For membership queries, the learner sends an input sequence to the SUT through the mapper, and compares the observed output with the expected output. Equivalence queries are sent to the tester. These queries cannot be answered directly (since the state machine is unknown), and are instead approximated with a *conformance testing algorithm* [20]. The conformance testing algorithm will attempt to find a counterexample for a given state machine hypothesis, by sending test queries to the SUT and observing the output; if no counterexample can be found, the hypothesis of the learner is considered equivalent to the actual state machine.

The system under test can be any given TLS implementation. It has no knowledge of the test setup, as it will simply expose a TLS server. The SUT does not understand the abstract messages generated by the tester, and the tester can't handle the concrete TLS messages sent by the SUT. The mapper therefore sits between the tester and the SUT and provides a connection to both. Abstract messages received by the tester are converted to concrete TLS messages and sent to the system under test, and vice versa. This allows the tester and the system under test to communicate as if they were talking directly to each other.

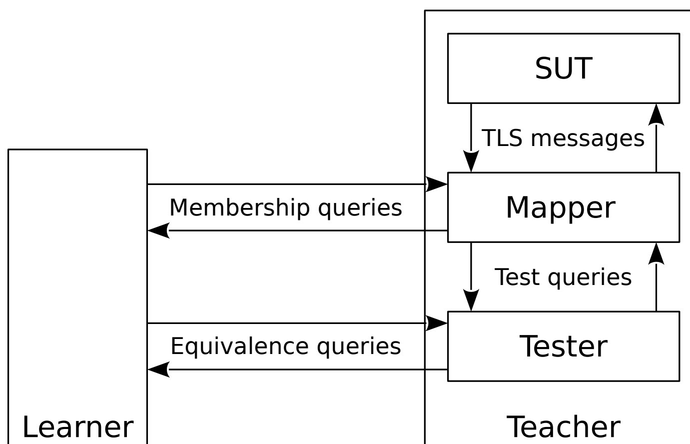


Figure 6.3: Conceptual learning setup

This conceptual learning setup is visualized in Figure 6.3. We run this setup for multiple TLS implementations in parallel in our learning pipeline, as shown in Figure 6.2. After a model has been learned, it will be stored in the learn manager for later use. The next section will provide more details of our learning setup.

6.2.2 Details

In our setup we rely on `LearnLib` [18], a Java library for learning Mealy Machines. It providing various learning algorithms, as well as multiple conformance testing algorithms. This means that in practice, the learner and tester from from Figure 6.3 are contained in a single learner component. The mapper and the system under test remain the same. The resulting setup is shown in Figure 6.4.

Figure 6.4: Learning setup in practice

The specific components we use in our learning manager are based on those used in [10, 28]:

- **Learner**: As mentioned, we rely on `LearnLib` for the learning the models of the SUTs. We do not invoke `LearnLib` directly, instead we use a tool called `StateLearner`[3]. `StateLearner`, among other things, provides a wrapper around `LearnLib`, handles the connections with the mapper and outputs the learned model. It also uses its own conformance testing algorithm, a modified W-method [29], for better efficiency in this particular learning setup.

- **Mapper**: For the mapper we use `TLSAttackerConnector`[4], which takes care of the communication between the learner and the system under test. This mapper uses `TLS-Attacker`[5] to convert the abstract input alphabet into concrete TLS messages and manage the connection, including TLS state, with the SUT.

- **SUT**: The system under test is a specific version of a given TLS implementation, one of the build artifacts from the build stage. It has no knowledge of the test setup, and simply exposes a TLS server, similar to normal production usage.

The learner and mapper we use in our setup are forks of StateLearner[6] and TLSAttackerConnector[7]. We packaged both tools in a Docker image to include them our pipeline, and made some minor changes to the mapper[8].

The learning pipeline is similar to the build pipeline; the different sub-pipeline are executed independently from each other, and failures do not affect other sub-pipelines. The exact steps of each sub-pipelines (also shown in Figure 6.5) are:

- **Clone**: In this step the Git repository of the learn manager is cloned by Drone, and the configuration for this sub-pipeline is loaded.

- **SUT**: Start the target implementation in the background, with the TLS server listening, in preparation of learning.

- **Connector**: Start the mapper, TLSAttackerConnector, in the background, in preparation of learning. The TLS version to use is configured here, as the concrete TLS messages are created by the mapper.

- **Learner**: Run the learner, StateLearner, to infer the state machine of the TLS implementation. Prior to learning, an extra check is performed to see if a model for this configuration is present, and aborts if this is the case. This can happen if the pipeline is (manually) triggered without regenerating the pipeline configuration. The result from this stage is the learned model, a state machine in the DOT format, a graph description language.

- **Commit**: The learned model will be stored in the learn manager by creating a commit with this model file and adding it to the `tlsprint/models` repository. Because this commit action will occur frequently in this pipeline, we do not want to use push to the repository directly, as this will result in conflicts. Instead, we use the GitLab API to create these commits.

[3] https://github.com/tlsprint/statelearner
[4] https://github.com/tlsprint/TLSAttackerConnector
[5] https://github.com/RUB-NDS/TLS-Attacker
[6] https://github.com/jderuiter/statelearner/
[7] https://github.com/jderuiter/TLSAttackerConnector
[8] https://github.com/jderuiter/TLSAttackerConnector/compare/74542f7504e5edd2a68ff0bbbeb27edf731a0dde...tlsprint:9b298f275e0d776e380e7faec1d0ea9ad372ede8

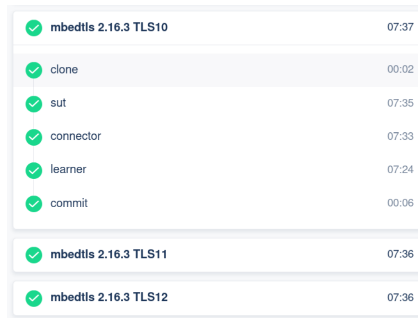


Figure 6.5: Learning pipeline on Drone

6.3 Learning alphabet

In order to infer models from the TLS server implementations, we need to define a set of abstract input messages available to the learner, this is the *learning alphabet*. As discussed earlier in Chapter 4, we excluded TLS 1.3 from our setup. As such, we only include messages from the TLS 1.0 handshake in our alphabet. The complete input alphabet, which includes messages usually send by both client and server for extra test coverage, is displayed in Table 6.1. Most of these message are directly defined in the TLS 1.0 handshake, and can thus be used as is. For some messages however, the semantic meaning is also determined by their content. These include, but are not limited to, `ClientKeyExchange`, `ServerKeyExchange` and alerts. The structure and the possible responses of the exchange messages are dependent on multiple factors: the cipher suites and key exchange methods supported by the client and the server, the previous messages sent, etc. Therefore, multiple messages of this type are defined. For the alerts, we choose to only include the one that is most often used, `AlertWarningCloseNotify`, to limit the input alphabet size. This yields the following input alphabet:

Table 6.1: Learning alphabet

Message	Description
`AlertWarningCloseNotify`	Alert message of typr `close_notify`, often used to indicate that one of the peers will close the connection.
`ApplicationData`	Message containing application data. In a normal protocol run this message is sent after the handshake.
`Certificate`	Certificate message, containing either a client or server certificate.
`CertificateRequest`	Message sent by a server, requesting the certificate of a client.
`ChangeCipherSpec`	Signals transition in ciphering strategies.
`ClientHello`	Sent by the client to start the connection and notify server of cababilities.
`DHClientKeyExchange`	A `ClientKeyExchange` containing Diffie-Hellman parameters.
`ECDHClientKeyExchange`	A `ClientKeyExchange` containing Elliptic Curve Diffie-Hellman parameters.
`RSAClientKeyExchange`	A `ClientKeyExchange` containing RSA parameters.
`DHEServerKeyExchange`	A `DHEServerKeyExchange` containing Diffie-Hellman parameters.
`Finished`	Used to signal the end of the handshake phase.
`ServerHello`	Server response to a `ClientHello`, specifing the capabilities of the server.

Message	Description
`ServerHelloDone`	Message sent by the server during the handshake phase.

The learning alphabet only covers the input alphabet, not the output alphabet, as we cannot know ahead of time all possible responses a system under test might return. There is also a special `RESET` message, which is only understood by the mapper, and not forwarded to the SUT. When the mapper receives this message, it will reset it's connection with the SUT, which places the SUT in it's initial state.

6.4 Learning results

The models resulting from learning are deterministic Mealy machines (Section 2.2.1). They are stored as DOT files, which can be visualized using Graphviz[9]. Figure 6.6 shows an example, the state machine of OpenSSL version 0.9.7 with TLS version 1.0. In this example, the number of edges has been reduced in order for the image to fit the page and be somewhat readable; the primary purpose of this image is to give an impression of the size and complexity of these models. All models, independent of the implementation and version, have some structure in common:

- All models have a single start state, from which all other states can be reached.
- At each state, all input messages are possible and result in an output.
- Most paths eventually end with a `ConnectionClosed`. Cycles are an exception to this.
- There is at least one sink state, where messages resulting in a `ConnectionClosed` end up. This sink state corresponds to a closed connection, so each input results in a `ConnectionClosed` again.

Because the build and learning processes are both automated, new models are added continuously. At the time of writing, the pipeline is configured for two implementations, OpenSSL and mbed TLS, and has learned models for 134 and 114 different versions respectively. Because a lot of versions support multiple TLS versions, the number of models learned is higher. Table 6.2 provides an overview of the model count for every implementation and protocol version.

Table 6.2: Models learned per implementation per TLS version

Name	TLS 1.0	TLS 1.1	TLS 1.2	Total
mbed TLS	114	114	104	332
OpenSSL	134	65	65	264

Not every implementation has a unique model; multiple versions can have an equivalent model. Table 6.3 shows the number of unique models per TLS version (combining all implementations), and the average, largest, and smallest size of the models. In this table, model size refers to the number of states. Table 6.4 shows the amount of unique models per implementation. More detailed statistics about each model can be found in Appendix A.

Table 6.3: Number of unique models per TLS version

TLS version	Unique models	Average model size	Smallest model	Largest model
TLS 1.0	20	9.8	6	14
TLS 1.1	16	9.2	6	14

[9] http://www.graphviz.org/

TLS version	Unique models	Average model size	Smallest model	Largest model
TLS 1.2	15	9.4	6	14

Table 6.4: Number of unique models per implementation

Name	TLS 1.0	TLS 1.1	TLS 1.2
mbed TLS	6	6	5
OpenSSL	14	10	10

All learned models are stored at a directory in the learn manager[10]. In case the reader would like to view statistics on the most recent collection of models, the tables above are generated by our `tlsprint` tool through the `stats` subcommand.

6.5 Discussion

In this section we will discuss some limitations and considerations regarding implementation details and the resulting models.

6.5.1 Implementation details

Similar to the build pipeline, the learning pipeline runs on a self-hosted version of Drone. It also makes use of Docker, but less extensively than the build pipeline. In the learning pipeline, Docker is primarily used to start the learning setup in an isolated environment, which could be replaced by something else, as long as the build artifacts of the different implementations are available.

During the parallel learning pipeline, multiple commits are pushed to the repository. Sometimes, even when using the GitLab API, these commits clash and one of them is rejected. While this is a nuisance, it is not a big problem, as this model will still be included in the pipeline run the next day.

For some versions of mbed TLS, learning a model consistently fails. For most versions of mbed TLS, the learning process takes less than 10 minutes, but for versions 1.2.12 to 1.2.18 and versions 1.3.9 to 1.3.22, the process never terminates by itself. The current solution is a one hour time-out on all jobs, so they are automatically killed if they take to long. Additionally version 1.2.0 of mbed TLS always crashes shortly after the learning starts. Although we suspect non-determinism to be the cause of the never-ending learning, the reasons and solutions for both problems are unclear, and require further investigating in each case.

The only reason `tlsprint/models` is linked to `tlsprint/tls-docker-images`, is because it was the simplest way to reuse the metadata about which versions of the TLS protocol each implementation version supports. If possible, an alternative solution is preferable, such as adding this meta data to the Docker images.

6.5.2 Models

We assume that the learning setup approximates real-life scenarios, in the sense that the models learned in our lab match the behavior of TLS implementations running in production environments. However, it is possible that out test lab yields certain models, which could be different on a different network architecture (e.g. middle boxes, firewalls, different network

[10]https://github.com/tlsprint/models/tree/master/models

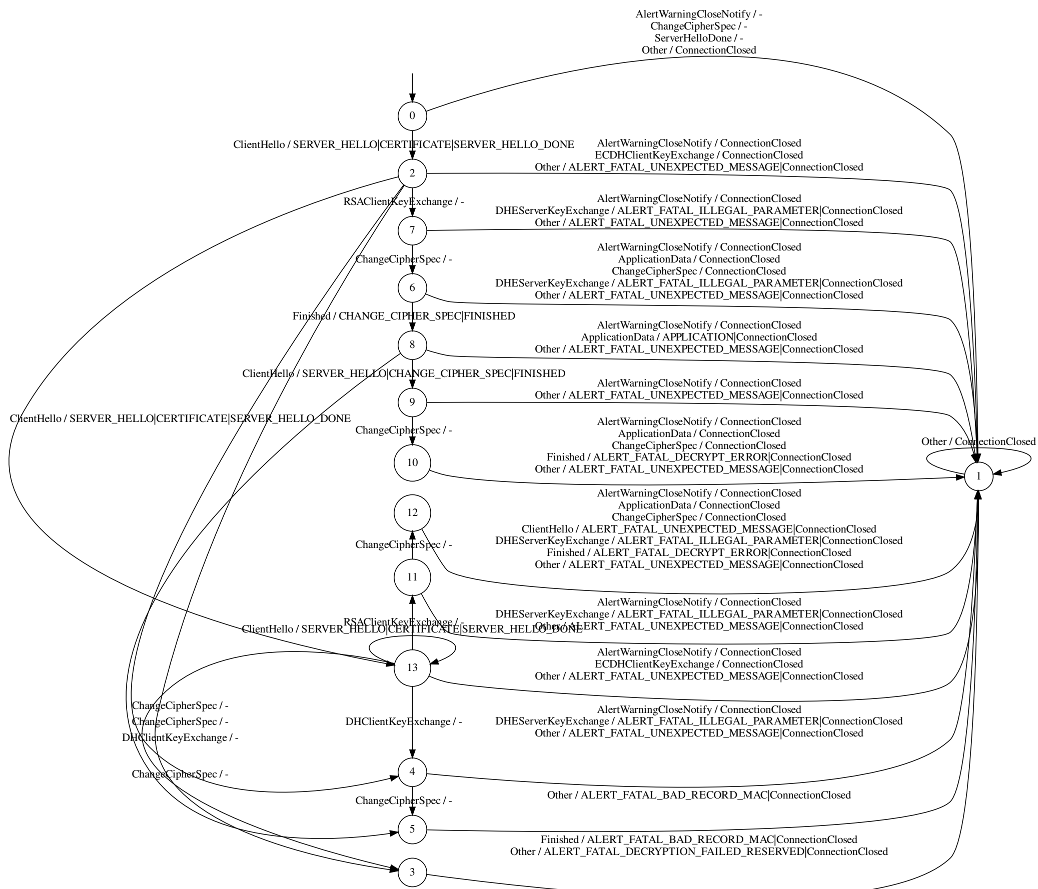


Figure 6.6: Reduced model for OpenSSL 0.9.7 with TLS 1.0

stack / OS, latency). We did not have access to a large collection of TLS implementations running on a different infrastructure (or else we would have used that directly for our learning), so we could not verify this.

Compared to the large number of models, one for every implementation and TLS version combination, the number of unique models is small. This means that many implementations share the same model, and cannot be distinguished from each other with our current method. For future work, it would be beneficial to try to increase the number of unique models, by making them more specific. There are multiple approaches worth exploring:

- Increase the size of the learning alphabet. Include more message types, but also multiple versions of the same message. For example, include a specific `ClientHello` for each combination of supported key types and other algorithms. Care must be taken to account for different server configurations, as this might influence the results. For example, administrations might disable certain cipher suites, use a specific certificate type, or require client side authentication. All of these factors can influence the output for a given input sequence.
- Instead of only looking at the message type of the responses, also look at their structure and contents. This can remain high-level, by looking at how the TLS messages are fragmented in the record layer (for example, is the response to a `ClientHello` embedded in a single TLS record layer block, or multiple), or more detailed, by looking at individual fields.
- Look at timing. This is susceptible to infrastructure, especially when testing in a local environment, and identifying across the Internet, but might provide additional distinguishing information.
- Use different fuzzing techniques. In the current learning setup only valid messages are sent. Additional distinguishing information might be obtained by sending invalid TLS messages.

Chapter 7

Identification

Now that the models of all implementation versions have been learned for multiple versions of the TLS protocol, the final step is to use these models for identification. This is the *identify stage* from Section 4.2. In this stage, we have the same setup as described in Figure 6.4, the difference here is that we do not know which TLS implementation is used by the SUT. We have access to all the learned models, and, by sending input messages to the unknown server, we try match the behavior of the SUT with one of our models. If the SUT is an implementation for which we have no model, a match will be incorrect or not possible. This setting where we have a large group of candidate models and can actively send messages to the SUT is called *active group matching* [32].

A simple but highly inefficient way to determine which model corresponds with an unknown implementation, is to perform model learning on the unknown implementation and compare the result to all known models. Another naive way would be to perform a conformance test for each model to see which one matches. Both of these methods are resource and time intensive, which is undesirable when identifying a large number of implementations, or with servers that are not under your control. To perform the group matching more efficiently (faster and requiring fewer input messages), we applied two different methods: *adaptive distinguishing graph* and *heuristic decision tree.*

The adaptive distinguishing graph (ADG) [7] is a direct generalization of Lee & Yannakakis' algorithm for computing adaptive distinguishing sequences, which we discussed in Section 2.2.3. It pre-computes a decision tree with fixed inputs, and each model is identified through a unique input-output sequence. We collaborated with one of the authors of [7] to make their implementation suitable for our use case. This approach is described in Section 7.3.

In parallel, we developed a new method which we call the heuristic decision tree (HDT). This approach does not pre-compute any sequences, but instead compares all models simultaneously during the identification procedure and dynamically chooses which input to send based on heuristics. We provide more details in Section 7.4.

Before we discuss each method, we first go over the general process of performing the identification with `tlsprint` in Section 7.1. In Section 7.2 we introduce examples models that we will use throughout this chapter.

7.1 General process

The identification stage is different from the previous build and learn stages. The targets for the building and learning stages are mostly constant: there is a list of TLS implementations, new implementations are periodically added, and artifacts can be cached. As such they are

implemented as a pipeline which runs periodically. Identification is more dynamic: it uses the learned models to identify a specified target on demand. So instead of a pipeline, this stage is implemented through our `tlsprint` tool.

Just like the previous stages, we kept this stage as modular as possible. Regardless of which method is used to match the behavior of the SUT, the steps for this stage remain largely the same: remove duplicate models, construct the *model tree*, and perform the identification.

7.1.1 Removing duplicate models

As mentioned in Section 6.4, multiple TLS implementation versions can share the same model. In both identification methods we describe in this chapter, a larger number of models results in longer computation times. Therefore, as an optimization, we first deduplicate these models. This preprocessing step greatly speeds up both processes, while it doesn't change the result.

Currently this deduplication is done based on the file contents of the stored DOT models. This check is both fast, and sufficiently unique; the DOT files which are outputted by the learning setup are consistent in their naming and formatting, even across multiple runs. This is because both the learning algorithm and equivalence algorithm used by `StateLearner` (`LearnLib`'s L^* and a modified version of the W-Method respectively) are deterministic. The deduplication is included as part of the `tlsprint` tool, using the following command:

```
tlsprint dedup [OPTIONS] MODEL_DIRECTORY OUTPUT_DIRECTORY
```

This command reads the directory containing the models, and assumes this directory is the result of the learning stage. As mentioned in Section 6.1.2, this means that the path of the model contains the implementation name, implementation version and TLS version, in the following format:

```
models/$IMPLEMENTATION/$VERSION/$TLS_VERSION/learnedModel.dot
```

Because learning is done separately for every version of the TLS protocol, the models are also deduplicated separately for every TLS version. In the resulting output directory, the TLS versions are therefore the top level directories.

7.1.2 Construct model tree

For both methods there is a processing step before identification can take place: ADG precomputes distinguishing sequences, HDT performs a normalization on the models. In both cases, the data structure which is eventually used for for the identification is a tree, and both have a similar structure. To make these similar trees compatible, we include an additional step in which we use either tree to construct a *model tree*. This model tree ensures that both trees are structured in the same way, and it also contains extra metadata such as the mapping between the models and the implementations. This model tree is then passed to the `identify` command, which can perform the identification regardless of which method is used for matching the behavior.

The layout of the model tree is the same as the normalized heuristic decision tree, which will be further described in Section 7.4.1. The model tree is stored as a serialized Python object, which can be given to the identify command as input. The tree is constructed with the following command, where the `--tree-type` option can specify `adg` or `hdt`:

```
tlsprint construct [OPTIONS] DEDUP_DIRECTORY OUTPUT
```

When to use this command is depending on the method: for the ADG this command is used after generating the graph, for the HDT it is used during the normalization step.

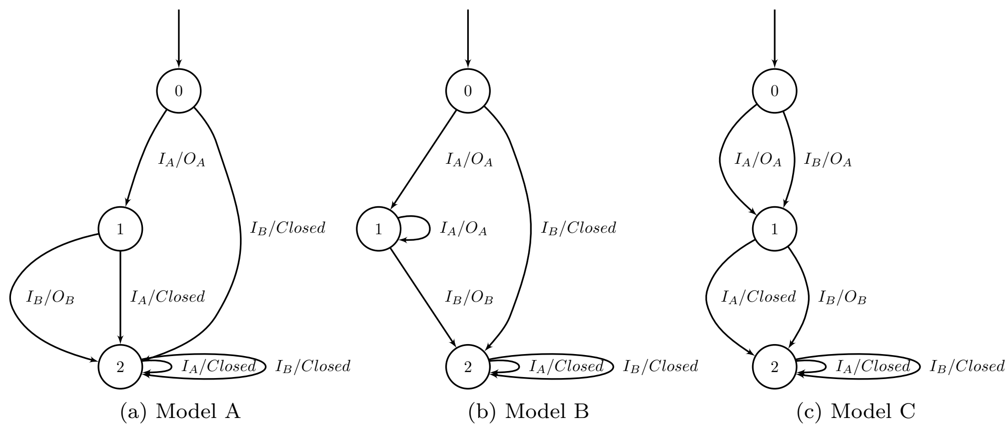
7.1.3 Identification

When the model tree is constructed from either the ADG or the HDT, it can be used to identify a TLS server implementations. The command for this is:

```
tlsprint identify [OPTIONS] TARGET
```

By default this will use the model tree that is supplied with the code, but another can be used with the `--tree` option. This commands starts `TLSAttackerConnector` with a connection to the target, and stops after the process is done. The identification is performed by traversing the model tree based on the specified inputs and received outputs. If a match between a model and the SUT has been made, the implementations corresponding to that model are returned as output.

7.2 Running example



(a) Model A (b) Model B (c) Model C

Figure 7.1: Example models

To explain the different methods, we use an example consisting of three models, for a simple protocol. This example protocol has two inputs (I_A and I_B) and three outputs (O_A, O_B, *Closed*). The expected protocol flow is: send I_A and receive O_A, then send I_B and receive O_B, then the connection is closed. This protocol is not remotely close to the TLS protocol, but suffices to explain the principles of our identification algorithms. Below are three different models of hypothetical implementations, which we shall use as examples.

- Model A (Figure 7.1a) is a strict implementation: it allows the protocol as described, and any deviation will result in a *Closed* response, terminating the protocol.
- Model B (Figure 7.1b) is more forgiving. It allows an infinite number of I_A inputs, and answers them all with O_A, until a I_B is sent.
- The third model, model C (Figure 7.1c), always returns O_A after the first message, regardless of the input. It then only excepts and I_B, returning *Closed* on I_A.

Note, that all models conform to the description of the protocol. This directly demonstrates the problem with protocol descriptions in prose: it leaves room for ambiguity.

7.3 Distinguishing sequences

In Section 2.2.3, we discussed the concept of *distinguishing sequences* and how they can solve the *state identification problem*. For our problem where we want to match the behavior of a

SUT with a set of known models, distinguishing sequences quickly come to mind. We explored various methods for computing distinguishing sequences on our models, and eventually arrived at the *adaptive distinguishing graph* (ADG) [7]. In this section, we will elaborate on how we arrive at this method by discussing some of the approaches we explored first in Section 7.3.1 and Section 7.3.2. Then in Section 7.3.3 we will introduce the adaptive distinguishing graph, and in Section 7.3.4 how we describe how we applied this in our project.

7.3.1 Pairwise distinguishing sequences

In the typical setup for the state identification problem, there is a single state machine, and any of the states can be an initial state. In our case, we have multiple separate state machines – our models – of which we already know the initial state. Since we want the distinguishing the models from each other, we specifically want to distinguish the initial states and do not care about the others. Because when we know in which specific initial state the SUT started, we know the matching model and therefore the corresponding implementations.

A simple solution would then be to compute the set of all pairwise distinguishing sequences. That is, given the set of our candidate models as $C = \{M_1, \ldots, M_k\}$, we need to compute a sequence $seq \in I^*$ for every pair M_i and M_j such that $f_{output}(M_i, seq) \neq f_{output}(M_j, seq)$ [32]. To identify the SUT, these sequences are sent to the SUT and the list of possible candidates is refined based on the output. After every sequence, the connection with the SUT must be reset to place the SUT back in it's initial state.

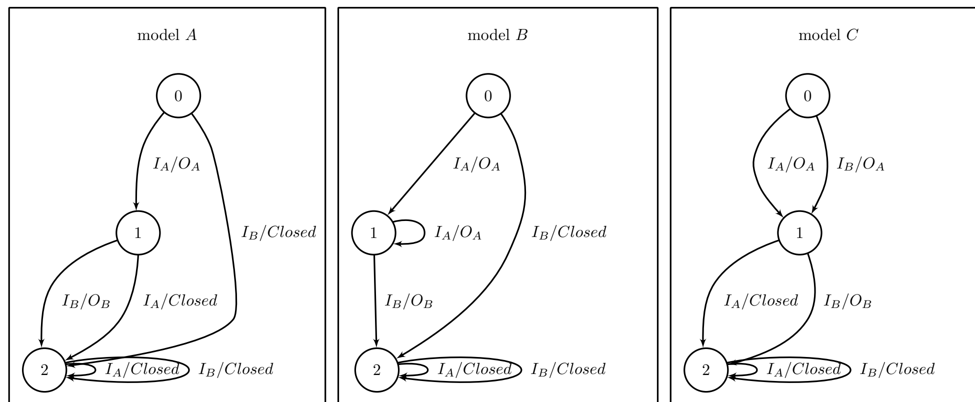
Since we deduplicated the models beforehand (Section 7.1.1) we know that the remaining models are all different and therefore such sequences can be computed. However, regardless of the actual algorithm used to compute these pairwise distinguishing sequences, the performance for this approach is quite poor. For k models, the set of all pairwise distinguishing sequences has cardinality $O(k^2)$. In the best case, only k sequences have to be sent to the SUT to arrive at a conclusion, but this is not guaranteed [32]. For only 20 models, this means computing 400 sequences with varying lengths, and testing at least 20 sequences.

7.3.2 Lee & Yannakakis

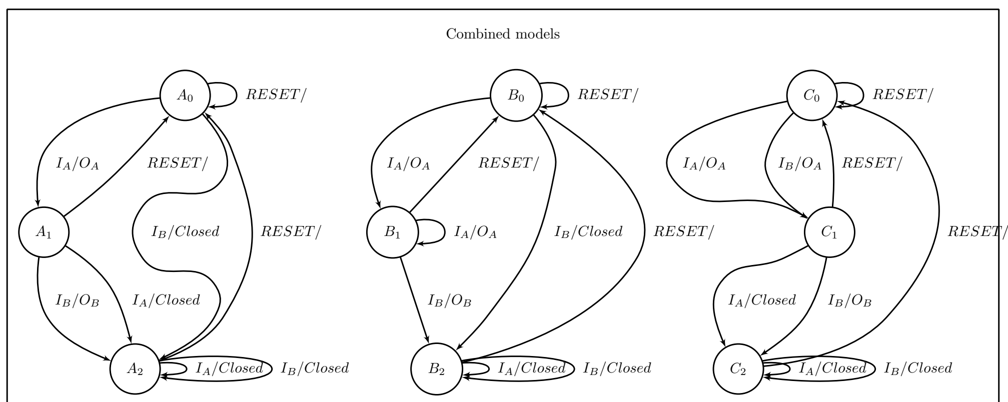
In the approach for pairwise distinguishing sequences, only two models are compared at a time. However, there might be a sequence that distinguishing one model from multiple others, or a sequence that splits the candidate set in two which directly eliminates half the of the possibilities. So a more efficient approach would be to compute a single distinguishing sequence on all models simultaneously.

To do this, instead of having the models as separate state machines we combine them in one large state machine, which then consists of multiple disconnected models. For every model we also add new edges from each state to the initial state with input `RESET` and an empty output. Recall from Section 6.3 that this special message causes the mapper to reset the connection with the SUT, placing the implementation back in it's initial state. These extra edges allow the algorithm computing the sequences to reset the connection if necessary. They ensures that an adaptive distinguishing sequence exists to identify the SUT, because at the very least all pairwise distinguishing sequences can be chained with `RESET`s. This conversion, performed on the example models, is shown in Figure 7.2.

Given this single state machine, we now want to compute the distinguishing sequences either for the initial states, or for all the states (after which we only keep the sequences related to the initial states). To achieve this, we wanted to apply Lee & Yannakakis' efficient adaptive distinguish sequence algorithm, which we mentioned in Section 2.2.3. Unfortunately, this was not possible due to a lack of *valid inputs*, to further elaborate this we will first provide a high-level overview of the algorithm Lee & Yannakakis published.



(a) Separate state machines

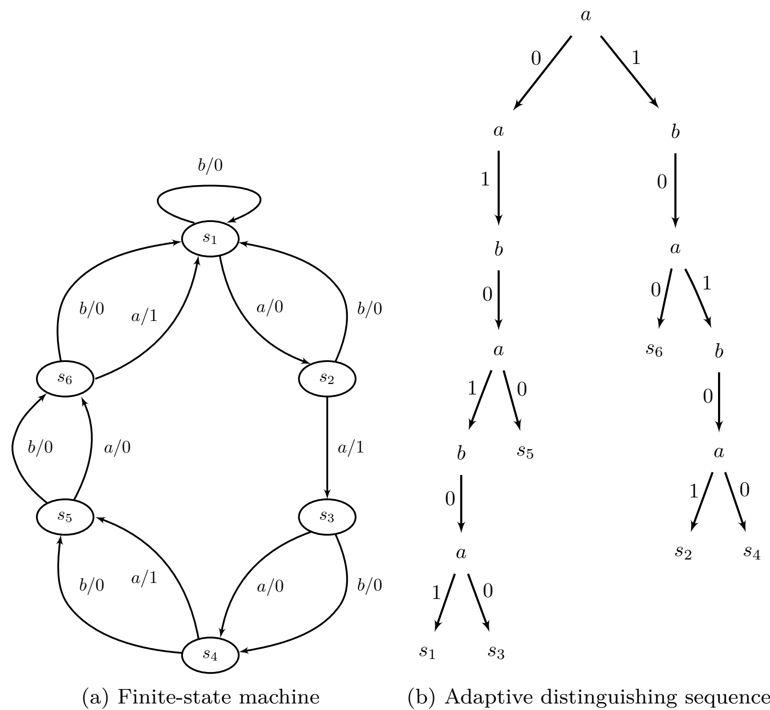


(b) Combined state machines

Figure 7.2: Conversion from separate to combined models

The adaptive distinguishing sequence algorithm starts by initializing a partition π with a single block containing all states. For the example state machine from Section 2.2.3 (shown again in Figure 7.3 for convenience), we have $\pi = \{\{s_1, s_2, s_3, s_4, s_5, s_6\}\}$. The next step is to partition this further by finding inputs that are *valid*. To determine if an input a is valid for a partition block, [21] provides the following:

> "*To check whether an input a is valid for a block B_i, we first partition the states of B_i according to their output symbol on input a, ie., the function $\lambda(\cdot, a)$, and then we further partition each subset according to the function $\delta(\cdot, a)$ (the next state). This partitions B_i into a family of subsets where two states are in the same subset if on input a they produce the same output and move to the same next state. If one of the subsets contains two or more elements, then a is not valid for B_i. If every subset contains one element then a is valid for B_i.*"



(a) Finite-state machine (b) Adaptive distinguishing sequence

Figure 7.3: Example of an adaptive distinguishing sequence for a finite-state machine (duplicate of Section 2.2.3)

Given the initial partition π for this example, we evaluate both possible inputs: a and b. For input symbol a we see that states s_1, s_3 and s_5 output 0 and the others output 1, and the states that produce the same output all transition to a different state. This makes input a valid, and it gives us a new partition $\pi_1 = \{\{s_1, s_3, s_5\}, \{s_2, s_4, s_6\}\}$. On the other hand, input b is not valid here: all states produce the same output, but s_1, s_2 and s_6 all transition to s_1. After this transition, these states can no longer be distinguished from each other, making this input invalid.

This process of finding valid inputs is repeated for each block of the partition, until the partition can no longer be refined. If the result is a partition where all blocks are singletons, then the machine has an adaptive distinguishing sequence.

We now return to our models for which, as we mentioned before, the ADS algorithm can not find valid inputs. This is caused especially by the sink states present in our models. Many,

if not all, states in our models have one or more inputs that lead to the sink state. These inputs cause the SUT to abort the connection, resulting in error or `ConnectionClosed` output in the model. For the sink state, all inputs result in `ConnectionClosed` and keep the state machine in the sink state. As such, every input message results in some states becoming indistinguishable from the sink state, making it invalid. Since there are no invalid input, the algorithm cannot distinguish any states at all.

This problem can be illustrated with the models from the running example. For the state machine in Figure 7.2b we have two inputs, I_A and I_B, and start with a partition of single block B with all states: $\pi = \{B\}$, $B = \{A_0, A_1, A_2, B_0, B_1, B_2, C_0, C_1, C_2\}$. First we look at I_A and partition B based on the output, this gives us $\{\{A_0, B_0, B_1, C_0\}, \{A_1, A_2, B_2, C_1, C_2\}\}$. When we partition this further based on the next state, the result is $\{\{A_0\}, \{B_0, B_1\}, \{C_0\}, \{A_1, A_2\}, \{B_2\}, \{C_1, C_2\}\}$. Since there are subsets with more than one state, input I_A is invalid. The same happens when we apply I_B, after partitioning based on output and next state, the result is $\{\{C_0\}, \{A_1\}, \{B_1\}, \{C_1\}, \{A_0, A_2\}, \{B_0, B_2\}, \{C_2\}\}$. Again, there are subsets with more than one element, which makes I_B invalid as well.

7.3.3 Adaptive distinguishing graph

Lee & Yannakakis' algorithm has several limitations which prevent us from applying it to our models. This lead us to the *adaptive distinguishing graph*, a direct generalization of Lee & Yannakakis' algorithm, published by Van den Bos and Vaandrager [7]. This algorithm is able to compute distinguishing sequences for our models, and we collaborated with one of the authors to use their implementation for our project. In this section we will highlight some key points of the algorithm; we refer the reader to [7] for more details.

The ADG generalizes Lee & Yannakakis' algorithm in two ways. The first generalization is that the ADG allows splits that the distinguishing sequence algorithm would consider invalid, and this is the reason we can apply it our models. When trying to find an adaptive distinguishing sequence, Lee & Yannakakis' algorithm constructs a splitting graph where valid inputs split the state partitions in disjoint sets. The splitting graph created by the ADG algorithm extends this by allowing transitions that split the state partitions in intersecting sets, as long as these splits meet other conditions. For the exact details, we refer the reader to [7].

The second generalization is that ADG doesn't use the finite-state machine framework, but instead works with a *labeled transition system* (LTS) [38], a much more general framework for representing systems. In an FSM, inputs and outputs strictly alternate, and the output is defined purely by the input and the state. In an LTS, transitions are labeled by either an input or an output and both can be present at any point. Among others, an LTS allows output nondeterminism and states without transitions for all inputs [7], making it capable of representing more complex systems than an FSM. This second generalization is not directly relevant for our application, but it means that in order to apply the ADG algorithm to our models, we have modify their representation. We keep the same approach as before, combining all the models and adding the `RESET` edges. Then, to convert them to an LTS, we split all edges in separate input and output edges. The result for the example models is shown in Figure 7.4.

When we apply the ADG algorithm on the LTS version of the example models, the result is an ADG to distinguish *all* the states from our models. Since we are only interested in the initial states, we discard everything not related to those from the ADG. The result is shown in Figure 7.5, formatted in a similar style as the adaptive distinguishing sequence in Section 2.2.3.

The identification procedure is straightforward, the same as for the ADS: start at the top, send the specified input and move down depending on the output until a leaf node is reached. This leaf node then indicates the model that matches the observed behavior. It is important

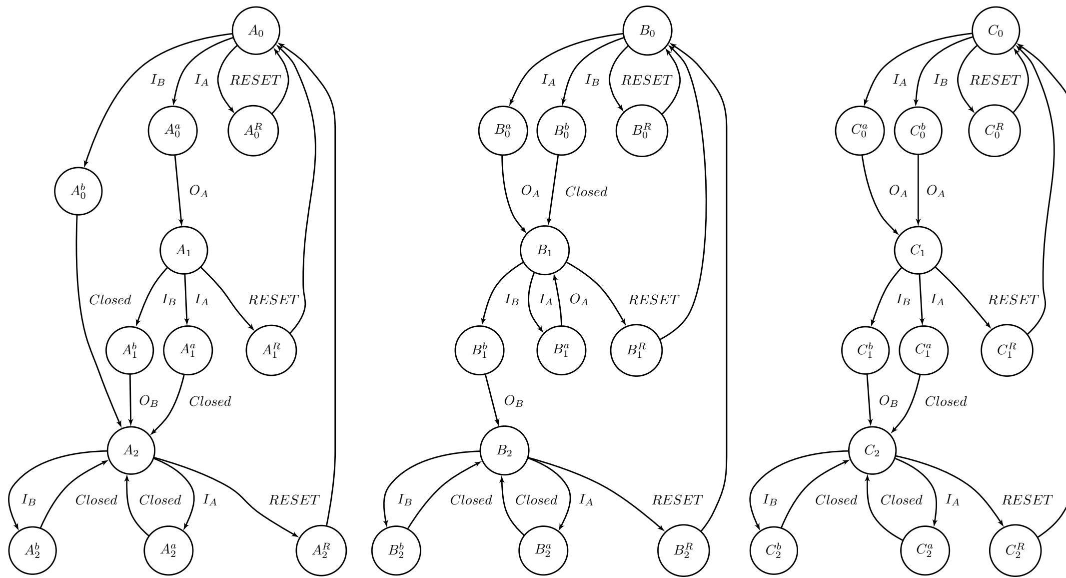


Figure 7.4: Example models converted to a labeled transition system

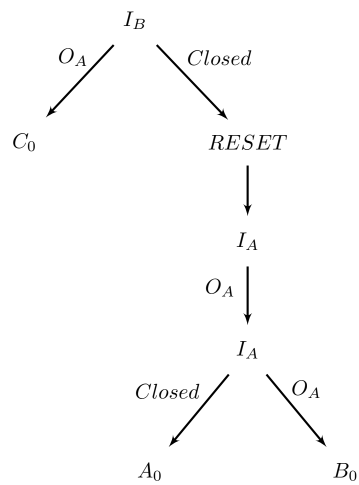


Figure 7.5: Adaptive distinguishing graph of the example models

to note that if this approach is used to match an implementation for which a model is not yet learned, the results will be unpredictable. The implementation might yield an output that is not present in the ADG which will rightfully prevent a match, but if the behavior is similar enough to any of the known models it will result in a false match.

7.3.4 Integrating ADG

To incorporate the Haskell implementation from [7] in our project, we collaborated with one of the authors to extend their implementation for our use case. Specifically, the result of our collaboration was a command-line tool called `adg-finder`. `adg-finder` takes the learned models as input files, computes the ADG for the combined models, discards everything not related to the initial states, and outputs the result as a DOT file. From this DOT file we can then create the model tree as discussed in Section 7.1.2, which can then be used for the identification. The model tree version of the ADG for the example models is shown in Figure 7.6. We packaged this tool as a Docker image[1], so it can fit in the rest of our pipeline.

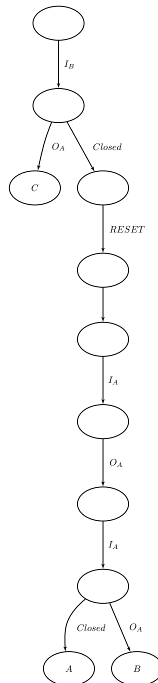


Figure 7.6: Model tree of the adaptive distinguishing graph for the example models

The input files for `adg-finder` should be provided as JSON files. Converting our (deduplicated) models to this JSON structure can be done with `tlsprint` using the following command:

```
tlsprint convert [OPTIONS] INPUT OUTPUT
```

This conversion command has two options: `--add-resets` and `--name`. The first option adds the reset edges the initial state. The second options specifies the name for this model which is then prefixed to every node, this is the part that converts state 0 of model A, to state A_0 in the combined model.

To summarize, the complete process for going from the learned models to identification is as follows:

- Deduplicate the models (`tlsprint dedup`)

[1] https://hub.docker.com/r/tlsprint/adg-finder

- Converted deduplicated models from DOT to JSON (`tlsprint convert`).
- Compute the ADG from the JSON files (`adg-finder`). For our combined models this takes around 10 to 15 minutes on our build server.
- Construct the model tree from the ADG (`tlsprint construct`).
- Use the model tree to identify targets (`tlsprint identify`).

In the next section we will describe our second approach for identification, the heuristic decision tree. We will compare the two methods in Section 7.5.

7.4 Heuristic decision tree

The second approach we discuss for identification is the *heuristic decision tree* (HDT), which we developed during this project. The idea of this approach is that instead of precomputing sequences, the SUT is compared to all available models simultaneously during the identification process. This will allow for a more dynamic and possibly more efficient identification process. It also provides practical benefits, such as the ability to easily include new models in the comparison without any expensive precomputation.

During the identification, which input to send to the SUT is determined by looking at all possible inputs for all models, and choosing one based on certain heuristics (such as largest possible split in candidate models, or a weight assigned to the input messages). The selected input is sent to the SUT, and based on the output a state transition is performed in all available models. The models which have a different output to the given input are removed from the set of candidates, as they do not match the observed behavior. Based on the remaining models available, and their current state, a new suitable input message is determined and sent. By repeating this process, one model will eventually remain (or none, if the implementation does not match any of the models).

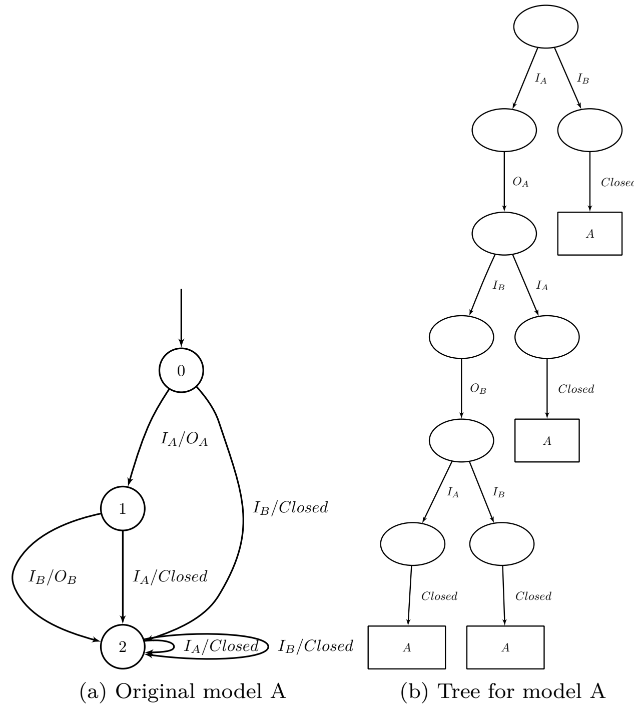
In our implementation we first perform a normalization to more easily compare all models simultaneously. We do this by converting each model to a tree, which we can then merge together into a single large tree. This process will be explained in more detail in Section 7.4.1. During the identification, the normalized tree can be traversed and pruned, as desired, which will be the topic of Section 7.4.2.

7.4.1 Normalize models

In the heuristic decision tree approach, we want to compare all models simultaneously. To aid this process, all models are normalized into the same structure before the identification process begins. We choose to do this by converting a model into a tree, where each branch represents a possible path through the model, an example is shown in Figure 7.7. Because the models are mostly directed towards a sink state, these branches usually end with a `ConnectionClosed` message.

As can be seen in the example, for each possible path in the model, there is a unique branch in the tree identified by the inputs and outputs. The inputs and outputs are on a separate edge, to allow different outputs for a single input when the models are merged together. This labeling is similar to the labeled transition system used by the adaptive distinguishing graph from Section 7.3.3. The name of the model will be added to each leaf of the tree as label, to indicate which paths corresponds to which model when the trees are merged together. The other nodes in the tree are not labeled, as the only significant information is contained in the edges.

In constructing the tree, we use the fact that all paths lead to a sink state, and all paths end with a terminating message (named `Closed` in the example and `ConnectionClosed` in the actual models), to recursively unwrap the model to a tree. The only thing that makes this more complicated, is the fact that some models contain cycles, so we cannot solely use the



(a) Original model A (b) Tree for model A

Figure 7.7: Model A with normalized tree result

recursive “stop when a `ConnectionClosed` is found.” The most simple solution to remedy this, is to specify a maximum recursion depth for the number of states, which should be the same for all models being normalized. The value we use for the maximum depth, is the number of nodes of the largest model, as we assume this would be the maximum depth if no cycles would be present.

After normalization, the models in tree form can then be merged into a single large tree. If two models contain the same path, then the leaf of the corresponding branch should contain both model names as label. Figure 7.8a shows the tree resulting from normalization and merging the three example models. This tree contains a lot of redundant information. For example, after input I_B and output O_A, all remaining sequences correspond to model C. The same goes for the sequence (I_A, O_A, I_A, O_B), after which only model B remains. Additionally, the branches after (I_A, O_A, I_B) provide no distinguishing information, as all three example models have the same behavior. In what we call the *condense* step, we make this tree smaller by removing branches with no distinguishing information, and shorten branches as much as possible. The result is the condensed tree as shown in Figure 7.8. It must be noted that even though condensing does not remove distinguishing information, it does remove *some* information. As a result, unknown implementations with behavior similar to known models have an increased change of being matched falsely; is also the case with the ADG as noted in Section 7.3.3. If this is problematic, this step can be skipped to reduce this probability, but it will yield a (much) larger tree.

After condensing the tree, the normalization step for the heuristic decision tree is finished. During the identification stage, this tree will be pruned further depending on the output of the actual system under test, which will be described in more detail in the next section.

The normalization step is provided by the `construct` command of our `tlsprint` tool, which takes the result of the deduplication stage (described in Section 7.1.1) as input. As we discussed in Section 7.1.2, the output of this step is the model tree, which in this case is the heuristics

(a) Full huristic decision tree

(b) Condensed heuristic decision tree

Figure 7.8: Heuristic decision tree for example models

decision tree with additional metadata added.

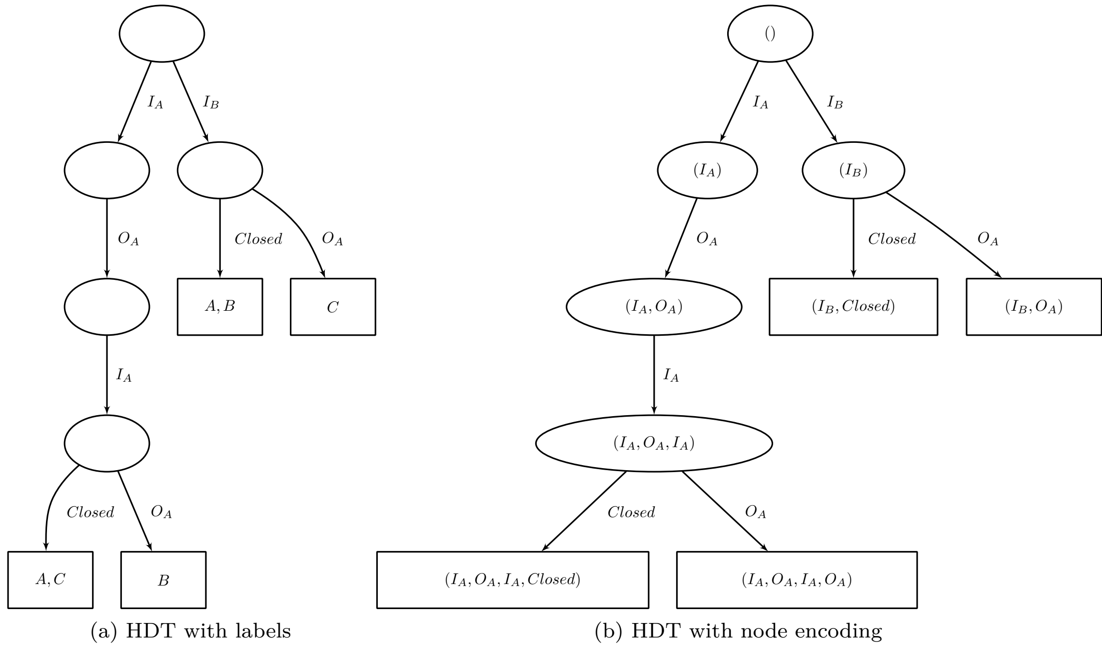
7.4.2 Identification procedure

After the normalization stage, we have a tree with all (distinguishing) paths from all models. We use this tree to match the behavior of the SUT to one of the available models. When matched, we can then lookup the list of possible TLS implementation versions that correspond to this model.

An implementation detail: in the pseudocode listings in this section we will refer to nodes in the heuristic decision tree as the path required to reach them; a tuple of all inputs and outputs. This is not the label, but an internal node encoding. Figure 7.9 shows the example heuristic decision tree twice; once with the labels, and once with the node encoding.

The identification procedure for the heuristic decision tree consists of three subroutines. These are repeated in a loop until one candidate model remains, or behavior is encountered that doesn't match any of the models. We call these subroutines *descent*, *prune* and *condense*, and will describe each in more detail in this section. Pseudo code for the identification procedure using these subroutines can be seen in Listing 7.1.

- **Descent**: This recursive procedure starts at the top of the tree, and moves downwards depending on the chosen inputs and received outputs. At the start of every round, it looks at the possible inputs; this corresponds to the edges originating from the current node. Based on heuristics or other rules, an input is selected and sent to the system under test (more on input selection in Section 7.4.3). When the output is received the subroutine moves downwards in the tree, following the appropriate edges. It then starts another round and selects another input. This is repeated until a leaf node is reached. If at any point the output is not recognized, the identification process stops, as the result no longer matches any of the models. If a node is reached with only one model in



(a) HDT with labels (b) HDT with node encoding

Figure 7.9: Node encoding example for heuristic decision tree

its label, this model will be returned as the match. Pseudo code for this subroutine is shown in Listing 7.2.

- **Prune**: When a leaf node is reached, we know that the observed behavior matches all models listed in this node. This means all other models can be removed from the comparison, as their behavior doesn't match the observation. This is done by removing the names of these unmatched models from all labels, and removing the nodes which then no longer have a label.
- **Condense**: After pruning, some nodes will now have a label consisting of all remaining candidate models. These nodes no longer provide any distinguishing information, and can be removed. In the condense step, all nodes and branches with no distinguishing information are removed. This is the same procedure which is used after the normalization step described in Section 7.4.1 and shown in Figure 7.8.

After the *condense* subroutine, the special `RESET` message is sent. As discussed in Section 6.3, the mapper doesn't forward this message to the SUT but instead closes the current connection with the SUT and starts a new one. This places the SUT back in the initial state, at the root of the tree, and the *descent* step can be performed again.

7.4.3 Input selection

The selection of an input message in the descent step is a central part of the identification procedure for the heuristic decision tree approach; choosing the proper input messages can results in a faster, and more resource efficient identification. Because the HDT encodes a lot of information, choosing an input can be done based on different heuristics.

These input selectors can range from simple to complex, and in our implementation these are customizable. Two very simple input selection rules are "pick a random input," and a more deterministic "pick the first input." These rules are easy to implement and require minimal computation. They were the initial rules we implemented in our tool, and while such simple

Listing 7.1 Psuedocode for identification

```
tree = The heuristic decision tree

while len(tree.models) > 1
    leaf_node = tree.descent()

    if not leaf_node:
        Return: Unknown behavior encountered

    if leaf_node.models == 1:
        Return: Version information corresponding to remaining model

    tree.prune(tree.models - leaf_node.models)
    tree.condense()
    send("RESET")
```

Listing 7.2 Psuedocode for the descent subroutine

```
tree = The heuristic decision tree
current_node = root node
descending = True

while descending:
    possible_inputs = edges starting at 'current_node'
    input = pick(possible_inputs)
    response = send(input)
    reponse_node = current_node + (input, output)

    if reponse_node not in tree:
        Terminate, behavior does not match any model

    if response_node in tree.leaves:
        descending = False

    current_node = response_node

Return: current_node
```

rules already result in a successful identification, they are not the most efficient in terms of the number of inputs sent, or required number of `RESET` messages. After all, some inputs will always result in the same output, such as starting with a `ClientHello`, while others provide more distinguishing information.

For another input selector we took inspiration from the *impurity measures* used in decision tree learning [34]. Decision tree learning (not to be confused with our heuristic decision tree) is a commonly used technique in data mining and machine learning for generating predictive (classification) models. We adapted the *Gini impurity*, which is defined as

$$\text{Gini}(t) = 1 - \sum_{i=0}^{c-1} \big(p(i \mid t)\big)^2.$$

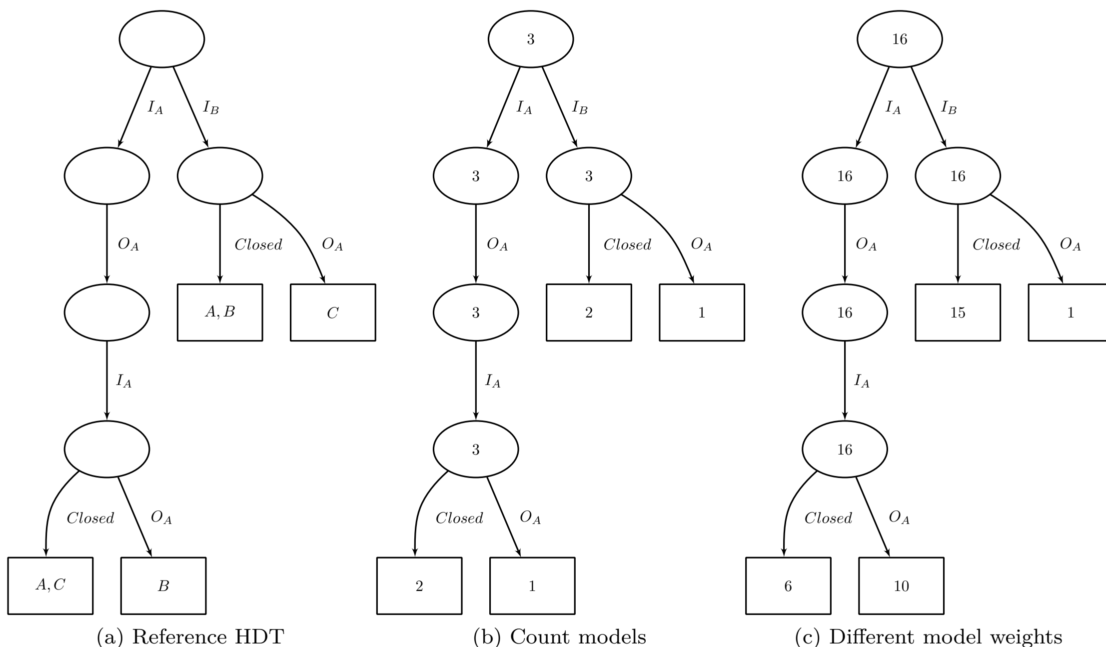
In the context of decision tree learning, this $p(i|t)$ is defined as the fraction of records belonging to class i at a given node t, where c is the number of classes. If the impurity is high, it means

that there is a relatively uniform distribution, if the impurity is low, the distribution is skewed.

In order to use the Gini impurity for input selection during the descent of the HDT, we first have to define the variables, and the way we compute $p(i \mid t)$. Also, when choosing an input, we want to *maximize* the impurity, as this yields the split with the most distinguishing information.

First, let $\boldsymbol{t}$ be the node for which we want to compute the impurity, which, as discussed in Section 7.4.2 and shown in Figure 7.9, is encoded as the path required to reach this node. For the input selection, we always want to compute the impurity for different inputs, so the path $\boldsymbol{t}$ always ends with an input message. With I and O as the set of inputs and outputs respectively (as defined in Section 2.2.1), $i_k \in I$ and $o_k \in O$, this gives us $\boldsymbol{t} = (i_1, o_1, \dots, i_n)$.

Next, we introduce a *weight function* `node_weight` which takes any node as input, and returns a numeric value. This weight function is flexible, as it can leverage all information stored in the HDT. For example, `node_weight` could count the number of models in that subtree, and return that as the weight. Figure 7.10b shows the weight of each node when this weight function is applied to the example HDT. Not all models are equal, some describe the behavior of a single TLS implementation, where others correspond to many. Instead of treating them the same, another weight function `model_weight` can assign weights to each model. These can be based on the number of implementations corresponding to a model, or known usage statistics about different implementations. Figure 7.10c shows an example with the following weights for each model: $A = 5, B = 10, C = 1$.



(a) Reference HDT (b) Count models (c) Different model weights

Figure 7.10: Different weight functions applied to the example heuristic decision tree

We can then define $p(o \mid \boldsymbol{t})$ as a function on the input node $\boldsymbol{t}$ and a possible output edge $o \in O$. Using the weight function, we define $p(o \mid \boldsymbol{t})$ as

$$p(o \mid \boldsymbol{t}) = \frac{\text{weight}\big((\boldsymbol{t}, o)\big)}{\text{weight}\big(\boldsymbol{t}\big)} \tag{7.1}$$

where $(\boldsymbol{t}, o)$ is a shorthand for the node $(i_1, o_1, \dots, i_n, o)$. This can then be used to compute

the impurity and perform input selection. An example using the Gini impurity is shown in the next section.

7.4.4 Example

Given our example models from Section 7.2, we are tasked to match the behavior of a given SUT with our models. We use the Gini impurity for input selection, and assign each model the same weight, just as in Figure 7.10b. Suppose the SUT is running an implementation corresponding to model B.

We start in the descent phase at the root, and need to select an input. At the top of the tree are two input edges, I_A and I_B, so we compute the Gini impurity for each input message (visualized in Figure 7.11):

$$\begin{aligned}
\text{Gini}\big((I_A)\big) &= 1 - \sum_{o \in O} p\big(o \mid (I_A)\big)^2 \\
&= 1 - p\big(O_A \mid (I_A)\big)^2 \\
&= 1 - \left(\frac{\text{weight}\big((I_A, O_A)\big)}{\text{weight}\big((I_A)\big)}\right)^2 \\
&= 1 - \left(\frac{3}{3}\right)^2 = 0
\end{aligned}$$

$$\begin{aligned}
\text{Gini}\big((I_B)\big) &= 1 - \sum_{o \in O} p\big(o \mid (I_B)\big)^2 \\
&= 1 - p\big(\mathit{Closed} \mid (I_B)\big)^2 - p\big(O_B \mid (I_B)\big)^2 \\
&= 1 - \left(\frac{\text{weight}\big((I_B, \mathit{Closed})\big)}{\text{weight}\big((I_B)\big)}\right)^2 - \left(\frac{\text{weight}\big((I_B, O_B)\big)}{\text{weight}\big((I_B)\big)}\right)^2 \\
&= 1 - \left(\frac{2}{3}\right)^2 - \left(\frac{1}{3}\right)^2 = \frac{4}{9}
\end{aligned}$$

To maximize the distinguishing information, we pick the input with the highest impurity, I_B. We send this message to the SUT, which responds with the output *Closed*, and we descent the tree to node (I_B, Closed) (see Figure 7.12a).

This node is a leaf node. We check which models are listed in this leaf, and prune the others. This means that model C will be removed from the tree, the result is shown in Figure 7.12b.

In the condense step, all nodes that have both A and B in their label are also removed (Figure 7.12c). The SUT is then reset, and the descent start at the top of the HDT again. For this example, no further choices can be made, as only one input is possible at each round. The descent reaches the node that only has model B in it's label, and this is returned as the result.

Even tough the example is overly simplistic, it demonstrates the identification algorithm and the input selection. In Section 7.5, we shall compare the performance of multiple input selectors when applied to the models learned in Chapter 6, and also compare this to the performance of the adaptive distinguishing graph from Section 7.3.3.

7.4.5 Future work

Our method for matching the behavior of a SUT with learned models works, but there is room for improvement. In this section we will discuss some limitations and offer directions for

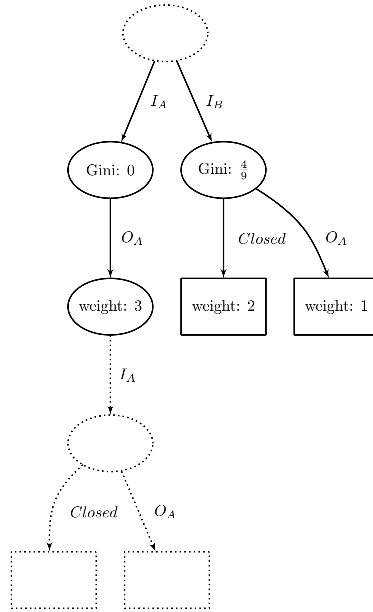
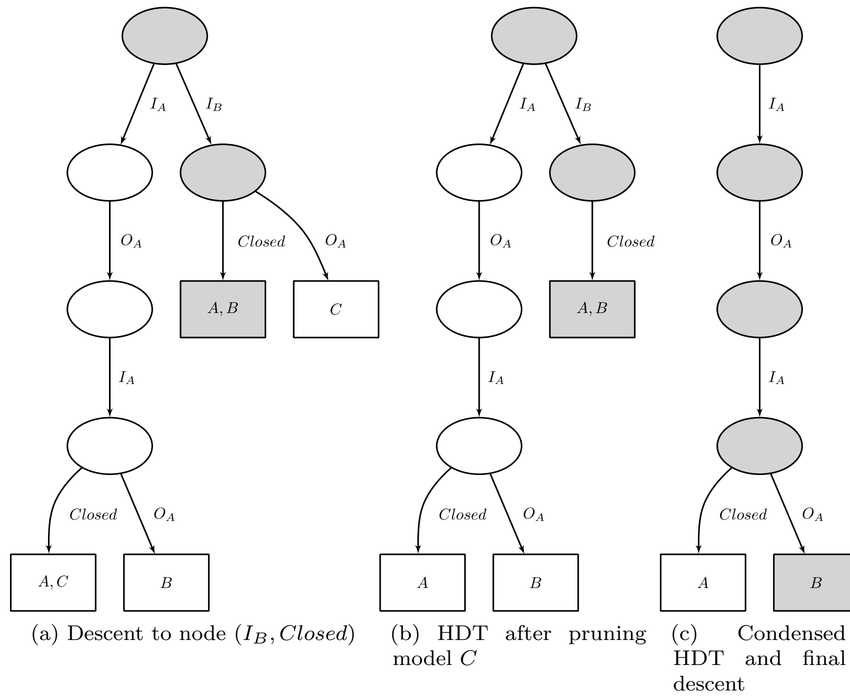


Figure 7.11: Gini impurities for the input selection of the first descent



(a) Descent to node (I_B, *Closed*)

(b) HDT after pruning model C

(c) Condensed HDT and final descent

Figure 7.12: Identification for example models

future work.

7.4.5.1 Normalization

Handling of cycles in models currently uses a simple approach: recursively unwrapping cycles until a maximum depth. A possible improvement would be to detect a cycle instead of unwrapping it and add information about this cycle to the tree (e.g. length of the cycles, involved messages, identifier of the node where the cycle starts and ends). Then after all models are merged, these cycles are expanded, but only as far as necessary to prevent confusion with other paths. If models share (part of) a cycle, then this should also be accounted for. Doing this properly will ensure that all paths are present in the tree (and not truncated before of the max depth), and the size of the tree is minimal (currently these cycles inflate the size significantly).

7.4.5.2 Identification

If the algorithm encounters a TLS implementations for which a model is not yet learned, but the behavior is similar to one of the learned models, then the algorithm might make an incorrect match. If this is a problem, then the original version of the candidate model (not the condensed version in the tree) can be used to verify some or all paths afterwards, as this might find a path that doesn't match.

7.4.5.3 Input selection

We discussed three input selectors: random, pick the first input, and a selector using the Gini impurity. The first two are trivial to implement, but they do not perform any optimization. The selector using the Gini impurity performs better (see the benchmark in Section 7.5.2), but the current implementation only looks one input ahead, as visualized in Figure 7.11. For the example shown in Figure 7.10c, only looking one input ahead means that model C is identified after one input, but both A and B need three. With the different weights for each model, this means that $(15 \cdot 3 + 1) / 16 = 2.875$ input message are sent on average. If the selector would look further ahead, it would see that choosing I_A as the first input is more efficient, as the next split has a significantly higher Gini impurity. Starting with I_A would mean only $(10 \cdot 2 + 6 \cdot 3) / 16 = 2.375$ inputs are sent on average. Optimizing the input selectors is a topic for future work.

We kept our implementation as modular as possible, many parts can be customized with relative ease. There is a loose coupling between the input selectors and the different weight functions, so any input selector can use any weight function. Extending the implementation with a new input selector or weight function only involves writing a single Python function, which is then injected in the identification subroutine.

Some input messages are more computationally expensive than others due to cryptographic operations (`ChangeCipherSpec` for example). It might be desirable to minimize these operations, to reduce load on the host system and/or the SUT. This could by done by introducing an input weight function which is used instead, or in conjunction with, the model weight.

7.5 Comparison

In the previous two sections, we discussed the adaptive distinguishing graph and the heuristic decision tree in detail. In this section we will compare them and see how they perform through a benchmark. To recap first, a high level overview of the key differences:

- **Preparation**: The ADG spends most of the computation time up front by computing the graph, which takes around 10 to 15 minutes on our build server. The HDT only needs to normalize the models, which does not take a lot of time (less than a minute).

- **Tree size**: Both methods result in a compatible model tree. The ADG is relatively small and simple, as it only contains the paths from the distinguishing graph. The size of the HDT is larger and contains more information, as it has all the paths that provide some distinguishing information. Table 7.2 shows the number of nodes for each tree.
- **Tree structure**: The ADG includes the `RESET` inputs and the following path in the graph, while the identification for the HDT sends a `RESET` after reaching a leaf and then starts again in the top of the tree. This means the ADG is a narrow and deep graph compared to the HDT, which is very wide and less deep.
- **Identification**: Since the ADG performed the computation beforehand, the identification process is simple: follow the path depending on the provided inputs and the resulting output. Once a leaf has been reached, the identification is complete. For the HDT, this phase is where most of the work happens: inputs are determined and depending on the output a portion of the candidate models are removed from the comparison.
- **Customization**: The ADG provides a single input each time, and only transitions depending on the received output. Any changes in the input decisions must be done while computing the graph during the preparation and the resulting graph is then unchanged during the remainder of the identification. The HDT is dynamic at its core. This introduces additional complexity during the identification but also allows different optimizations when selecting inputs. It's also possible to change the selection criteria at any time while performing the identification.
- **Including new models**: To include a new model in the ADG, the entire graph has to be recomputed, which will take longer as more models are added. This process is easier with the HDT, only the normalization has to be performed again. While this will also take longer as more models are added, the actual computation time will still low compared to the ADG.

In total we have six different methods that we want to compare: the ADG, and the HDT with three different selectors (first input, random input and Gini impurity), where the Gini method will be applied with three different weights functions. To compare the performance of these methods, we will look at three metrics: number of input messages, number of `RESET`s, and computation time. Both the ADG and the HDT aim to minimize the number of inputs required for a successful identification, so this will be the primary metric. The number of `RESET`s is not part of the optimization, but it is interesting to see how often the connection has to be reset for either method as this operation is relatively expensive. We want to measure the computation time required to perform an identification, because the HDT is performing most of its decisions here, unlike the ADG which has already precomputed the graph. Specifically, we want to know if the additional time required by the HDT is acceptable, since it will impact every identification performed. In the rest of this section, we will describe the benchmark setup (Section 7.5.1) and discuss the results (Section 7.5.2).

7.5.1 Benchmark setup

For the benchmark setup, the initial thought was to use the identification setup as discussed in the beginning of the chapter; start the mapper, start the SUT with the implementation on which we want to run the benchmark, send messages, and collect results. However, this is a very time consuming process, as the overhead – starting the servers for the mapper and the SUT, network latency, cryptographic operations – is quite significant, especially when performing a lot of these tests. We therefore opted for a different approach: using the models directly. Since the models describe the behavior of the implementations, we can also see how they would respond to a given input message without actually sending it. This resulted in a benchmark that is much faster, and it also allowed us to better measure the computation time since it eliminates most of the overhead.

Using the models to respond to inputs is done as follows. The identification functionality in the `tlsprint` tool can use different connectors. One is the connector for the mapper and the other

is the benchmark connector, both can be provided with input messages and return output messages. The connector for the mapper takes care of initializing `TLSAttackerConnector`, sends the inputs to the mapper and returns the received outputs. The benchmark connector does not do any of this, it only simulates a given model. When it receives an input, it looks up which output that model would give, and returns that. Additionally it also keeps track of certain data, such as which messages were sent and received, which will be used for the comparison later.

We also want to compare the performance of the methods when using different model weight functions. As discussed before, not all models correspond to an equal number of implementations, which might skew the results. For example, one of the identification methods could perform significantly better or worse on some model that happens to correspond with only one implementation. Or one of methods has terrible performance for the model of an implementation that is rarely used; this could then be weighed less compared to more commonly used implementations. The model weight functions we include are:

- Equal: Treat all the models as equal, assigning a weight of 1 to each model.
- Count: Count the implementations corresponding to a model and use this as the weight.
- Recent: Here we include a larger weight to more recent implementations, as these are suspected to be more likely deployed in practice. This gives older models, which are generally larger, a low weight in comparison. The weight of the model is then the sum of the weights of the corresponding implementations. The specific weights we choose are:
  - A weight of 20 to OpenSSL version 1.1.1 and later. This is the most recent major version, first released in September 2018.
  - A weight 5 to OpenSSL versions between 1.1.0 and 1.1.1. Version 1.1.0 is the previous version, first released in August 2016.
  - A weight of 20 to mbed TLS 2.16 and up, the most recent long-term support (LTS) version, first released in December 2018.
  - A weight of 5 to mbed TLS versions between 2.7 (the previous LTS version) and 2.16.
  - All other versions receive a weight of 1.

  These weights are rather arbitrary, but they are sufficient to provide a higher weight to only some models.

In Table 7.1, we show the different weights for all models learned for TLS version 1.2. The values for the other TLS versions can be found in Appendix B. These weight functions will primarily be used to evaluate the performance of the different methods in Section 7.5.2. The only identification method that uses these weights during the identification is HDT with the Gini selector.

Table 7.1: Different weight functions applied to models learned for TLS 1.2

Model	equal	count	recent
model-1	1	13	13
model-2	1	13	65
model-3	1	2	2
model-4	1	11	11
model-5	1	1	1
model-6	1	9	9
model-7	1	1	1
model-8	1	8	160
model-9	1	3	3
model-10	1	4	4

Model	equal	count	recent
model-11	1	27	27
model-12	1	33	540
model-13	1	24	108
model-14	1	14	14
model-15	1	6	6

We perform the benchmark for all models we have learned for each TLS version. As can be seen in Table 6.3, we have 51 models in total. Combining this with 6 different identification methods gives us 306 different benchmarks to perform. To get more accurate timing information (and message counts for the random method), we perform the benchmark 200 times for each model and average the results. Performing more iterations doesn't yield much more accurate results. The benchmark is part of the `tlsprint` tool, and can be executed with the command `tlsprint benchmark generate`.

The benchmarks are performed on a laptop with an Intel Core i5-3320M CPU with 4 cores running at 2.60 GHz. This is mainly relevant for the timing information, as the other metrics are independent of the hardware used. The benchmark is parallelized and can run on all cores simultaneously. Performing the 306 tests 200 times each takes approximately 30 minutes.

7.5.2 Benchmark results

We start with some statistics about the specific model trees used for this benchmark. The sizes of the different model trees are shown in Table 7.2. It can be seen that ADG yields a much smaller tree than the HDT even though it already includes the `RESET` messages. This is expected because the HDT includes much more information since the inputs are yet to be determined. Both methods yield smaller trees for later TLS versions, which can be explained by the fact that later versions are supported by fewer implementations and thus have fewer models. Also, more recent implementations have smaller models, as can be seen in Table 6.3 and Appendix A.

Table 7.2: Number of nodes of each model tree

Method	TLS 1.0	TLS 1.1	TLS 1.2
ADG	122	102	101
HDT	343	306	305

As described in the benchmark setup, we measured three different metrics: number of input messages, number of resets, and computation time. In Table 7.3 and Table 7.4 we show some of the results: the number of input messages required to identify each model of TLS 1.2 with every identification method. All methods are deterministic in their input selection with the exception of HDT Random, this method has therefore been given a separate table showing the distribution of the values. Similar tables for other metrics and TLS versions can be found in Appendix C. All the visualizations and tables in this chapter can be generated from the benchmark results using the `tlsprint` with the command `tlsprint benchmark visualize`. Based on these results, we can start to evaluate the performance of the different identification methods.

Table 7.3: Number of inputs for each model of TLS 1.2

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	7	7	5	7	5
model-2	3	3	3	3	3
model-3	9	10	9	8	9
model-4	6	6	4	5	4
model-5	9	10	9	8	9
model-6	7	7	5	7	5
model-7	8	9	12	11	12
model-8	3	3	3	3	3
model-9	8	9	12	11	12
model-10	4	4	7	6	7
model-11	16	16	11	11	11
model-12	3	3	3	3	3
model-13	6	6	6	6	6
model-14	16	16	11	11	11
model-15	11	11	11	11	11

Table 7.4: Number of inputs, distribution of values for each model of TLS 1.2 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	8.89	2.9	5	7	8	11	17
model-2	3.16	0.6	3	3	3	3	7
model-3	9.67	2.44	6	8	9	11	17
model-4	9.61	2.97	3	8	9	12	18
model-5	10.12	2.6	5	8	10	12	16
model-6	6.73	2.7	3	5	6	8	16
model-7	10.48	2.59	5	9	10	13	16
model-8	3.19	0.66	2	3	3	3	6
model-9	12.54	2.52	8	10	13	14	19
model-10	9.54	3.94	3	6	9	13	20
model-11	12.56	2.35	9	11	12	14	18
model-12	4.89	1.58	2	4	4	6	8
model-13	6.92	0.68	6	6	7	7	9
model-14	12.36	2.55	8	11	12	13	18
model-15	11.18	1.29	6	10	11	12	15

7.5.2.1 Detailed results for each model

When we compare the results for the number of inputs required to identify each individual model, the first observation we can make is that the choice of identification method has a definite impact on the results. For most models there are some methods that perform better than others, but no method outperforms all others for all models. For example, in Table 7.3 it can be seen that `ADG` requires fewer inputs than `HDT Gini (count)` for model 9, but more for model 4. For model 1, `HDT Gini (count)` performs better than both `HDT First` and `HDT Gini (equal)`, but worse for model 7. For TLS version 1.2 `HDT First` never performs better than `ADG`, but it does for model 6 of TLS 1.0 as can be seen in Appendix C. If we look at Table 7.4 we can see that on average `HDT Random` is mostly outperformed by the other methods, with some exceptions. For instance, on average `HDT Random` performs better than `ADG` and

HDT First for model 14 and better than all Gini variants for model 7. For many models, the lowest number of inputs is seen in the min column of HDT Random, but the probability of that happening is relatively low and the max values are the highest of all methods for all models.

For the number of resets the results are the same: there are varying differences between the methods and no single one outperforms the others. There is a correlation between the number of inputs and the number of resets: more inputs required often means more resets. Computing the Pearson correlation between inputs and resets over all the results gives a value of 0.90. This makes sense as there is only a limited number of inputs one can send before a model reaches its sink state and a reset is required. A method that requires fewer inputs to identify a model compared to another method often requires less resets as well. There are exceptions, but these only involve cases where the number of inputs differ between methods while the number of resets is the same; an example for this is model 5 of TLS 1.2. There are no models where two methods require an equal number of inputs but a different number of resets, or where one method identifies a model with fewer inputs while using more resets.

Two identification methods that do not differ for number of inputs and resets are HDT Gini (count) and HDT Gini (recent); the results of these two methods are identical for all models. The other Gini method, HDT Gini (equal), does yield different results which means that the provided weight function impacts the choices made by the Gini input selector, but only to a certain extend. We can try to explain the equality by looking at the weights assigned to the models in Appendix B and by the fact that our current implementation only looks one input ahead. The weight function count already assigns some models with higher weight, which impacts the impurity measure of an input when compared to no weights. However, increasing the weight of some of these models even further, as done with usage, doesn't result in a different input being selected.

Looking at the average computation times for each model and method, we see that ADG is consistently the fastest, followed by HDT First. This was to be expected, since ADG has already precomputed the inputs and HDT First doesn't perform any computation either but still has to navigate and prune the larger tree. HDT Gini (recent) performs worst when it comes to computation time, this is caused by the overhead of the model weight function. There is some correlation between the computation time and the number of inputs required, but the amount varies for the different methods. By computing the Pearson correlation between inputs and time for each method, we see that HDT First has the weakest correlation with 0.47, followed by ADG and HDT Random with 0.61 and 0.62 respectively. The Gini methods have a slightly higher correlation; HDT Gini (recent), HDT Gini (count) and HDT Gini (equal) score 0.72, 0.75 and 0.82. The recorded timing includes the overhead of setting up the identification. which included creating a copy of the tree. For all methods except ADG it also includes the time spent condensing and pruning the tree after a descent. It is likely that these constant factors contribute relatively more for methods that do not perform any computation, which explains the lower correlation for HDT First, ADG and HDT Random. For HDT Gini (count) and HDT Gini (recent) the extra time spent evaluating the model weights is also independent of the number of inputs, as it has more to do with the specific models being compared.

7.5.2.2 Statistics of aggregated results

While the results for each individual model are insightful, the data is too fine-grained to easily draw conclusion about how the different methods compare to each other, especially when we consider that the models might have different weights associated with them. We therefore include visualizations of the distribution of the values and tables with statistics such as the mean and quantiles, both for the entire dataset and for each TLS version. Figure 7.13 and Table 7.5 show an example of such a graph and table, the rest can be found in Appendix D. These graphs and tables are generated for all weight functions we described in Section 7.5.1

in order to evaluate the potential performance in different scenarios. Weighing the results is done with the Python packages `seaborn` [40] for the graphs, and with `statsmodels` [30] for the tables. The most interesting statistic here is the average number of inputs required, as this would be the expected performance if the weight function would approximate real-world usage.

Table 7.5: Benchmark summary: Number of inputs with weight function 'equal' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	7.73	3.99	3	4	7	9	16
HDT First	8	4.07	3	4	7	10	16
HDT Gini (count)	7.4	3.36	3	4	7	11	12
HDT Gini (equal)	7.4	2.98	3	5	7	11	11
HDT Gini (recent)	7.4	3.36	3	4	7	11	12
HDT Random	8.79	3.85	2	6	9	12	20

In Section D.1 the statistics and graphs for the weight function "equal" are shown. Since no weighing is performed, it simply shows the distribution of the results per model. In the graphs for each metric we see the difference between the different identification methods we noted earlier. What we can also see is how the results vary between the TLS versions for the same method – especially between TLS 1.0 and the other versions – which shows that it is important to analyze the results separately for each TLS version. Looking at the average number of inputs, we see that (with the exception of TLS 1.0) the relative order of the different methods is the same. The three Gini methods perform best with the same average value, followed by `ADG`, `HDT First` and `HDT Random` in that order. For TLS 1.0 `HDT Gini (equal)` has a slightly lower mean than the other Gini methods. For the number of resets this pattern doesn't hold, but it's still one of the Gini methods with the lowest mean for each TLS version.

Looking at the weighted data, the relative order when it comes to average number of inputs requires remains largely the same, the only difference is that `HDT Gini (equal)` has a slightly higher value than the other Gini methods. For the weighted data, this pattern now also holds for the number of resets: lowest average by `HDT Gini (count)` and `HDT Gini (usage)`, followed by `HDT Gini (equal)`, `ADG`, `HDT First` and `HDT Random` in that order. The differences between the methods become smaller for the "recent" weight function, as can be seen in the graphs and tables.

The mean isn't the only important statistic however, the spread of the values also matters. For both inputs and resets, the standard deviation often follows the pattern we mentioned above; except for "equal" and "count" where `HDT Random` has a lower standard deviation than `HDT First` for the number of inputs.

Looking at the quantiles for the required number of inputs for all three weight functions, we can make some observations. The minimal value, which is independent of the weight function, is the always lowest for `HDT Random`, the other methods tie in second place. For maximum value, which is also independent of the weight function, we see that lowest values are always achieved by one of the Gini methods, followed by `ADG` with a difference of 4 or 5 more inputs. The maximum number of inputs for `HDT First` is either equal to `ADG` or higher and `HDT Random` has the highest maximum values. For the intermediate quantiles 25%, 50% and 75% the results are more mixed and methods often have the same value, especially when weights are assigned. The weight function "recent" even causes the values of all methods (with the exception of `HDT Random`) to be equal for these quantiles. These observations are the same when looking at the number of resets.

For the computation time we see that `ADG` outperforms every other method, being 3 to 10

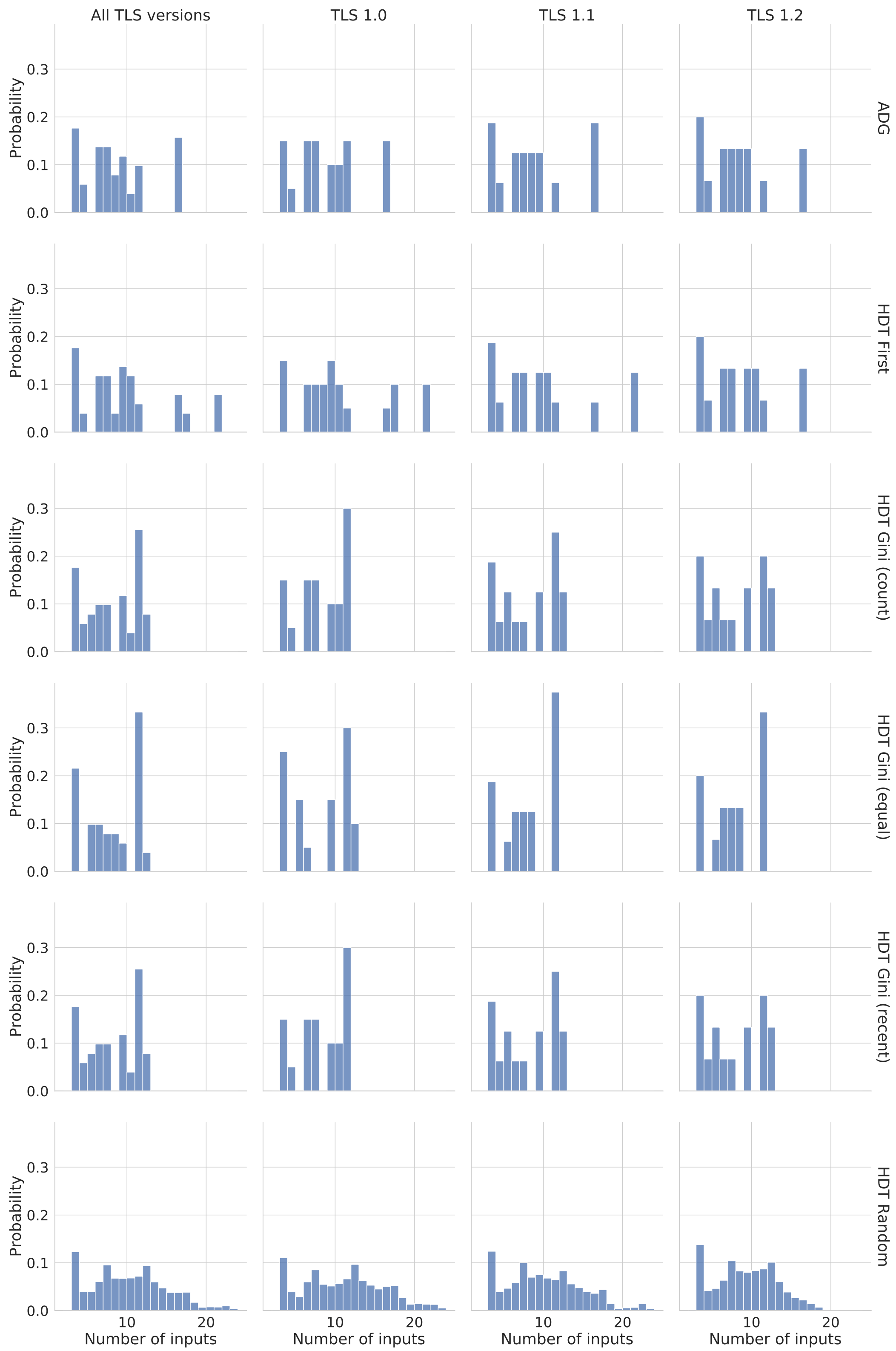


Figure 7.13: Benchmark results: Number of inputs with weight function ‘equal’

times faster on average than the other methods. ADG's longest computation times are lower than the average of any other method, and the value for the 75% quantile of ADG is also lower than the minimal value of any other methods, regardless of the weight function. For the rest we see the observations we made in the previous section confirmed. The second fastest method is HDT First and the slowest is HDT Gini (recent). HDT Random is on average about as fast as HDT Gini (equal) and HDT Gini (count) but with a wider spread and larger variance.

7.5.2.3 Summary

When it comes to number of inputs or resets, the best performance is achieved by one of the Gini methods, regardless of the weight function or TLS version. They are followed by ADG, which has slightly higher average values for both inputs and resets but a much lower computation time. HDT First and HDT Random performed poorly on average with regard to both inputs and resets and didn't make up for it with a significantly reduced computation time.

HDT Gini (count) and HDT Gini (recent) yield the same values for both inputs and resets, but since HDT Gini (count) is faster it can be considered better. When comparing HDT Gini (equal) and HDT Gini (count) for the "equal" weight function, neither performs exclusively better than the other. For all other weight functions HDT Gini (count) results in lower values. Since the "equal" weight is likely not representative for real world usage, HDT Gini (count) seems to be the best HDT variant.

Compared to HDT Gini (count), ADG is much faster since it already pre-computed the graph beforehand. The average computation times of ADG lie between 0.02 and 0.03 seconds, making it around five times faster than HDT Gini (count), which takes 0.1 to 0.15 seconds on average. On the other hand, depending on the TLS version and weight function, ADG requires 0.3 to 1.6 more inputs and 0.1 to 0.4 more resets than HDT Gini (count) on average. ADG also has higher maximum values for both the inputs and resets.

Even though HDT Gini (count) is slower than ADG to compute the inputs, the additional computation cost is still relatively small. When using these methods to identify a TLS implementation, there be additional overhead that will make this difference less noticeable, e.g. network latency and cryptographic operation. If HDT Gini (count) consistently requires fewer inputs and resets than ADG, and less input messages also means less overhead, it can even be the case that HDT Gini (count) is faster in practice, but this should be tested.

Even though the benchmark results are in favor of HDT Gini (count), the differences are generally small, so we are reluctant to conclude that HDT Gini (count) is a better method for performing fingerprint matching than ADG. In our tests we included two major TLS implementations, but there are many more. Including more TLS implementations will result in more unique models, which will inherently increase the complexity of the required input sequences. This affects both methods and it is possible that the performance of ADG will be better than HDT Gini (count) with other models. We must also keep in mind that the implementations of both methods are still a prototype and as such not yet optimized. We used ADG as a black-box, but modifications to its input selection might be possible. HDT Gini (count) can definitely be optimized to reduce the computation time. Experimenting with new input selectors for the HDT might also prove worthwhile, especially those that look more than one input ahead.

What we can conclude, is that both the adaptive distinguishing graph and the heuristic decision tree are capable of finding efficient input sequences to perform fingerprint matching. Even the longest input sequence of ADG, which consists of 16 inputs and 4 resets, is still much more efficient than using pairwise distinguishing sequences.

Chapter 8

Conclusion

Given the broad usage and importance of the Transport Layer Security protocol, it is important that TLS implementations on production systems do not contain any known vulnerabilities. In order to look up a TLS implementation in the CVE database – or just check if it is up to date – its name and version number should be known. Unfortunately, as we mentioned in the introduction, retrieving this information isn't straightforward; the lack of a version banner means fingerprinting is required. We discussed the limitations of existing fingerprinting methods for TLS in Section 3.2: (passive) content based methods are suitable to fingerprint an application but not underlying implementation and current behavior based methods are limited by their reliance on hand-picked sequences.

By leveraging formal methods we developed a new general approach for generating and matching fingerprints of TLS server implementations. Specifically we used *model learning* to generate the fingerprint and approached fingerprint matching as a *state identification* problem; both topics were introduced in Section 2.2. To test our approach we applied it to a large number of versions of two major TLS implementations: OpenSSL (134 versions) and mbed TLS (114 versions).

A large part of our work was focused on developing tooling and setting up supporting infrastructure and automation. This was done to improve reproducibility, future-proof our work through automated updates, and increase the usability of our results. In Chapter 4 we laid out the design of our solution, which consists of three stages: build, learn and identify. We implemented these stages as loosely coupled components that can be executed and developed independent of each other for increased flexibility.

In Chapter 5 we laid out how the build stage consists of multiple pipelines to automatically fetch, build and package new versions of each implementation. These pipelines are scheduled to run periodically – currently daily – and this is expected to remain this way after the conclusion of this thesis project. Even though the build stage is primarily a supporting stage, it might also prove useful for future research, as the artifacts are portable and published to a public location.

After the build stage, we used model learning to infer the state machine of each implementation version for various versions of the TLS protocol. From the previous research we discussed in Section 3.1 we knew that this model can be seen as a fingerprint for that implementation version. Similar to the build stage, we implemented learning as a scheduled pipeline that performs learning daily and publishes the results to a public repository. In Chapter 6 we discussed the details of our learning setup, the learning alphabet, and the resulting models. We also noted that not every implementation resulted in a unique model, depending on the TLS version we had 20 to 15 unique models.

With the fingerprints in the form of the learned models we moved to the identification. We approached fingerprint matching as a state identification problem and applied two methods for computing distinguishing sequences. A Python package with the name `tlsprint` was developed to perform the identification with either method. In Section 7.3.1 we argued that input sequences to identify each model must exist. At the very least we should be able to compute pairwise distinguishing sequences, but that is an inefficient approach that scales poorly.

To compute more efficient sequences, we wanted to apply Lee & Yannakakis' algorithm for adaptive distinguishing sequences. Unfortunately, as we discussed in Section 7.3.2, this algorithm has limitations which prevented us from applying it to our models. In Section 7.3.3 we examined another algorithm called the *adaptive distinguishing graph* (ADG), a direct generalization of Lee & Yannakakis' algorithm. This generalization didn't suffer the same limitations and allowed us to compute distinguishing sequences for our models. After collaborating with one of its original authors, we successfully integrated the ADG in our solution, as we described in Section 7.3.4.

In parallel to finding a solution based on Lee & Yannakakis algorithm, we developed a new method which we called the *heuristic decision tree* (HDT). This configurable and extendible approach doesn't compute a sequence beforehand, but instead compares all models simultaneously during the identification. It supports multiple algorithms for selecting the input message based on heuristics. We presented this method in great detail in Section 7.4.

We compared the ADG and the HDT extensively in Section 7.5. Besides highlighting the most importance differences, we performed benchmarks in which we measured three metrics: number of inputs required to successfully identify an implementation, the number of times the connection must be reset, and computation time spent during the identification. Because the HDT is configurable with multiple input selectors, we included five variants of the HDT in the comparison, alongside the ADG. These different methods included two very simple selectors (random input and always pick the first input) and three versions using the Gini impurity with different weights associated to the models. The choice of input selectors had a large impact on the performance of the HDT – the simple selectors performed relatively poor – but the impact of the weights was smaller.

The best performing identification methods were the ADG and a variant of the HDT using the Gini impurity we called `HDT Gini (count)`. Compared to `HDT Gini (count)`, the ADG was much faster since it already pre-computed the graph. The average computation times of the ADG lied between 0.02 and 0.03 seconds, which made it around five times faster than `HDT Gini (count)`, which took 0.1 to 0.15 seconds on average. On the other hand, the ADG required on average 0.3 to 1.6 more inputs and 0.1 to 0.4 more resets than `HDT Gini (count)`, ADG also had higher maximum values for both the inputs and resets.

Given these results, we concluded that both the ADG and the HDT with the Gini impurity were capable of finding efficient input sequences to perform fingerprint matching. Even though `HDT Gini (count)` scored better for the number of inputs, it would be premature to conclude that this method performs better than the ADG in general. The current set of models is relatively small and both methods have room for optimizations. Further research is required to see how both methods behave when more models are added. For now, we can conclude that it is possible to automatically find efficient distinguishing sequences with both the ADG and the HDT.

Most limitations and directions for future work have already been discussed throughout this thesis, we summarize the most significant here. An important limitation of our method, is that two implementations sharing the same model cannot be distinguished from each other. For future work, it is worthwhile to investigate if the number of unique models can be increased, by making them contain more distinguishing information. There are multiple approaches

that might accomplish this, which we discussed in Section 6.5.2. For example, instead of only looking at the message type of the responses, it might be useful to also look at their structure and contents. This could be the values of specific fields, or by looking at how the TLS messages are fragmented in the record layer. Another approach could be to apply additional fuzzing techniques. In the current learning setup only valid messages are sent, but additional distinguishing information might be obtained by sending invalid TLS messages as well.

To improve the usability of our approach and tool for real-world usage, there are two additional suggestions for future work. The first suggestion is to add more TLS implementations the build and learn stages, making their fingerprints available for the identification. These implementations could be standalone libraries such as Botan, BoringSSL or LibreSSL, but also programming language specific implementations such as the `crypto/tls` package in Go's standard library. The second suggestion is to learn models for TLS 1.3, the most recent version of the protocol. For the most part, this means extending the tools used in the learning setup, as these do not support TLS 1.3 at the moment.

There is also room for improvement when it comes to the identification process. The implementations of both the ADG and the HDT are still a prototype and as such not yet optimized. We used the ADG algorithm as a black-box, but modifications to its input selection might be possible. The HDT can be optimized to reduce computation time, as this might become more of an issue when more models are added. Additional pre-computation might help to reduce the size of the tree itself, which will improve the speed and potentially reduce the number of inputs further. Experimenting with new input selectors for the HDT might also prove worthwhile, especially those that look more than one input ahead. More suggestions for improving the HDT are given in Section 7.4.5.

Lastly an interesting subject for future research is to apply the approach presented here to other protocols. While we focused on TLS in this thesis, the approach is not tied to the TLS protocol and can be applied more widely.

References

[1] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S. and Zimmermann, P. 2018. Imperfect forward secrecy: How Diffie-Hellman fails in practice. *Communications of the ACM*. 62, 1 (2018), 106–114.

[2] Albanese, M., Battista, E. and Jajodia, S. 2015. A deception based approach for defeating OS and service fingerprinting. *2015 IEEE conference on communications and network security (CNS)* (2015), 317–325.

[3] Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Information and computation*. 75, 2 (1987), 87–106.

[4] Apple's SSL/TLS bug: 2014. *https://www.imperialviolet.org/2014/02/22/applebug.html*.

[5] Aviram, N. et al. 2016. DROWN: Breaking TLS with SSLv2. *25th USENIX security symposium* (Aug. 2016).

[6] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y. and Zinzindohoue, J.K. 2017. A messy state of the union: Taming the composite state machines of TLS. *Communications of the ACM*. 60, 2 (2017), 99–107.

[7] Bos, P. van den and Vaandrager, F.W. 2020. State identification for labeled transition systems with inputs and outputs. *Formal Aspects of Component Software, FACS 2019* (2020), 191–212.

[8] Carroll, T.E., Crouse, M., Fulp, E.W. and Berenhaut, K.S. 2014. Analysis of network address shuffling as a moving target defense. *2014 IEEE international conference on communications (ICC)* (2014), 701–706.

[9] Chapter 8. Remote OS detection: 2011. *https://nmap.org/book/osdetect.html*.

[10] Daniel, L., Poll, E. and de Ruiter, J. 2018. Inferring OpenVPN state machines using protocol state fuzzing. *2018 IEEE European symposium on security and privacy workshops (EuroS PW)* (2018), 11–19.

[11] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M. and Halderman, J.A. 2014. The matter of Heartbleed. *Proceedings of the 2014 conference on internet measurement conference* (2014), 475–488.

[12] Fardan, N.J.A. and Paterson, K.G. 2013. Lucky thirteen: Breaking the TLS and DTLS record protocols. *2013 IEEE symposium on security and privacy* (2013), 526–540.

[13] Fiterău-Broştean, P., Lenaerts, T., Poll, E., Ruiter, J. de, Vaandrager, F.W. and Verleg, P. 2017. Model learning and model checking of SSH implementations. *Proceedings of the 24th ACM SIGSOFT international SPIN symposium on model checking of software* (New York, NY, USA, 2017), 142–151.

[14] Frolov, S. and Wustrow, E. 2019. The use of TLS in censorship circumvention. *Network and Distributed Systems Security (NDSS) Symposium 2019* (2019), 24–27.

[15] Greenwald, L.G. and Thomas, T.J. 2007. Toward undetected operating system fingerprinting. *WOOT '07 proceedings of the first USENIX workshop on offensive technologies* (2007), 6.

[16] Husák, M., Čermák, M., Jirsík, T. and Čeleda, P. 2016. HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. *Eurasip Journal on Information Security.* 2016, 1 (2016), 6.

[17] Isberner, M. 2015. Foundations of active automata learning: An algorithmic perspective. (2015).

[18] Isberner, M., Howar, F. and Steffen, B. 2015. The open-source LearnLib. *Computer aided verification* (Cham, 2015), 487–495.

[19] Kampanakis, P., Perros, H. and Beyene, T. 2014. SDN-based solutions for moving target defense network protection. *Proceeding of IEEE international symposium on a world of wireless, mobile and multimedia networks 2014* (2014), 1–6.

[20] Lee, D. and Yannakakis, M. 1996. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE.* 84, 8 (1996), 1090–1123.

[21] Lee, D. and Yannakakis, M. 1994. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers.* 43, 3 (1994), 306–320.

[22] Lei, C., Zhang, H.-Q., Tan, J.-L., Zhang, Y.-C. and Liu, X.-H. 2018. Moving target defense techniques: A survey. *Security and Communication Networks.* 2018, (2018).

[23] Möller, B., Duong, T. and Kotowicz, K. 2014. This POODLE bites: Exploiting the SSL 3.0 fallback. (2014).

[24] Moore, E.F. 1956. Gedanken-experiments on sequential machines. *Automata studies.* 34, (1956), 129–153.

[25] Rescorla, E. 2018. The Transport Layer Security (TLS) protocol version 1.3. RFC 8446; RFC Editor.

[26] Rescorla, E. and Dierks, T. 2008. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246; RFC Editor.

[27] Rescorla, E. and Modadugu, N. 2006. Datagram Transport Layer Security. RFC 4347; RFC Editor.

[28] Ruiter, J. de 2016. A tale of the OpenSSL state machine: A large-scale black-box analysis. *Secure IT systems* (Cham, 2016), 169–184.

[29] Ruiter, J. de and Poll, E. 2015. Protocol state fuzzing of TLS implementations. *24th USENIX security symposium (USENIX security 15)* (Washington, D.C., 2015), 193–206.

[30] Seabold, S. and Perktold, J. 2010. Statsmodels: Econometric and statistical modeling with python. *9th python in science conference* (2010).

[31] Shamsi, Z., Nandwani, A., Leonard, D. and Loguinov, D. 2014. Hershel: Single-packet OS fingerprinting. *ACM SIGMETRICS Performance Evaluation Review.* 42, 1 (2014), 195–206.

[32] Shu, G. and Lee, D. 2011. A formal methodology for network protocol fingerprinting. *IEEE Transactions on Parallel and Distributed Systems.* 22, 11 (Nov. 2011), 1813–1825.

[33] Shu, G. and Lee, D. 2007. Testing security properties of protocol implementations – a machine learning based approach. *27th international conference on distributed computing systems (ICDCS'07)* (2007), 25–25.

[34] Tan, P.-N., Steinbach, M. and Kumar, V. 2006. *Introduction to data mining.* Pearson.

[35] TLS fingerprinting: Smarter defending & stealthier attacking: 2015. *https://blog.squarelemon.com/tls-fingerprinting/*.

[36] TLS fingerprinting with JA3 and JA3S: 2019. *https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967*.

[37] TLS prober: *https://github.com/WestpointLtd/tls_prober*. Accessed: 2019-08-03.

[38] Tretmans, J. 2008. Model based testing with labelled transition systems. *Formal methods and testing.* (2008), 1–38.

[39] Vaandrager, F.W. 2017. Model learning. *Communications of the ACM.* 60, 2 (Jan. 2017), 86–95.

[40] Waskom, M. and team, the seaborn development 2020. *Mwaskom/seaborn.* Zenodo.

[41] Watson, D., Smart, M., Malan, G.R. and Jahanian, F. 2004. Protocol scrubbing: Network security through transparent flow modification. *IEEE/ACM transactions on Networking.* 12, 2 (2004), 261–273.

[42] Way, O., Ray, M., Dispensa, S. and Rescorla, E. Transport Layer Security (TLS) renegotiation indication extension. RFC 5746; RFC Editor.

[43] Zhuang, R., DeLoach, S.A. and Ou, X. 2014. Towards a theory of moving target defense. *Proceedings of the first ACM workshop on moving target defense* (2014), 31–40.

Appendix A

Details of learned models

A.1 TLS 1.0

Table A.1: Details of models learned for TLS 1.0

Model	Number of states	OpenSSL versions	mbed TLS versions
model-1	14	33	0
model-2	10	13	0
model-3	14	5	0
model-4	8	13	0
model-5	11	11	0
model-6	11	5	0
model-7	14	7	0
model-8	11	11	0
model-9	13	3	0
model-10	8	9	0
model-11	8	8	0
model-12	14	11	0
model-13	11	4	0
model-14	10	1	0
model-15	6	0	27
model-16	8	0	33
model-17	6	0	24
model-18	6	0	14
model-19	6	0	10
model-20	6	0	6

Table A.2: Implementation versions of models learned for TLS 1.0

Model	Implementation	Versions
model-1	openssl	0.9.7e, 0.9.7f, 0.9.7g, 0.9.7h, 0.9.7i, 0.9.7j, 0.9.7k, 0.9.7l, 0.9.7m, 0.9.8, 0.9.8a, 0.9.8b, 0.9.8c, 0.9.8d, 0.9.8e, 0.9.8f, 0.9.8g, 0.9.8h, 0.9.8i, 0.9.8j, 0.9.8k, 0.9.8m, 0.9.8n, 0.9.8o, 0.9.8p, 0.9.8q, 0.9.8r, 1.0.0, 1.0.0a, 1.0.0b, 1.0.0c, 1.0.0d, 1.0.0e
model-2	openssl	1.0.2, 1.0.2a, 1.0.2b, 1.0.2c, 1.0.2d, 1.0.2e, 1.0.2f, 1.0.2g,

Model	Implementation	Versions
		1.0.2h, 1.0.2i, 1.0.2j, 1.0.2k, 1.0.2l
model-3	openssl	0.9.7, 0.9.7a, 0.9.7b, 0.9.7c, 0.9.7d
model-4	openssl	1.1.0, 1.1.0a, 1.1.0b, 1.1.0c, 1.1.0d, 1.1.0e, 1.1.0f, 1.1.0g, 1.1.0h, 1.1.0i, 1.1.0j, 1.1.0k, 1.1.0l
model-5	openssl	0.9.8zb, 0.9.8zc, 0.9.8zd, 0.9.8ze, 0.9.8zf, 0.9.8zg, 0.9.8zh, 1.0.0n, 1.0.0o, 1.0.1i, 1.0.1j
model-6	openssl	1.0.0p, 1.0.0q, 1.0.0r, 1.0.0s, 1.0.0t
model-7	openssl	0.9.8y, 1.0.0k, 1.0.0l, 1.0.1d, 1.0.1e, 1.0.1f, 1.0.1g
model-8	openssl	1.0.1k, 1.0.1l, 1.0.1m, 1.0.1n, 1.0.1o, 1.0.1p, 1.0.1q, 1.0.1r, 1.0.1s, 1.0.1t, 1.0.1u
model-9	openssl	0.9.8za, 1.0.0m, 1.0.1h
model-10	openssl	1.0.2m, 1.0.2n, 1.0.2o, 1.0.2p, 1.0.2q, 1.0.2r, 1.0.2s, 1.0.2t, 1.0.2u
model-11	openssl	1.1.1, 1.1.1a, 1.1.1b, 1.1.1c, 1.1.1d, 1.1.1e, 1.1.1f, 1.1.1g
model-12	openssl	0.9.8u, 0.9.8v, 0.9.8w, 0.9.8x, 1.0.0h, 1.0.0i, 1.0.0j, 1.0.1, 1.0.1a, 1.0.1b, 1.0.1c
model-13	openssl	0.9.8s, 0.9.8t, 1.0.0f, 1.0.0g
model-14	openssl	0.9.8l
model-15	mbedtls	2.0.0, 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7, 2.1.8, 2.1.9, 2.1.10, 2.1.11, 2.1.12, 2.1.13, 2.1.14, 2.1.15, 2.1.16, 2.1.17, 2.1.18, 2.2.0, 2.2.1, 2.3.0, 2.4.0, 2.4.1, 2.4.2, 2.5.0
model-16	mbedtls	2.11.0, 2.12.0, 2.13.0, 2.13.1, 2.14.0, 2.14.1, 2.15.0, 2.15.1, 2.16.0, 2.16.1, 2.16.2, 2.16.3, 2.16.4, 2.16.5, 2.16.6, 2.16.7, 2.16.8, 2.17.0, 2.18.0, 2.18.1, 2.19.0, 2.19.0d1, 2.19.0d2, 2.19.1, 2.20.0, 2.20.0d0, 2.20.0d1, 2.21.0, 2.22.0, 2.22.0d0, 2.23.0, 2.24.0, 3.0.0p1
model-17	mbedtls	2.5.1, 2.6.0, 2.6.1, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5, 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.7.12, 2.7.13, 2.7.14, 2.7.15, 2.7.16, 2.7.17, 2.8.0, 2.9.0, 2.10.0
model-18	mbedtls	1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.10, 1.2.11, 1.3.6, 1.3.7, 1.3.8
model-19	mbedtls	1.0.0, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8
model-20	mbedtls	1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5

A.2 TLS 1.1

Table A.3: Details of models learned for TLS 1.1

Model	Number of states	OpenSSL versions	mbed TLS versions
model-1	10	13	0
model-2	8	13	0
model-3	11	2	0
model-4	11	11	0
model-5	13	1	0
model-6	8	9	0
model-7	13	1	0
model-8	8	8	0
model-9	14	3	0

Model	Number of states	OpenSSL versions	mbed TLS versions
model-10	13	4	0
model-11	6	0	27
model-12	8	0	33
model-13	6	0	24
model-14	6	0	14
model-15	6	0	10
model-16	6	0	6

Table A.4: Implementation versions of models learned for TLS 1.1

Model	Implementation	Versions
model-1	openssl	1.0.2, 1.0.2a, 1.0.2b, 1.0.2c, 1.0.2d, 1.0.2e, 1.0.2f, 1.0.2g, 1.0.2h, 1.0.2i, 1.0.2j, 1.0.2k, 1.0.2l
model-2	openssl	1.1.0, 1.1.0a, 1.1.0b, 1.1.0c, 1.1.0d, 1.1.0e, 1.1.0f, 1.1.0g, 1.1.0h, 1.1.0i, 1.1.0j, 1.1.0k, 1.1.0l
model-3	openssl	1.0.1i, 1.0.1j
model-4	openssl	1.0.1k, 1.0.1l, 1.0.1m, 1.0.1n, 1.0.1o, 1.0.1p, 1.0.1q, 1.0.1r, 1.0.1s, 1.0.1t, 1.0.1u
model-5	openssl	1.0.1h
model-6	openssl	1.0.2m, 1.0.2n, 1.0.2o, 1.0.2p, 1.0.2q, 1.0.2r, 1.0.2s, 1.0.2t, 1.0.2u
model-7	openssl	1.0.1d
model-8	openssl	1.1.1, 1.1.1a, 1.1.1b, 1.1.1c, 1.1.1d, 1.1.1e, 1.1.1f, 1.1.1g
model-9	openssl	1.0.1e, 1.0.1f, 1.0.1g
model-10	openssl	1.0.1, 1.0.1a, 1.0.1b, 1.0.1c
model-11	mbedtls	2.0.0, 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7, 2.1.8, 2.1.9, 2.1.10, 2.1.11, 2.1.12, 2.1.13, 2.1.14, 2.1.15, 2.1.16, 2.1.17, 2.1.18, 2.2.0, 2.2.1, 2.3.0, 2.4.0, 2.4.1, 2.4.2, 2.5.0
model-12	mbedtls	2.11.0, 2.12.0, 2.13.0, 2.13.1, 2.14.0, 2.14.1, 2.15.0, 2.15.1, 2.16.0, 2.16.1, 2.16.2, 2.16.3, 2.16.4, 2.16.5, 2.16.6, 2.16.7, 2.16.8, 2.17.0, 2.18.0, 2.18.1, 2.19.0, 2.19.0d1, 2.19.0d2, 2.19.1, 2.20.0, 2.20.0d0, 2.20.0d1, 2.21.0, 2.22.0, 2.22.0d0, 2.23.0, 2.24.0, 3.0.0p1
model-13	mbedtls	2.5.1, 2.6.0, 2.6.1, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5, 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.7.12, 2.7.13, 2.7.14, 2.7.15, 2.7.16, 2.7.17, 2.8.0, 2.9.0, 2.10.0
model-14	mbedtls	1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.10, 1.2.11, 1.3.6, 1.3.7, 1.3.8
model-15	mbedtls	1.0.0, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8
model-16	mbedtls	1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5

A.3 TLS 1.2

Table A.5: Details of models learned for TLS 1.2

Model	Number of states	OpenSSL versions	mbed TLS versions
model-1	10	13	0
model-2	8	13	0
model-3	11	2	0
model-4	11	11	0
model-5	13	1	0
model-6	8	9	0
model-7	13	1	0
model-8	8	8	0
model-9	14	3	0
model-10	13	4	0
model-11	6	0	27
model-12	8	0	33
model-13	6	0	24
model-14	6	0	14
model-15	6	0	6

Table A.6: Implementation versions of models learned for TLS 1.2

Model	Implementation	Versions
model-1	openssl	1.0.2, 1.0.2a, 1.0.2b, 1.0.2c, 1.0.2d, 1.0.2e, 1.0.2f, 1.0.2g, 1.0.2h, 1.0.2i, 1.0.2j, 1.0.2k, 1.0.2l
model-2	openssl	1.1.0, 1.1.0a, 1.1.0b, 1.1.0c, 1.1.0d, 1.1.0e, 1.1.0f, 1.1.0g, 1.1.0h, 1.1.0i, 1.1.0j, 1.1.0k, 1.1.0l
model-3	openssl	1.0.1i, 1.0.1j
model-4	openssl	1.0.1k, 1.0.1l, 1.0.1m, 1.0.1n, 1.0.1o, 1.0.1p, 1.0.1q, 1.0.1r, 1.0.1s, 1.0.1t, 1.0.1u
model-5	openssl	1.0.1h
model-6	openssl	1.0.2m, 1.0.2n, 1.0.2o, 1.0.2p, 1.0.2q, 1.0.2r, 1.0.2s, 1.0.2t, 1.0.2u
model-7	openssl	1.0.1d
model-8	openssl	1.1.1, 1.1.1a, 1.1.1b, 1.1.1c, 1.1.1d, 1.1.1e, 1.1.1f, 1.1.1g
model-9	openssl	1.0.1e, 1.0.1f, 1.0.1g
model-10	openssl	1.0.1, 1.0.1a, 1.0.1b, 1.0.1c
model-11	mbedtls	2.0.0, 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7, 2.1.8, 2.1.9, 2.1.10, 2.1.11, 2.1.12, 2.1.13, 2.1.14, 2.1.15, 2.1.16, 2.1.17, 2.1.18, 2.2.0, 2.2.1, 2.3.0, 2.4.0, 2.4.1, 2.4.2, 2.5.0
model-12	mbedtls	2.11.0, 2.12.0, 2.13.0, 2.13.1, 2.14.0, 2.14.1, 2.15.0, 2.15.1, 2.16.0, 2.16.1, 2.16.2, 2.16.3, 2.16.4, 2.16.5, 2.16.6, 2.16.7, 2.16.8, 2.17.0, 2.18.0, 2.18.1, 2.19.0, 2.19.0d1, 2.19.0d2, 2.19.1, 2.20.0, 2.20.0d0, 2.20.0d1, 2.21.0, 2.22.0, 2.22.0d0, 2.23.0, 2.24.0, 3.0.0p1
model-13	mbedtls	2.5.1, 2.6.0, 2.6.1, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5, 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.7.12, 2.7.13, 2.7.14, 2.7.15, 2.7.16, 2.7.17, 2.8.0, 2.9.0, 2.10.0
model-14	mbedtls	1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.10, 1.2.11, 1.3.6, 1.3.7, 1.3.8
model-15	mbedtls	1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5

Appendix B

Model weights

B.1 TLS 1.0

Model	equal	count	recent
model-1	1	33	33
model-2	1	13	13
model-3	1	5	5
model-4	1	13	65
model-5	1	11	11
model-6	1	5	5
model-7	1	7	7
model-8	1	11	11
model-9	1	3	3
model-10	1	9	9
model-11	1	8	160
model-12	1	11	11
model-13	1	4	4
model-14	1	1	1
model-15	1	27	27
model-16	1	33	540
model-17	1	24	108
model-18	1	14	14
model-19	1	10	10
model-20	1	6	6

B.2 TLS 1.1

Model	equal	count	recent
model-1	1	13	13
model-2	1	13	65
model-3	1	2	2
model-4	1	11	11
model-5	1	1	1
model-6	1	9	9
model-7	1	1	1
model-8	1	8	160
model-9	1	3	3
model-10	1	4	4
model-11	1	27	27
model-12	1	33	540
model-13	1	24	108
model-14	1	14	14
model-15	1	10	10
model-16	1	6	6

B.3 TLS 1.2

Model	equal	count	recent
model-1	1	13	13
model-2	1	13	65
model-3	1	2	2
model-4	1	11	11
model-5	1	1	1
model-6	1	9	9
model-7	1	1	1
model-8	1	8	160
model-9	1	3	3
model-10	1	4	4
model-11	1	27	27
model-12	1	33	540
model-13	1	24	108
model-14	1	14	14
model-15	1	6	6

Appendix C

Benchmark results per model

C.1 TLS 1.0

C.1.1 Number of inputs

Table C.1: Number of inputs for each model of TLS 1.0

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	10	17	10	12	10
model-2	7	7	7	5	7
model-3	10	17	10	9	10
model-4	3	3	3	3	3
model-5	9	9	9	9	9
model-6	11	10	11	11	11
model-7	4	6	4	5	4
model-8	11	10	11	11	11
model-9	9	9	9	9	9
model-10	7	7	7	5	7
model-11	3	3	3	3	3
model-12	7	9	7	12	7
model-13	6	8	6	3	6
model-14	6	8	6	3	6
model-15	16	21	11	11	11
model-16	3	3	3	3	3
model-17	6	6	6	6	6
model-18	16	16	11	11	11
model-19	16	21	11	11	11
model-20	11	11	11	11	11

Table C.2: Number of inputs, distribution of values for each model of TLS 1.0 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	15.74	3.38	7	13	16	18	23
model-2	10.19	3.63	5	7	10	13	21
model-3	11.18	3.15	5	9	11	13	20
model-4	3.22	0.71	3	3	3	3	7
model-5	9.94	2.94	6	8	9	12	22
model-6	13.18	2.47	8	12	13	15	21
model-7	10.07	4.7	3	6	9	14	24
model-8	13.46	2.35	8	12	13	15	21
model-9	10.61	3.52	6	8	10	13	21
model-10	7.4	3.15	3	5	7	9.25	19
model-11	3.15	0.64	2	3	3	3	7
model-12	14.98	4.04	7	12	15	18	23
model-13	9.57	5.04	3	6	8	13	24
model-14	8.98	3.92	2	6	9	12	19
model-15	16.87	2.85	10	15	17	18	23
model-16	4.92	1.59	3	4	4	7	9
model-17	6.92	0.68	6	6	7	7	9
model-18	12.88	2.58	8	11	12	15	18
model-19	17.11	3.27	9	15	17	20	23
model-20	11.16	1.21	8	10	11	12	14

C.1.2 Number of resets

Table C.3: Number of resets for each model of TLS 1.0

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	2	3	2	3	2
model-2	3	3	3	2	3
model-3	2	3	2	2	2
model-4	1	1	1	1	1
model-5	3	3	3	3	3
model-6	3	3	3	3	3
model-7	1	2	1	1	1
model-8	3	3	3	3	3
model-9	3	3	3	3	3
model-10	3	3	3	2	3
model-11	1	1	1	1	1
model-12	2	3	2	3	2
model-13	2	3	2	1	2
model-14	2	3	2	1	2
model-15	4	5	3	3	3
model-16	1	1	1	1	1
model-17	2	2	2	2	2
model-18	4	4	3	3	3
model-19	4	5	3	3	3
model-20	3	3	3	3	3

Table C.4: Number of resets, distribution of values for each model of TLS 1.0 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	4.36	1.32	0	4	4	5	7
model-2	3.94	1.12	2	3	4	5	7
model-3	3.01	1.02	0	2	3	4	6
model-4	1.1	0.31	1	1	1	1	3
model-5	3.42	0.92	2	3	3	4	6
model-6	4.12	0.97	2	3	4	5	7
model-7	3.08	1.39	0	2	3	4	7
model-8	4.24	1.01	1	4	4	5	7
model-9	3.52	1.05	1	3	3.5	4	6
model-10	3.06	1.16	1	2	3	4	7
model-11	1.06	0.34	0	1	1	1	3
model-12	4.4	1.08	2	4	4	5	7
model-13	3.34	1.46	1	2	3	4	7
model-14	3.02	1.15	0	2	3	4	6
model-15	4.4	0.85	2	4	4	5	6
model-16	2	0.65	1	2	2	2	3
model-17	2.62	0.49	2	2	3	3	3
model-18	3.68	0.8	2	3	4	4	5
model-19	4.43	0.93	2	4	4.5	5	6
model-20	3.26	0.74	1	3	3	4	4

C.1.3 Computation time

Table C.5: Time in seconds for each model of TLS 1.0

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	0.0245	0.1975	0.1982	0.1787	0.3405
model-2	0.0296	0.0824	0.1398	0.1233	0.2643
model-3	0.0292	0.196	0.201	0.1642	0.3391
model-4	0.0204	0.041	0.0617	0.0625	0.1452
model-5	0.0296	0.1414	0.1468	0.1466	0.2736
model-6	0.0312	0.0835	0.1403	0.142	0.274
model-7	0.0269	0.1377	0.1188	0.1305	0.2329
model-8	0.0308	0.0853	0.1421	0.1431	0.2732
model-9	0.03	0.142	0.1469	0.1478	0.2788
model-10	0.0296	0.0825	0.1393	0.1237	0.2644
model-11	0.0204	0.0412	0.0632	0.0617	0.1466
model-12	0.029	0.1914	0.1919	0.1775	0.32
model-13	0.0283	0.1904	0.1867	0.1212	0.308
model-14	0.0287	0.1864	0.186	0.1231	0.3095
model-15	0.0315	0.108	0.159	0.1584	0.3151
model-16	0.0229	0.0761	0.1051	0.1063	0.2011
model-17	0.0262	0.0918	0.1401	0.1405	0.2658
model-18	0.0243	0.1053	0.1618	0.1605	0.3119
model-19	0.0316	0.1081	0.1598	0.1608	0.3156
model-20	0.0297	0.1008	0.1635	0.1596	0.3128

Table C.6: Time in seconds, distribution of values for each model of TLS 1.0 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	0.2168	0.058	0.0587	0.1757	0.2199	0.2507	0.3954
model-2	0.1312	0.0371	0.0568	0.0949	0.1389	0.1492	0.2298
model-3	0.1818	0.0526	0.0547	0.1459	0.184	0.2141	0.3236
model-4	0.0466	0.0172	0.0385	0.0404	0.0409	0.0427	0.1366
model-5	0.1523	0.0432	0.0807	0.1336	0.1448	0.1736	0.3
model-6	0.1353	0.0359	0.0792	0.1001	0.1414	0.1518	0.2709
model-7	0.177	0.062	0.0474	0.1341	0.1836	0.2146	0.3374
model-8	0.135	0.0373	0.0547	0.0979	0.1417	0.1512	0.235
model-9	0.1784	0.0546	0.0669	0.1436	0.1849	0.209	0.3727
model-10	0.1296	0.0353	0.0632	0.0937	0.1382	0.1468	0.2216
model-11	0.0454	0.0126	0.0398	0.0407	0.0417	0.0428	0.1039
model-12	0.2149	0.0515	0.0955	0.1846	0.2076	0.2503	0.419
model-13	0.1951	0.0634	0.0772	0.1375	0.1944	0.2404	0.3889
model-14	0.1862	0.0562	0.0528	0.1372	0.1865	0.2304	0.3516
model-15	0.1412	0.0317	0.0631	0.1066	0.1512	0.1612	0.2426
model-16	0.1188	0.0286	0.0729	0.082	0.1326	0.1363	0.1949
model-17	0.1309	0.0362	0.076	0.0928	0.1433	0.1494	0.2485
model-18	0.1447	0.0311	0.0879	0.1099	0.1554	0.1644	0.2529
model-19	0.1388	0.036	0.067	0.1065	0.1488	0.16	0.3044
model-20	0.1392	0.0395	0.0621	0.1017	0.1459	0.1596	0.3742

C.2 TLS 1.1

C.2.1 Number of inputs

Table C.7: Number of inputs for each model of TLS 1.1

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	7	7	5	7	5
model-2	3	3	3	3	3
model-3	9	10	9	8	9
model-4	6	6	4	5	4
model-5	9	10	9	8	9
model-6	7	7	5	7	5
model-7	8	9	12	11	12
model-8	3	3	3	3	3
model-9	8	9	12	11	12
model-10	4	4	7	6	7
model-11	16	21	11	11	11
model-12	3	3	3	3	3
model-13	6	6	6	6	6
model-14	16	16	11	11	11
model-15	16	21	11	11	11
model-16	11	11	11	11	11

Table C.8: Number of inputs, distribution of values for each model of TLS 1.1 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	8.98	2.92	5	7	8.5	11	19
model-2	3.2	0.64	3	3	3	3	7
model-3	9.66	2.46	6	8	9	11	16
model-4	9.35	2.94	3	7	9	11.25	19
model-5	10.6	3.13	6	8	10	13	18
model-6	7.5	2.85	3	5	7	9	17
model-7	10.59	2.69	5	9	10	12.25	18
model-8	3.25	0.71	2	3	3	3	6
model-9	12.66	2.33	7	11	13	14	18
model-10	9.9	4.01	3	7	9	13	21
model-11	17.16	2.99	9	16	17	19	23
model-12	4.76	1.57	2	4	4	6	8
model-13	6.89	0.62	6	6.75	7	7	9
model-14	12.78	2.74	7	11	12	15	18
model-15	17.2	3.26	9	15	17	20	24
model-16	11.18	1.33	7	10	11	12	13

C.2.2 Number of resets

Table C.9: Number of resets for each model of TLS 1.1

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	3	3	2	3	2
model-2	1	1	1	1	1
model-3	3	3	3	3	3
model-4	2	2	1	2	1
model-5	3	3	3	3	3
model-6	3	3	2	3	2
model-7	2	2	3	3	3
model-8	1	1	1	1	1
model-9	2	2	3	3	3
model-10	1	1	2	2	2
model-11	4	5	3	3	3
model-12	1	1	1	1	1
model-13	2	2	2	2	2
model-14	4	4	3	3	3
model-15	4	5	3	3	3
model-16	3	3	3	3	3

Table C.10: Number of resets, distribution of values for each model of TLS 1.1 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	3.71	1	2	3	4	4	7
model-2	1.1	0.31	1	1	1	1	3
model-3	3.16	0.81	1	3	3	4	5
model-4	3.37	1.14	0	3	3	4	6
model-5	3.5	0.94	1	3	3	4	5
model-6	3.16	1.04	1	2	3	4	6
model-7	2.99	0.98	1	2	3	4	6
model-8	1.12	0.34	0	1	1	1	2
model-9	3.64	0.88	1	3	4	4	6
model-10	3.04	1.21	0	2	3	4	6
model-11	4.46	0.9	2	4	5	5	6
model-12	1.95	0.68	0	2	2	2	3
model-13	2.64	0.48	2	2	3	3	3
model-14	3.6	0.88	1	3	4	4	5
model-15	4.46	0.96	2	4	4	5	6
model-16	3.22	0.71	1	3	3	4	4

C.2.3 Computation time

Table C.11: Time in seconds for each model of TLS 1.1

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	0.0233	0.075	0.0938	0.0977	0.1629
model-2	0.0171	0.0346	0.0521	0.0513	0.111
model-3	0.0239	0.0784	0.1438	0.1395	0.2173
model-4	0.0226	0.0748	0.0985	0.0976	0.1686
model-5	0.0236	0.0783	0.1454	0.1414	0.2199
model-6	0.0232	0.0746	0.0941	0.099	0.1623
model-7	0.0227	0.0712	0.1562	0.1547	0.2324
model-8	0.0169	0.0342	0.0526	0.0509	0.1092
model-9	0.0223	0.0682	0.1565	0.1501	0.2363
model-10	0.0211	0.0614	0.1413	0.1347	0.2148
model-11	0.0284	0.1038	0.1556	0.1552	0.2822
model-12	0.0191	0.0673	0.0957	0.0955	0.1659
model-13	0.0228	0.0838	0.1296	0.1309	0.2299
model-14	0.0283	0.0976	0.1496	0.151	0.2775
model-15	0.0275	0.0989	0.1496	0.1501	0.2765
model-16	0.0256	0.0934	0.1499	0.1501	0.2784

Table C.12: Time in seconds, distribution of values for each model of TLS 1.1 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	0.1057	0.0266	0.0467	0.0796	0.1125	0.12	0.1647
model-2	0.0374	0.0113	0.0317	0.0334	0.0341	0.0349	0.0987
model-3	0.1141	0.0342	0.0447	0.0871	0.1161	0.1329	0.2478
model-4	0.1057	0.0278	0.0427	0.078	0.1139	0.1231	0.1768
model-5	0.1285	0.0345	0.0627	0.1067	0.1304	0.1507	0.2428
model-6	0.1043	0.0283	0.0448	0.0914	0.1096	0.12	0.179
model-7	0.122	0.0376	0.0512	0.0977	0.12	0.14	0.2857
model-8	0.0379	0.0111	0.0323	0.0338	0.0346	0.0348	0.0934
model-9	0.13	0.0346	0.0482	0.1148	0.1323	0.1452	0.2764
model-10	0.1252	0.0354	0.0421	0.1049	0.1301	0.1451	0.2445
model-11	0.1318	0.0329	0.0792	0.0976	0.1432	0.1521	0.2507
model-12	0.1044	0.0245	0.0408	0.0743	0.1166	0.1202	0.1798
model-13	0.114	0.0259	0.0686	0.0853	0.1224	0.1343	0.1912
model-14	0.1278	0.0295	0.078	0.0987	0.1344	0.1484	0.2174
model-15	0.1278	0.0305	0.0619	0.0982	0.1361	0.1462	0.2181
model-16	0.1244	0.0291	0.0531	0.094	0.1304	0.1446	0.2209

C.3 TLS 1.2

C.3.1 Number of inputs

Table C.13: Number of inputs for each model of TLS 1.2

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	7	7	5	7	5
model-2	3	3	3	3	3
model-3	9	10	9	8	9
model-4	6	6	4	5	4
model-5	9	10	9	8	9
model-6	7	7	5	7	5
model-7	8	9	12	11	12
model-8	3	3	3	3	3
model-9	8	9	12	11	12
model-10	4	4	7	6	7
model-11	16	16	11	11	11
model-12	3	3	3	3	3
model-13	6	6	6	6	6
model-14	16	16	11	11	11
model-15	11	11	11	11	11

Table C.14: Number of inputs, distribution of values for each model of TLS 1.2 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	8.89	2.9	5	7	8	11	17
model-2	3.16	0.6	3	3	3	3	7
model-3	9.67	2.44	6	8	9	11	17
model-4	9.61	2.97	3	8	9	12	18
model-5	10.12	2.6	5	8	10	12	16
model-6	6.73	2.7	3	5	6	8	16
model-7	10.48	2.59	5	9	10	13	16
model-8	3.19	0.66	2	3	3	3	6
model-9	12.54	2.52	8	10	13	14	19
model-10	9.54	3.94	3	6	9	13	20
model-11	12.56	2.35	9	11	12	14	18
model-12	4.89	1.58	2	4	4	6	8
model-13	6.92	0.68	6	6	7	7	9
model-14	12.36	2.55	8	11	12	13	18
model-15	11.18	1.29	6	10	11	12	15

C.3.2 Number of resets

Table C.15: Number of resets for each model of TLS 1.2

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	3	3	2	3	2
model-2	1	1	1	1	1
model-3	3	3	3	3	3
model-4	2	2	1	2	1
model-5	3	3	3	3	3
model-6	3	3	2	3	2
model-7	2	2	3	3	3
model-8	1	1	1	1	1
model-9	2	2	3	3	3
model-10	1	1	2	2	2
model-11	4	4	3	3	3
model-12	1	1	1	1	1
model-13	2	2	2	2	2
model-14	4	4	3	3	3
model-15	3	3	3	3	3

Table C.16: Number of resets, distribution of values for each model of TLS 1.2 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	3.62	0.93	2	3	4	4	6
model-2	1.08	0.29	1	1	1	1	3
model-3	3.2	0.84	2	3	3	4	5
model-4	3.44	0.98	0	3	3	4	6
model-5	3.4	0.8	1	3	3	4	5
model-6	2.9	1.06	1	2	3	4	6
model-7	2.98	0.95	0	2	3	4	5
model-8	1.08	0.34	0	1	1	1	2
model-9	3.63	0.94	1	3	4	4	6
model-10	2.91	1.2	0	2	3	4	5
model-11	3.5	0.76	2	3	3	4	5
model-12	1.94	0.69	0	1	2	2	3
model-13	2.63	0.48	2	2	3	3	3
model-14	3.54	0.87	1	3	4	4	5
model-15	3.22	0.71	1	3	3	4	4

C.3.3 Computation time

Table C.17: Time in seconds for each model of TLS 1.2

Model	ADG	HDT First	HDT Gini (count)	HDT Gini (equal)	HDT Gini (recent)
model-1	0.0223	0.0725	0.0936	0.0967	0.1603
model-2	0.016	0.0336	0.0517	0.0508	0.1075
model-3	0.0232	0.0765	0.1437	0.1393	0.2141
model-4	0.0221	0.0737	0.0961	0.0969	0.163
model-5	0.0229	0.0777	0.1425	0.14	0.2132
model-6	0.0224	0.0733	0.0946	0.0982	0.1597
model-7	0.0219	0.068	0.155	0.1506	0.2281
model-8	0.0161	0.0343	0.0514	0.0518	0.1049
model-9	0.0219	0.0675	0.1546	0.1508	0.2295
model-10	0.0208	0.0606	0.1416	0.1355	0.2132
model-11	0.0271	0.0971	0.1491	0.1489	0.2631
model-12	0.0186	0.0675	0.0948	0.0953	0.1604
model-13	0.0218	0.0843	0.1286	0.1264	0.2178
model-14	0.0268	0.0964	0.1494	0.1488	0.2628
model-15	0.0249	0.0917	0.1498	0.1489	0.2625

Table C.18: Time in seconds, distribution of values for each model of TLS 1.2 with HDT Random

Model	mean	std	min	25%	50%	75%	max
model-1	0.1049	0.0277	0.0472	0.0792	0.1126	0.1207	0.2001
model-2	0.0379	0.013	0.032	0.0341	0.0343	0.0345	0.104
model-3	0.1158	0.033	0.0533	0.0872	0.1172	0.1349	0.2296
model-4	0.1035	0.027	0.0424	0.0759	0.1127	0.118	0.1833
model-5	0.1248	0.0316	0.0602	0.1052	0.1281	0.1406	0.2408
model-6	0.1047	0.0257	0.0471	0.0941	0.1112	0.1197	0.1674
model-7	0.1187	0.0335	0.0442	0.1023	0.12	0.1377	0.2372
model-8	0.0376	0.0111	0.0308	0.0342	0.0343	0.0345	0.1064
model-9	0.1319	0.0338	0.0506	0.115	0.1375	0.1491	0.2453
model-10	0.1214	0.0372	0.0415	0.0933	0.129	0.1443	0.2312
model-11	0.1229	0.0269	0.0753	0.0936	0.1308	0.1429	0.2023
model-12	0.1019	0.0242	0.0412	0.0726	0.1164	0.1201	0.1417
model-13	0.1148	0.0269	0.0703	0.0848	0.1278	0.1347	0.1991
model-14	0.1263	0.0285	0.053	0.0983	0.1347	0.1442	0.2129
model-15	0.1194	0.0267	0.0727	0.092	0.128	0.1419	0.1836

Appendix D

Weighted benchmark statistics

D.1 Weight function "equal"

D.1.1 Number of inputs

Table D.1: Benchmark summary: Number of inputs with weight function 'equal' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	8.22	4.15	3	6	7	11	16
HDT First	9.16	5.18	3	6	9	11	21
HDT Gini (count)	7.63	3.2	3	5	7	11	12
HDT Gini (equal)	7.57	3.2	3	5	8	11	12
HDT Gini (recent)	7.63	3.2	3	5	7	11	12
HDT Random	9.78	4.67	2	6	10	13	24

Table D.2: Benchmark summary: Number of inputs with weight function 'equal' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	8.55	4.06	3	6	8	11	16
HDT First	10.05	5.41	3	6	9	13	21
HDT Gini (count)	7.8	2.91	3	6	8	11	11
HDT Gini (equal)	7.65	3.48	3	4	9	11	12
HDT Gini (recent)	7.8	2.91	3	6	8	11	11
HDT Random	10.58	5.03	2	7	11	14	24

Table D.3: Benchmark summary: Number of inputs with weight function 'equal' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	8.25	4.35	3	5	7	10	16
HDT First	9.12	5.6	3	5	8	10	21
HDT Gini (count)	7.62	3.37	3	4	8	11	12
HDT Gini (equal)	7.62	3.02	3	5	7	11	11
HDT Gini (recent)	7.62	3.37	3	4	8	11	12

Method	mean	std	min	25%	50%	75%	max
HDT Random	9.73	4.73	2	6	9	13	24

Table D.4: Benchmark summary: Number of inputs with weight function 'equal' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	7.73	3.99	3	4	7	9	16
HDT First	8	4.07	3	4	7	10	16
HDT Gini (count)	7.4	3.36	3	4	7	11	12
HDT Gini (equal)	7.4	2.98	3	5	7	11	11
HDT Gini (recent)	7.4	3.36	3	4	7	11	12
HDT Random	8.79	3.85	2	6	9	12	20

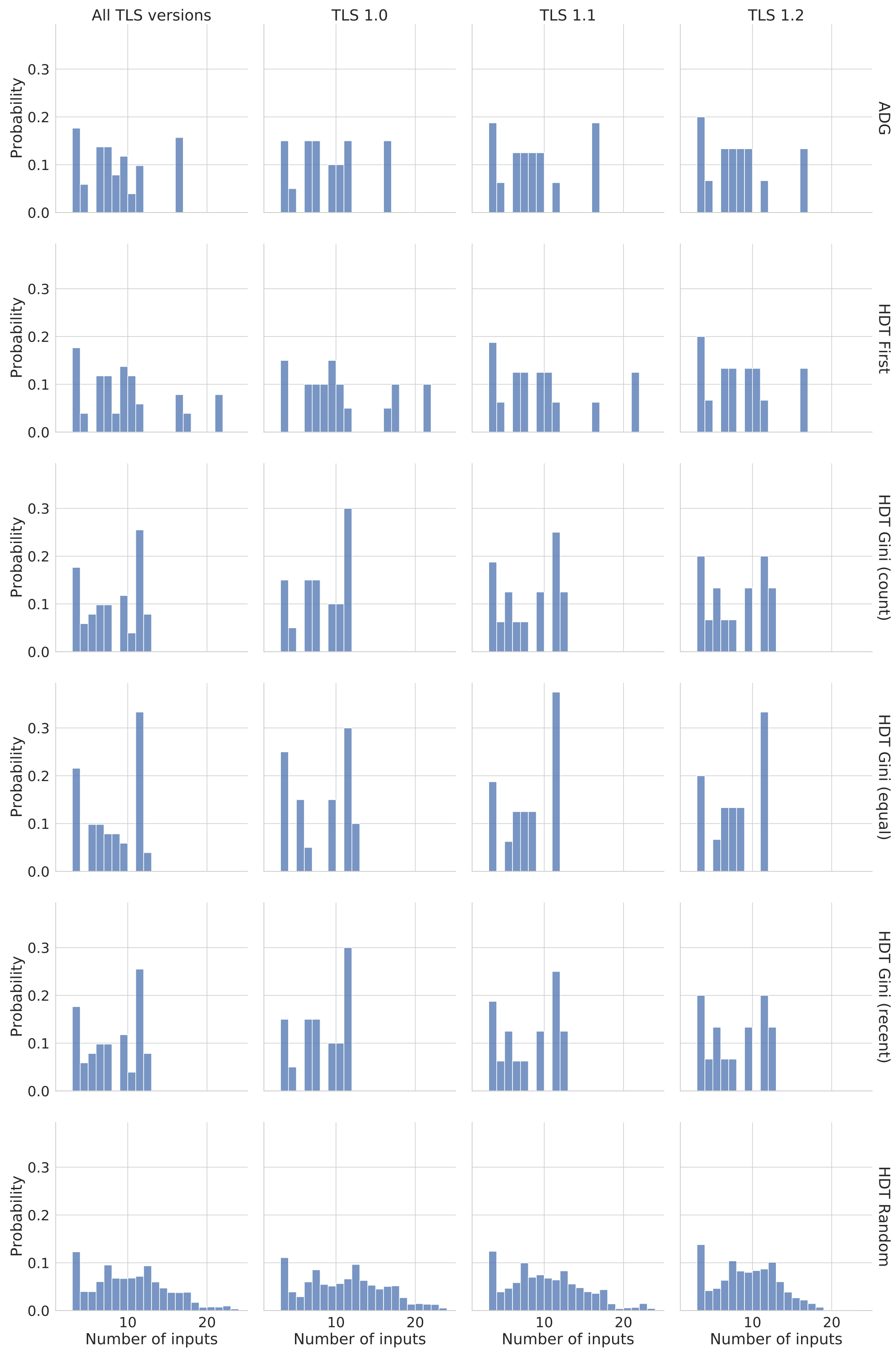


Figure D.1: Benchmark results: Number of inputs with weight function ‘equal’

D.1.2 Number of resets

Table D.5: Benchmark summary: Number of resets with weight function 'equal' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	2.41	1.01	1	2	2	3	4
HDT First	2.61	1.14	1	2	3	3	5
HDT Gini (count)	2.25	0.81	1	2	2	3	3
HDT Gini (equal)	2.33	0.83	1	2	3	3	3
HDT Gini (recent)	2.25	0.81	1	2	2	3	3
HDT Random	3.1	1.31	0	2	3	4	7

Table D.6: Benchmark summary: Number of resets with weight function 'equal' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	2.45	0.97	1	2	2	3	4
HDT First	2.85	1.06	1	2	3	3	5
HDT Gini (count)	2.3	0.78	1	2	2	3	3
HDT Gini (equal)	2.2	0.87	1	1	2	3	3
HDT Gini (recent)	2.3	0.78	1	2	2	3	3
HDT Random	3.3	1.39	0	2	3	4	7

Table D.7: Benchmark summary: Number of resets with weight function 'equal' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	2.44	1.06	1	1	2	3	4
HDT First	2.56	1.27	1	1	2	3	5
HDT Gini (count)	2.25	0.83	1	1	2	3	3
HDT Gini (equal)	2.44	0.79	1	2	3	3	3
HDT Gini (recent)	2.25	0.83	1	1	2	3	3
HDT Random	3.07	1.28	0	2	3	4	7

Table D.8: Benchmark summary: Number of resets with weight function 'equal' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	2.33	1.01	1	1	2	3	4
HDT First	2.33	1.01	1	1	2	3	4
HDT Gini (count)	2.2	0.83	1	1	2	3	3
HDT Gini (equal)	2.4	0.8	1	2	3	3	3
HDT Gini (recent)	2.2	0.83	1	1	2	3	3
HDT Random	2.87	1.16	0	2	3	4	6

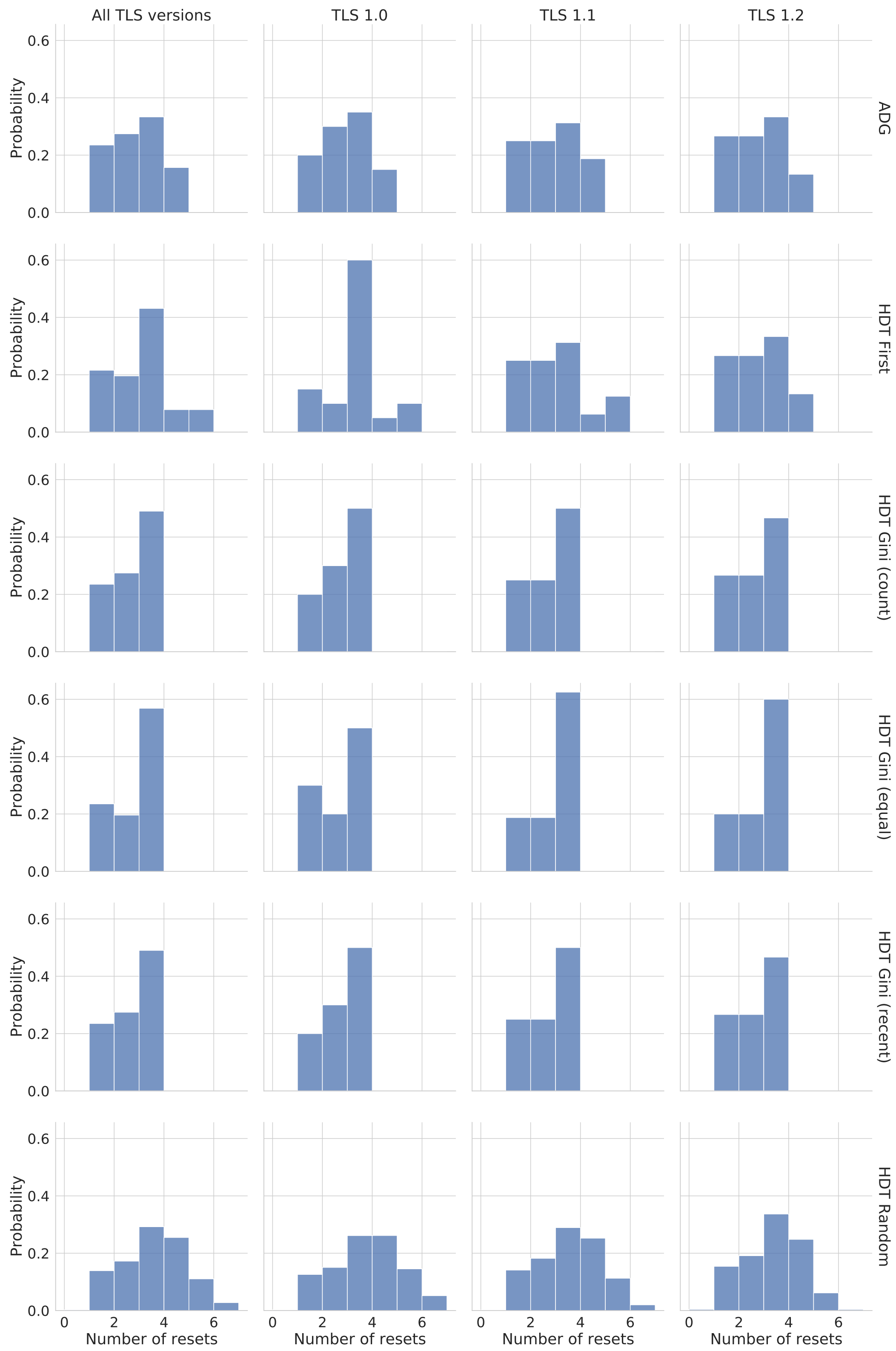


Figure D.2: Benchmark results: Number of inputs with weight function 'equal'

D.1.3 Computation time

Table D.9: Benchmark summary: Time in seconds with weight function 'equal' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	0.0245	0.0046	0.0155	0.0219	0.0235	0.0286	0.088
HDT First	0.0914	0.0414	0.032	0.0682	0.0811	0.0996	0.2644
HDT Gini (count)	0.1316	0.0391	0.0481	0.0967	0.1423	0.154	0.3045
HDT Gini (equal)	0.1267	0.0346	0.0486	0.098	0.139	0.1499	0.249
HDT Gini (recent)	0.2313	0.0645	0.0984	0.1663	0.2315	0.2761	0.4352
HDT Random	0.1229	0.0534	0.0308	0.0871	0.1244	0.1489	0.419

Table D.10: Benchmark summary: Time in seconds with weight function 'equal' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	0.0277	0.0045	0.0155	0.0266	0.0293	0.0299	0.088
HDT First	0.1194	0.0498	0.0387	0.083	0.106	0.1831	0.2644
HDT Gini (count)	0.1476	0.0387	0.0587	0.1382	0.1457	0.1833	0.3045
HDT Gini (equal)	0.1366	0.0322	0.0589	0.1217	0.1418	0.1596	0.2374
HDT Gini (recent)	0.2746	0.0553	0.1401	0.2631	0.2771	0.3134	0.4352
HDT Random	0.147	0.0617	0.0385	0.1016	0.1458	0.1837	0.419

Table D.11: Benchmark summary: Time in seconds with weight function 'equal' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	0.023	0.0033	0.0162	0.0216	0.023	0.0247	0.029
HDT First	0.0747	0.0201	0.0321	0.0675	0.0748	0.0913	0.198
HDT Gini (count)	0.1228	0.0358	0.0494	0.0947	0.1417	0.1494	0.2607
HDT Gini (equal)	0.1219	0.0351	0.0486	0.0971	0.1369	0.1496	0.249
HDT Gini (recent)	0.2091	0.0557	0.1054	0.1641	0.2174	0.2728	0.3643
HDT Random	0.1088	0.0408	0.0317	0.0806	0.1162	0.1373	0.2857

Table D.12: Benchmark summary: Time in seconds with weight function 'equal' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	0.0219	0.0031	0.0156	0.0209	0.0221	0.0232	0.0279
HDT First	0.0716	0.0185	0.032	0.067	0.0733	0.0831	0.1553
HDT Gini (count)	0.1198	0.0357	0.0481	0.0941	0.14	0.1488	0.2583
HDT Gini (equal)	0.1186	0.0342	0.0489	0.096	0.1349	0.1482	0.209
HDT Gini (recent)	0.1973	0.0507	0.0984	0.1597	0.2129	0.2291	0.3349
HDT Random	0.1058	0.0396	0.0308	0.0761	0.1149	0.1344	0.2453

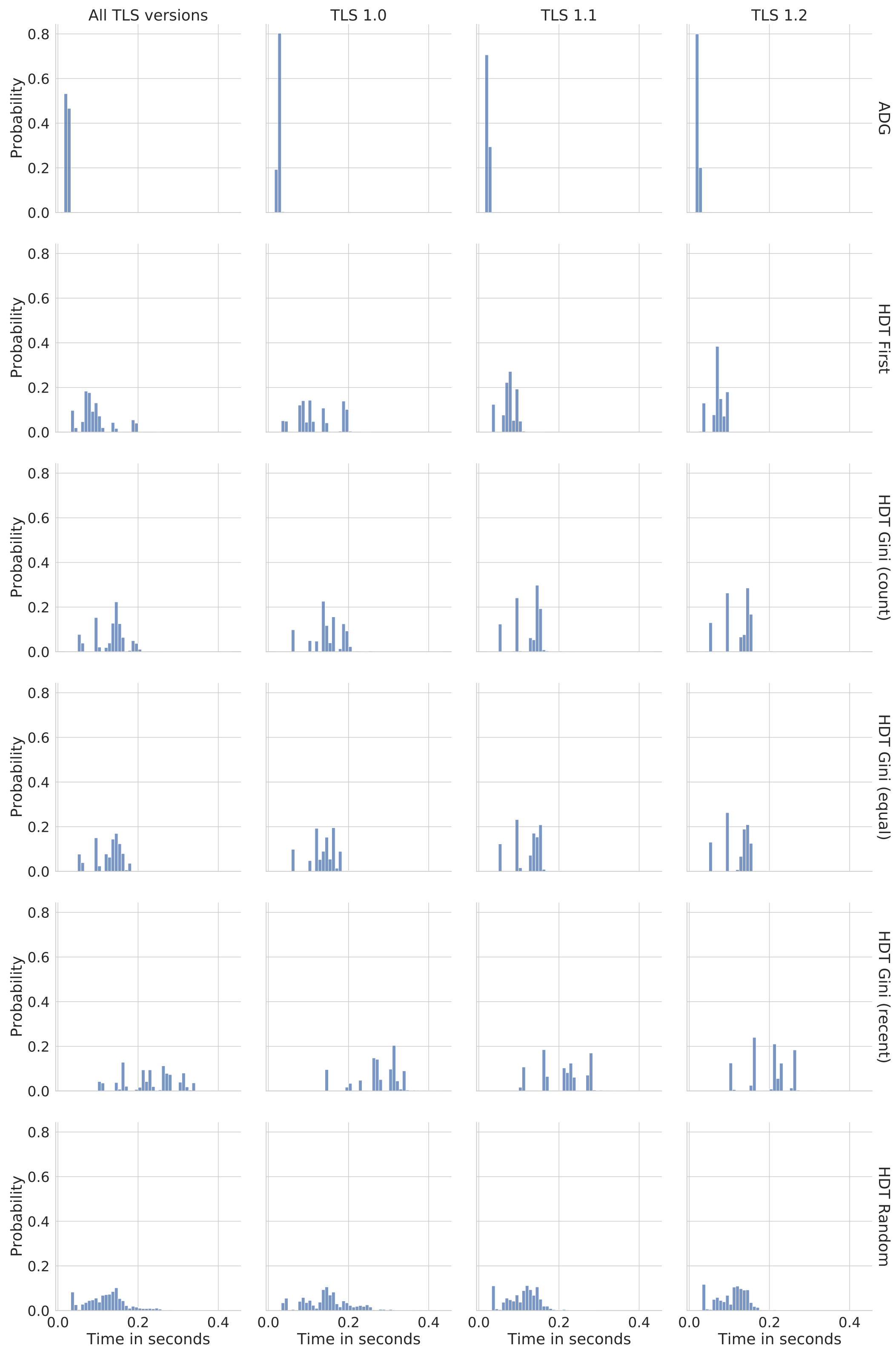


Figure D.3: Benchmark results: Computation time with weight function 'equal'

D.2 Weight function "count"

D.2.1 Number of inputs

Table D.13: Benchmark summary: Number of inputs with weight function 'count' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	8.33	4.9	3	3	7	11	16
HDT First	9.47	6.3	3	3	7	16	21
HDT Gini (count)	7	3.32	3	3	6	11	12
HDT Gini (equal)	7.26	3.44	3	3	6	11	12
HDT Gini (recent)	7	3.32	3	3	6	11	12
HDT Random	9.61	5.14	2	6	8	13	24

Table D.14: Benchmark summary: Number of inputs with weight function 'count' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	8.71	4.54	3	6	9	11	16
HDT First	10.65	6.39	3	6	9	17	21
HDT Gini (count)	7.68	3.12	3	6	9	11	11
HDT Gini (equal)	7.94	3.6	3	5	9	11	12
HDT Gini (recent)	7.68	3.12	3	6	9	11	11
HDT Random	10.81	5.43	2	6	11	15	24

Table D.15: Benchmark summary: Number of inputs with weight function 'count' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	8.28	5.22	3	3	6	16	16
HDT First	9.36	6.88	3	3	6	16	21
HDT Gini (count)	6.65	3.41	3	3	6	11	12
HDT Gini (equal)	6.89	3.27	3	3	6	11	11
HDT Gini (recent)	6.65	3.41	3	3	6	11	12
HDT Random	9.36	5.33	2	5	8	13	24

Table D.16: Benchmark summary: Number of inputs with weight function 'count' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	7.83	5.02	3	3	6	11	16
HDT First	7.87	5.02	3	3	6	11	16
HDT Gini (count)	6.39	3.33	3	3	5	11	12
HDT Gini (equal)	6.65	3.2	3	3	6	11	11
HDT Gini (recent)	6.39	3.33	3	3	5	11	12
HDT Random	8.11	3.92	2	5	7	11	20

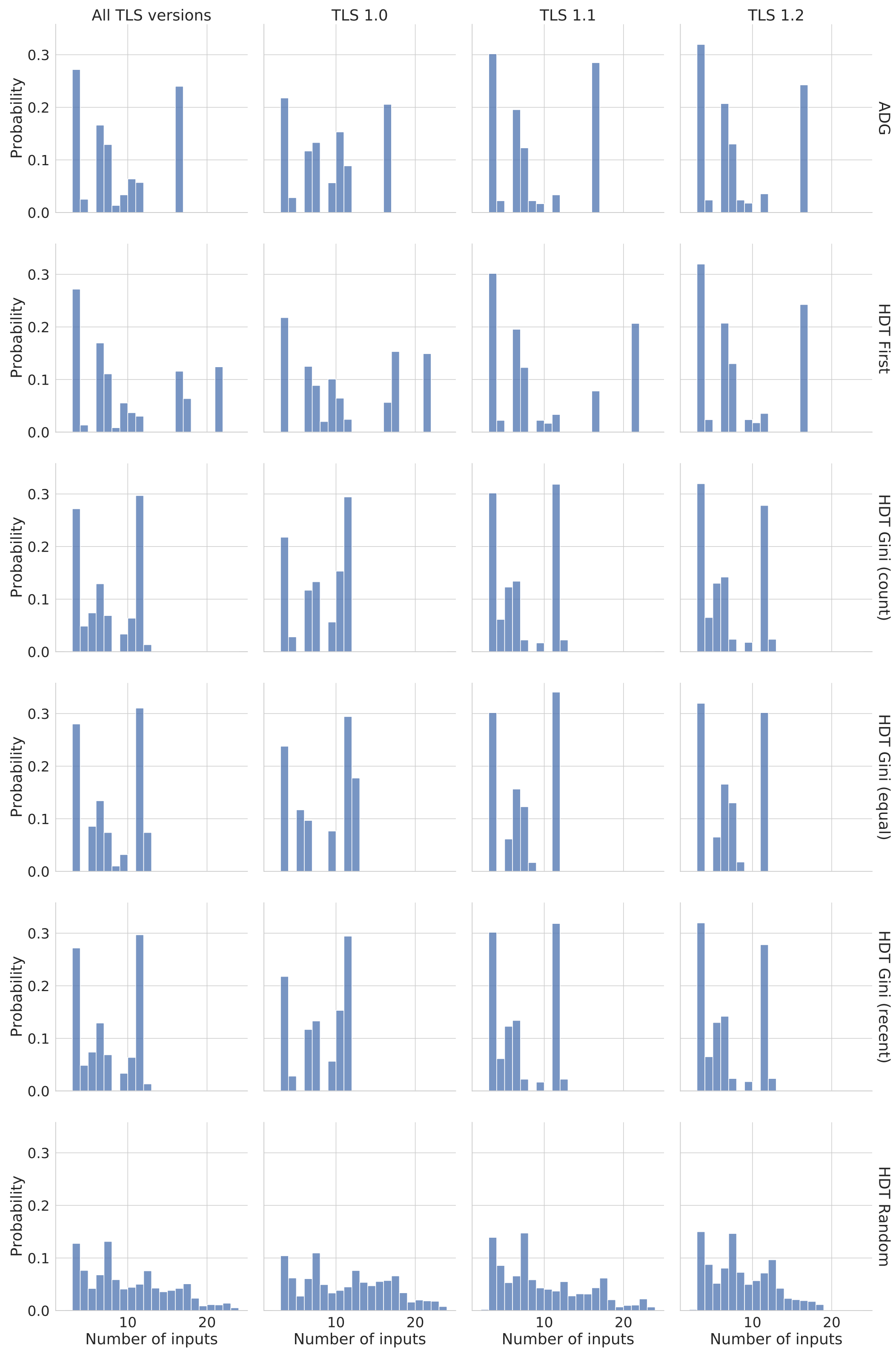


Figure D.4: Benchmark results: Number of inputs with weight function ‘count’

D.2.2 Number of resets

Table D.17: Benchmark summary: Number of resets with weight function 'count' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	2.38	1.14	1	1	2	3	4
HDT First	2.61	1.34	1	1	3	3	5
HDT Gini (count)	2.06	0.84	1	1	2	3	3
HDT Gini (equal)	2.2	0.86	1	1	2	3	3
HDT Gini (recent)	2.06	0.84	1	1	2	3	3
HDT Random	3.05	1.35	0	2	3	4	7

Table D.18: Benchmark summary: Number of resets with weight function 'count' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	2.4	1.07	1	2	2	3	4
HDT First	2.79	1.27	1	2	3	3	5
HDT Gini (count)	2.19	0.8	1	2	2	3	3
HDT Gini (equal)	2.26	0.85	1	1	3	3	3
HDT Gini (recent)	2.19	0.8	1	2	2	3	3
HDT Random	3.33	1.43	0	2	3	4	7

Table D.19: Benchmark summary: Number of resets with weight function 'count' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	2.42	1.21	1	1	2	4	4
HDT First	2.63	1.51	1	1	2	4	5
HDT Gini (count)	1.99	0.85	1	1	2	3	3
HDT Gini (equal)	2.18	0.87	1	1	2	3	3
HDT Gini (recent)	1.99	0.85	1	1	2	3	3
HDT Random	2.99	1.35	0	2	3	4	7

Table D.20: Benchmark summary: Number of resets with weight function 'count' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	2.33	1.18	1	1	2	3	4
HDT First	2.33	1.18	1	1	2	3	4
HDT Gini (count)	1.93	0.84	1	1	2	3	3
HDT Gini (equal)	2.13	0.87	1	1	2	3	3
HDT Gini (recent)	1.93	0.84	1	1	2	3	3
HDT Random	2.72	1.14	0	2	3	4	6

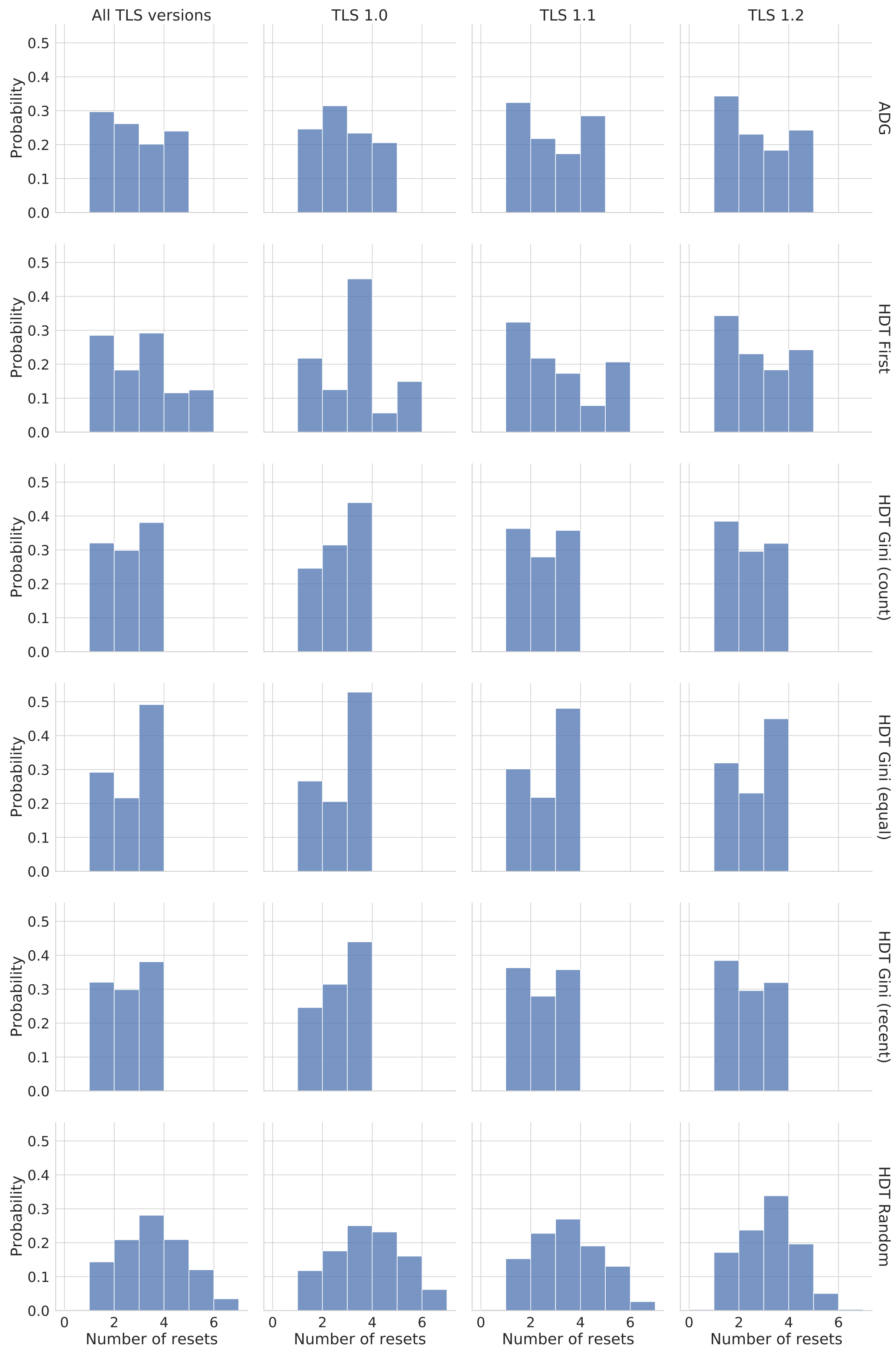


Figure D.5: Benchmark results: Number of inputs with weight function ‘count’

D.2.3 Computation time

Table D.21: Benchmark summary: Time in seconds with weight function ‘count’ for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	0.0243	0.0048	0.0155	0.0208	0.0233	0.0285	0.088
HDT First	0.092	0.0399	0.032	0.0719	0.0839	0.0998	0.2644
HDT Gini (count)	0.1275	0.0392	0.0481	0.0954	0.1384	0.15	0.3045
HDT Gini (equal)	0.1249	0.0357	0.0486	0.0966	0.1305	0.1502	0.249
HDT Gini (recent)	0.2309	0.0676	0.0984	0.164	0.2322	0.2771	0.4352
HDT Random	0.1219	0.0522	0.0308	0.0863	0.122	0.1467	0.419

Table D.22: Benchmark summary: Time in seconds with weight function ‘count’ for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	0.0269	0.0049	0.0155	0.0231	0.0291	0.03	0.088
HDT First	0.1142	0.0488	0.0387	0.0815	0.1039	0.1411	0.2644
HDT Gini (count)	0.1455	0.0388	0.0587	0.1362	0.1445	0.1617	0.3045
HDT Gini (equal)	0.1392	0.0334	0.0589	0.1214	0.143	0.1605	0.2374
HDT Gini (recent)	0.2728	0.0582	0.1401	0.2604	0.2731	0.316	0.4352
HDT Random	0.145	0.0602	0.0385	0.1017	0.1441	0.169	0.419

Table D.23: Benchmark summary: Time in seconds with weight function ‘count’ for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	0.0231	0.0039	0.0162	0.0192	0.0229	0.0275	0.029
HDT First	0.0777	0.0215	0.0321	0.0674	0.075	0.0972	0.198
HDT Gini (count)	0.1164	0.0347	0.0494	0.0945	0.1279	0.149	0.2607
HDT Gini (equal)	0.1166	0.0346	0.0486	0.0955	0.1284	0.1494	0.249
HDT Gini (recent)	0.2075	0.0597	0.1054	0.164	0.2153	0.2759	0.3643
HDT Random	0.1072	0.0386	0.0317	0.0798	0.1159	0.1343	0.2857

Table D.24: Benchmark summary: Time in seconds with weight function ‘count’ for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	0.0219	0.0036	0.0156	0.0187	0.022	0.0252	0.0279
HDT First	0.0748	0.0195	0.032	0.0672	0.0735	0.0922	0.1553
HDT Gini (count)	0.1127	0.0333	0.0481	0.0936	0.0967	0.1477	0.2583
HDT Gini (equal)	0.1126	0.0325	0.0489	0.0953	0.0984	0.1474	0.209
HDT Gini (recent)	0.1943	0.0535	0.0984	0.1594	0.1641	0.2602	0.3349
HDT Random	0.1036	0.0368	0.0308	0.0752	0.1146	0.1328	0.2453

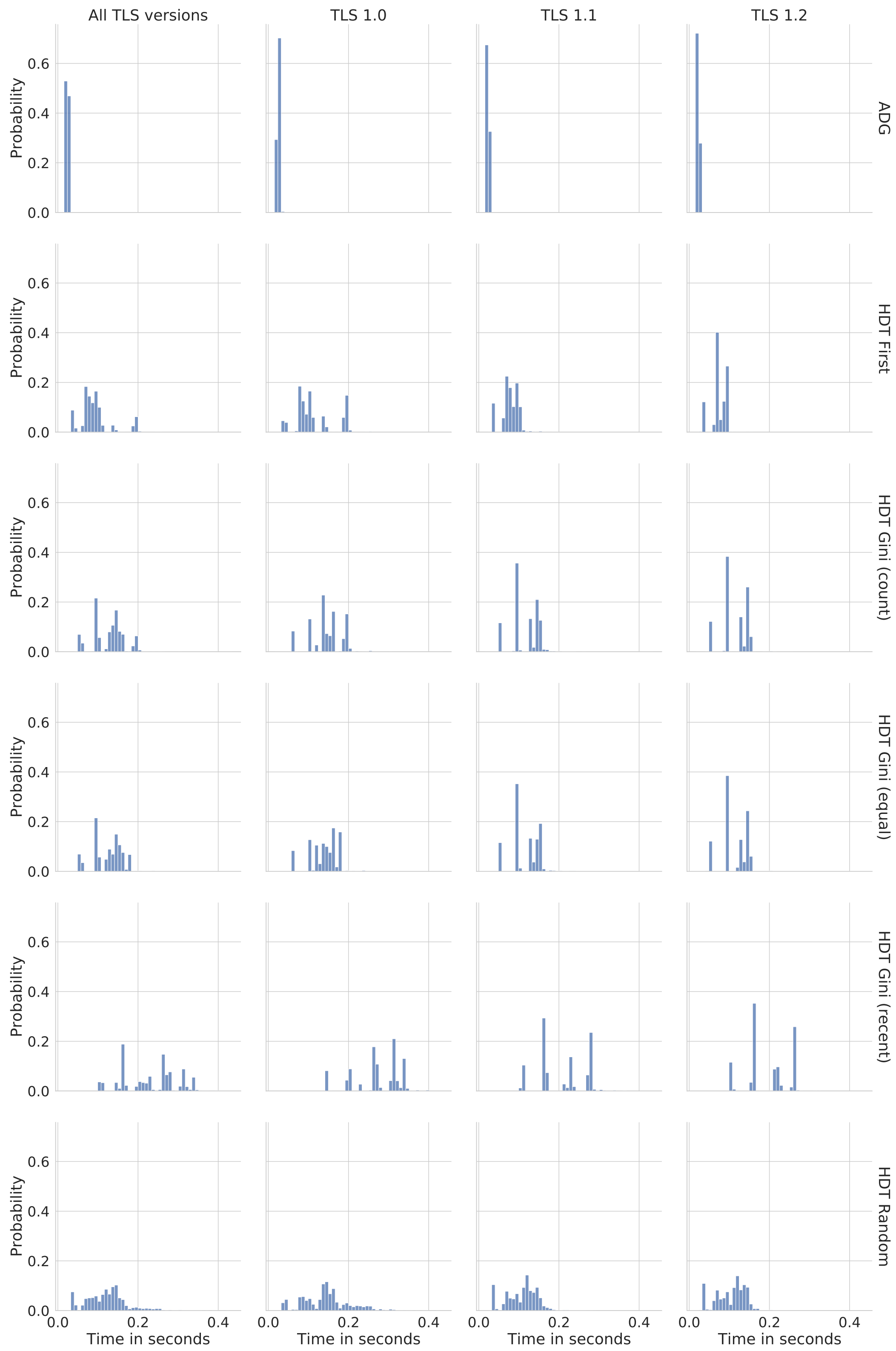


Figure D.6: Benchmark results: Computation time with weight function ‘count’

D.3 Weight function "recent"

D.3.1 Number of inputs

Table D.25: Benchmark summary: Number of inputs with weight function 'recent' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	4.32	3.08	3	3	3	3	16
HDT First	4.55	3.83	3	3	3	3	21
HDT Gini (count)	4.05	2.25	3	3	3	3	12
HDT Gini (equal)	4.1	2.35	3	3	3	3	12
HDT Gini (recent)	4.05	2.25	3	3	3	3	12
HDT Random	5.64	3.39	2	3	4	7	24

Table D.26: Benchmark summary: Number of inputs with weight function 'recent' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	4.6	3.29	3	3	3	6	16
HDT First	5.06	4.48	3	3	3	6	21
HDT Gini (count)	4.35	2.53	3	3	3	6	11
HDT Gini (equal)	4.42	2.76	3	3	3	5	12
HDT Gini (recent)	4.35	2.53	3	3	3	6	11
HDT Random	6.14	4	2	3	5	7	24

Table D.27: Benchmark summary: Number of inputs with weight function 'recent' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	4.23	3.07	3	3	3	3	16
HDT First	4.43	3.85	3	3	3	3	21
HDT Gini (count)	3.93	2.12	3	3	3	3	12
HDT Gini (equal)	3.97	2.14	3	3	3	3	11
HDT Gini (recent)	3.93	2.12	3	3	3	3	12
HDT Random	5.47	3.3	2	3	4	7	24

Table D.28: Benchmark summary: Number of inputs with weight function 'recent' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	4.11	2.84	3	3	3	3	16
HDT First	4.12	2.85	3	3	3	3	16
HDT Gini (count)	3.86	2	3	3	3	3	12
HDT Gini (equal)	3.9	2.03	3	3	3	3	11
HDT Gini (recent)	3.86	2	3	3	3	3	12
HDT Random	5.27	2.6	2	3	4	7	20

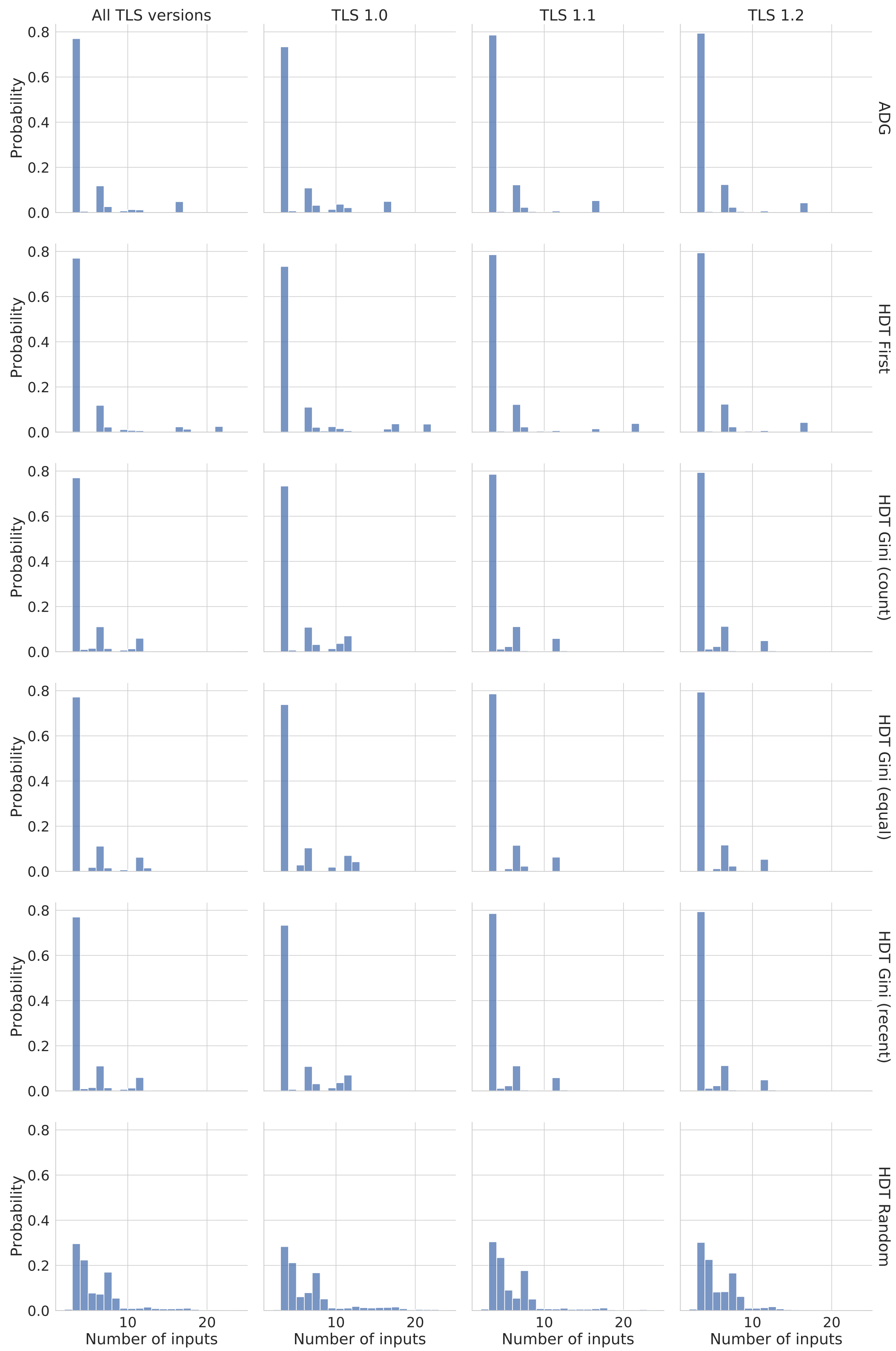


Figure D.7: Benchmark results: Number of inputs with weight function 'recent'

D.3.2 Number of resets

Table D.29: Benchmark summary: Number of resets with weight function 'recent' for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	1.36	0.77	1	1	1	1	4
HDT First	1.41	0.89	1	1	1	1	5
HDT Gini (count)	1.3	0.6	1	1	1	1	3
HDT Gini (equal)	1.32	0.64	1	1	1	1	3
HDT Gini (recent)	1.3	0.6	1	1	1	1	3
HDT Random	2.06	1.03	0	1	2	3	7

Table D.30: Benchmark summary: Number of resets with weight function 'recent' for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	1.41	0.8	1	1	1	2	4
HDT First	1.51	0.99	1	1	1	2	5
HDT Gini (count)	1.36	0.66	1	1	1	2	3
HDT Gini (equal)	1.38	0.7	1	1	1	2	3
HDT Gini (recent)	1.36	0.66	1	1	1	2	3
HDT Random	2.19	1.15	0	1	2	3	7

Table D.31: Benchmark summary: Number of resets with weight function 'recent' for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	1.35	0.78	1	1	1	1	4
HDT First	1.39	0.92	1	1	1	1	5
HDT Gini (count)	1.27	0.57	1	1	1	1	3
HDT Gini (equal)	1.3	0.62	1	1	1	1	3
HDT Gini (recent)	1.27	0.57	1	1	1	1	3
HDT Random	2.02	1	0	1	2	3	7

Table D.32: Benchmark summary: Number of resets with weight function 'recent' for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	1.32	0.73	1	1	1	1	4
HDT First	1.32	0.73	1	1	1	1	4
HDT Gini (count)	1.25	0.55	1	1	1	1	3
HDT Gini (equal)	1.29	0.6	1	1	1	1	3
HDT Gini (recent)	1.25	0.55	1	1	1	1	3
HDT Random	1.95	0.91	0	1	2	3	6

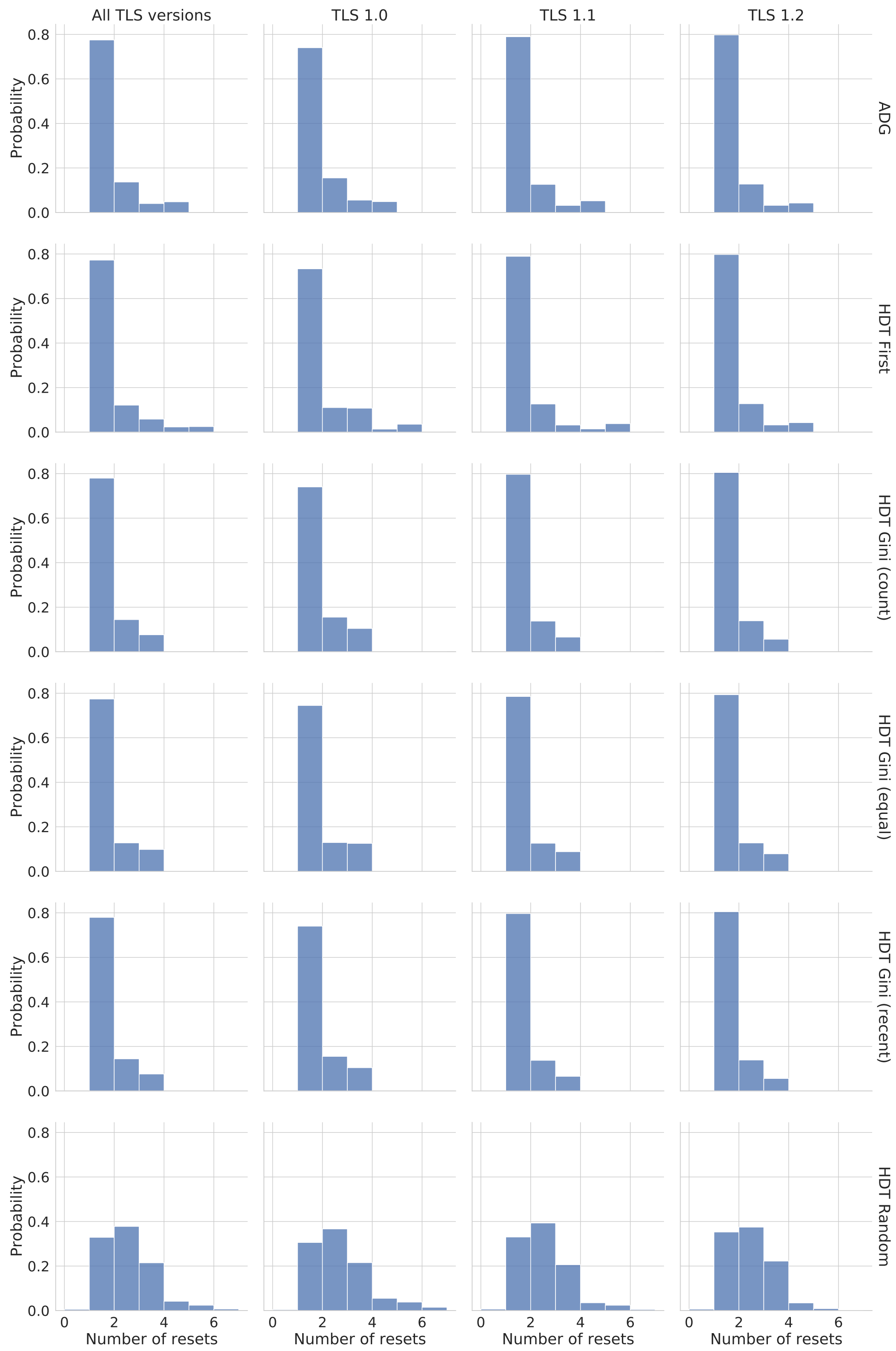


Figure D.8: Benchmark results: Number of inputs with weight function ‘recent’

D.3.3 Computation time

Table D.33: Benchmark summary: Time in seconds with weight function ‘recent’ for all TLS versions

Method	mean	std	min	25%	50%	75%	max
ADG	0.0208	0.0036	0.0155	0.0186	0.0193	0.0229	0.088
HDT First	0.0691	0.0263	0.032	0.0656	0.0676	0.0763	0.2644
HDT Gini (count)	0.0984	0.0316	0.0481	0.0923	0.0955	0.1055	0.3045
HDT Gini (equal)	0.098	0.0306	0.0486	0.0927	0.0957	0.1066	0.249
HDT Gini (recent)	0.1807	0.0525	0.0984	0.1588	0.1652	0.2019	0.4352
HDT Random	0.0981	0.0438	0.0308	0.0675	0.1145	0.1308	0.419

Table D.34: Benchmark summary: Time in seconds with weight function ‘recent’ for TLS 1.0

Method	mean	std	min	25%	50%	75%	max
ADG	0.0236	0.0033	0.0155	0.0224	0.0229	0.0235	0.088
HDT First	0.0796	0.0342	0.0387	0.0746	0.0759	0.0831	0.2644
HDT Gini (count)	0.1093	0.0347	0.0587	0.1025	0.1052	0.138	0.3045
HDT Gini (equal)	0.1082	0.0321	0.0589	0.1038	0.1059	0.1228	0.2374
HDT Gini (recent)	0.2126	0.0535	0.1401	0.1957	0.2012	0.2625	0.4352
HDT Random	0.1117	0.0507	0.0385	0.076	0.1314	0.1385	0.419

Table D.35: Benchmark summary: Time in seconds with weight function ‘recent’ for TLS 1.1

Method	mean	std	min	25%	50%	75%	max
ADG	0.0197	0.0027	0.0162	0.0187	0.0191	0.0193	0.029
HDT First	0.0637	0.0189	0.0321	0.0651	0.0672	0.0682	0.198
HDT Gini (count)	0.0934	0.0283	0.0494	0.0934	0.0954	0.0961	0.2607
HDT Gini (equal)	0.0931	0.0291	0.0486	0.0927	0.0951	0.0963	0.249
HDT Gini (recent)	0.1673	0.0449	0.1054	0.1625	0.1652	0.1664	0.3643
HDT Random	0.0918	0.0381	0.0317	0.0666	0.1143	0.1204	0.2857

Table D.36: Benchmark summary: Time in seconds with weight function ‘recent’ for TLS 1.2

Method	mean	std	min	25%	50%	75%	max
ADG	0.0189	0.0025	0.0156	0.0182	0.0186	0.0189	0.0279
HDT First	0.0632	0.0182	0.032	0.0659	0.0672	0.0678	0.1553
HDT Gini (count)	0.0917	0.0276	0.0481	0.0915	0.0946	0.0955	0.2583
HDT Gini (equal)	0.0918	0.0272	0.0489	0.092	0.0953	0.0959	0.209
HDT Gini (recent)	0.1597	0.0407	0.0984	0.1554	0.1598	0.1612	0.3349
HDT Random	0.0897	0.0371	0.0308	0.0661	0.0933	0.1203	0.2453

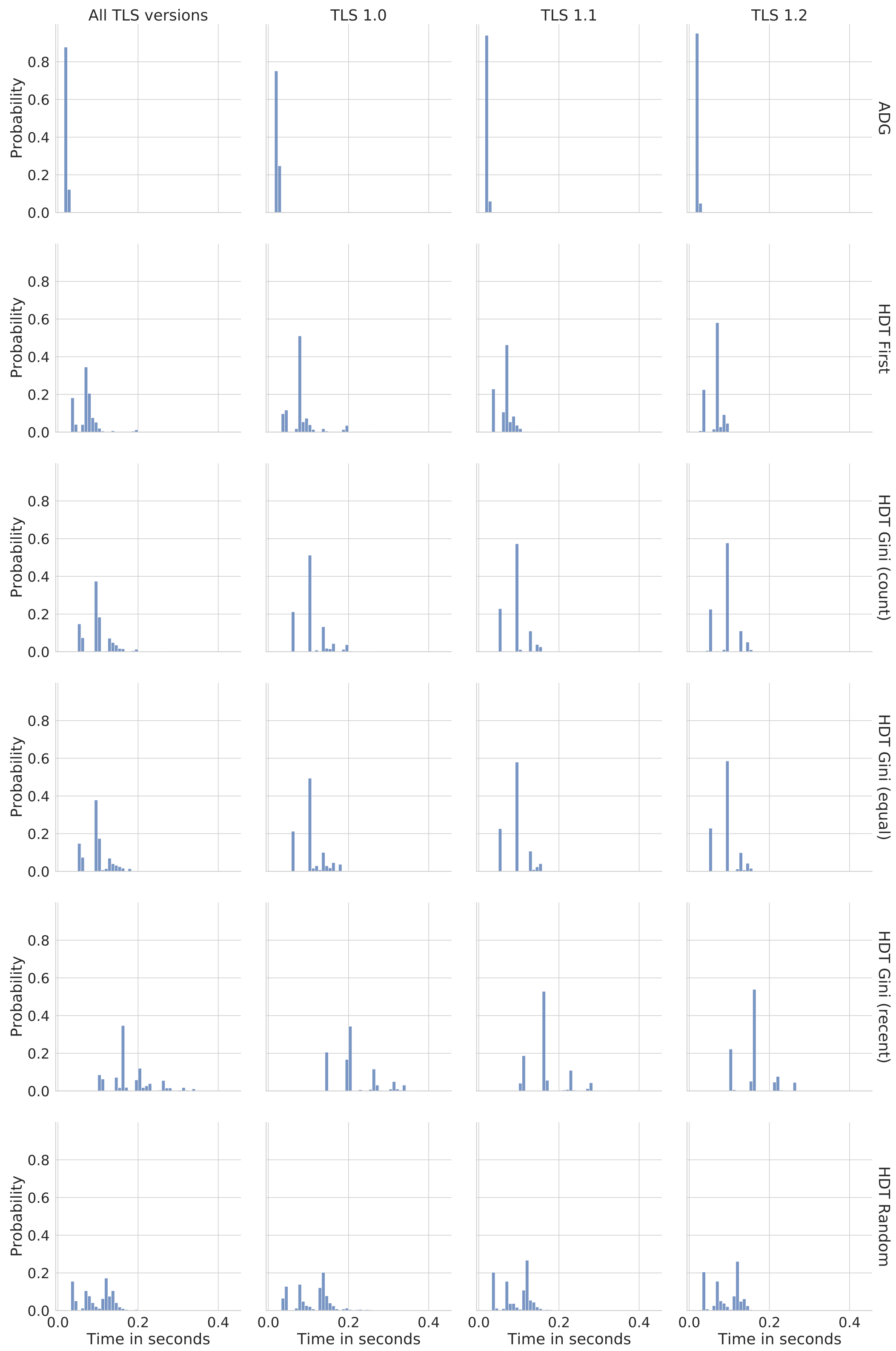


Figure D.9: Benchmark results: Computation time with weight function ‘recent’