Fingerprinting TLS Implementations with model learning

Benedict Blaschko
Department of Informatics
Technical University Munich
Munich, Germany
ge93kux@mytum.de

***Abstract*—Today over 70% of the HTTP traffic uses the Transport Layer Security (TLS) protocol to ensure a safe and protected connection [1]. However, this poses major challenges for monitoring and analyzing connections. We present tools that can look further down the TLS Stack and differentiate between different sub versions and implementations. By combining methods from model learning and fingerprinting we are able to identify small deviations in corner case scenarios to create fingerprints for communication partners. With the L* algorithm [2] we are able to construct a finite automaton to represent paths that depict the comprehensive behavior of the corresponding instance of the TLS implementation. With over 400 different implementations and different versions tested [3], this is an extensive look into a new way of fingerprinting.**

too long as abs

Keywords— Fingerprinting, model learning, L* algorithm

I. Introduction

There is an enormous variety of over 400 possible TLS stacks. For a specific fraction of them, there appears to be slight differences in the actual behavior. This results in an incredible opportunity to differentiate the underlying implementations. TLS is the most used protocol for secure messaging on the internet, however encrypted traffic is hard to analyze. In many cases, examine traffic is very important for network administration or protection against Denial of Service attacks and malicious bot networks. Fingerprinting can be an active tool set for researchers and administrators, which does not use deep packet inspection and does not hurt privacy. Unlike the research on fingerprinting on the used parameters of the connection [5], this work deals with identifying properties on a deeper level on the TLS stack, namely the used version of implementation of TLS. The huge variety of vendors and open source solutions for TLS enables differences in the interpretation of the TLS standard. For instance OpenSSL, NSS, Matrixssl etc. The intuitive thought that the standard forces a uniform behavior in all areas, is not true in all cases. We are able to find specific scenarios where implementations react differently to a specifically crafted input. This is especially present in error handling and unexpected, abnormal behavior like skipping messages, invalid messages and so on. This approach combines findings from previous fingerprinting research and the concept of learning from an unknown set to create the canonical finite automaton. Therefore, on is able to present clear and compact images of the behavior of a specific implementation.

really though?

Use cases and practical implications like identifying vulnerable implementations or the propagation of new versions are described in the discussion.

why not here ? this is the motivating intro !

II. Background

A. Background on the Transport Layer Security

The Transport Layer Security Protocol can be divided into an underlying TLS Record Protocol and four Protocols on top, namely the TLS Handshake Protocol, TLS Change Cipher Spec Protocol, TLS Alert Protocol and the TLS Application Data Protocol.The TLS Record Protocol is used for symmetric encryption using the Advanced Encryption Standard , or in early versions the obsolete DES, 3DES. And securing the messages integrity and authenticity with a message authentication code usually with a hash-based message authentication code. With the TLS Handshake Protocol, the two communication partners agree on a set of cryptographic algorithms in the later encrypted and integrity protected exchange. This set of algorithms is called a cipher suit. Because there are multiple cipher suites available, the client and server have to first agree on which they want to use. In addition, the TLS handshake provides mechanisms for authentication of communication partners between them based on asymmetric encryption and the Public Key Infrastructure. PKI, is a structure to bind public keys to entities and validate these binding with a digital signature.

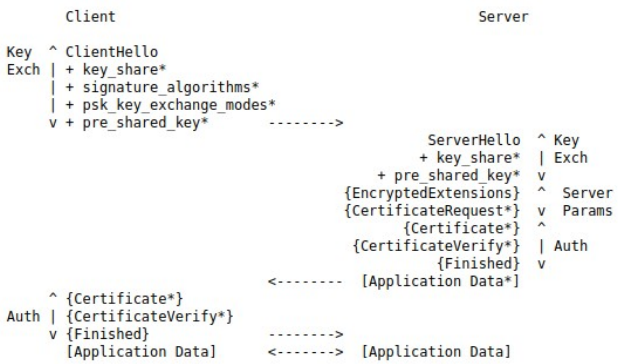

Fig. 1 The TLS1.3 Handshake
*(*mgs optinal, {} msg encrypted)*

The client can provide a keyshare in respect to every proposed ciphersuite or rely on a previously used pre-shared key. The server then respons with the his chosen ciphersuit and the respective keyshare. At this point the server is already able to use the established secure channel.

Fig.1 https://www.rfc-editor.org/rfc/rfc8446#section-4

He can transmite an additional authentication request via the 'CertificateRequest' to demand a certificate from the client and share his own certificate. The handshake is finished when the client provides his requested certificated and confirms all the data from the server.

The TLS Alert Protocol is for communicating errors. The difference in warning and fatal states that a fatal alert always terminates the connection. It covers protocol failures like failed decryption, problems with a certificate etc. Important for the fingerprinting in this paper are:

Handshake failure Alert, indicates that the sender was unable to negotiate the security parameters.

Decoding error Alert states that a message could not be decoded correctly, because fields were out of length or the message was incorrect somehow.

Unexpected message Alert shows that a message did not occure in the correct order.

Close Notify Alert alarms the peer that the sender will not send any other messages on this channel. Any other package send can be ignored.

Bad record mac is received, when there are problems occuring with the authentication verification.

what point?

The TLS Change Cipher Spec Protocol states that from this point on the agreed upon cipher suite is now used for the following messages. The TLS Application Data Protocol transports the actual application data. The message gets divided into blocks, compressed and encrypted depending on the state of the session.

A server can send also a NewSessionTicket message after the client confirm the connection with the Finished messges. The message creates a connection between the ticket value and a secret pre-shared key derived from a master secret.

what does this mean?

B. *Background on model learning with the L* algorithm*

1) *Overview*

This approach of TLS Fingerprinting uses the LSTAR [quote] algorithm. The LSTAR algorithm is a system identification algorithm developed to infer the canonical DFA of any back box system. The algorithm was created by Dana Angluin, a professor emeritus of computer science at Yale University. [quote] padding and todo?
The algorithm utilizes the Teacher-Learner-Approach, in which the TEACHER has full knowledge of the target system and the LEARNER is able to send queries about the system. The algorithm models the system as a Regular Set, which contains words that are also accepted by the system as input. Therefore the LEARNER can ask whether an input sequence is a word part of the Language. In addition, the LEARNER constructs a hypothesis Deterministic Finite Automaton DFA, sends it to the TEACHER and receives confirmation or a counterexample. counterexample how ?

2) *Observation Table*

figure would be helpful

A so-called Observation table monitors the internal state of the algorithm. It is represented by a 2-dimensional table. The rows are denoted as 'S' and refer to the start sequences ? of a word. Since different start sequences can lead to different states in a DFAs, the rows are used to create the different states of the DFA. However, two different start sequences can end up in the same state, therefore the columns add varying ending sequences to the table and are called 'E'. The cell in which a start sequence and an ending sequence of a word meet states whether the concatenation of start and end sequence is part of the langues (denoted with 1) or not part of the language (denoted with 0). As a matter of fact, the columns can now also be viewed as differentiable sequences for potential equivalent DFA states. If two rows, and therefore two starting sequences end up being part of the language with diverse ending sequences, they cannot represent the same state of the DFA. On the other hand, if two rows have the exact same entries for every ending sequence, then the LEARNER can assume that they refer to the same state of the DFA.

3) *Algorithm*

The algorithm starts with the Empty word lambda for start end ending sequence. With each iteration all possible combinations of words in 'S' are concatenated with all Letters of the Input Alphabet, so all allowed inputs to the system. All new created words, if not yet in 'S' are added as additional rows beneath 'S' into the section 'S.A'. Afterwards, the Observation table is challenged for two properties: The Observation table can be now seen as a table with rows which have different patterns of zeros and 1, corresponding the start sequence in combination with the end sequence is part of the language. These patterns are able to differentiate the rows. Closeness is the first property and states that no patterns should be present in the extended rows 'S.A' that are not also in 'S'. Otherwise, the row is added to 'S'. Consistency on the other hand, assumes that there are two rows which have the same ending patterns, and now forces for the corresponding starting words, to also have the same ending patterns when an arbitrary input letter is added to both of they, so that they have a consistent inherent logic. If this is not the case, the algorithm takes the found letter in which the two starting sequences have produced different outputs, and adds it as a column in 'E'.

v hard to understand

4) *Creating the Hypothesis*

When the Observation Table is closed and consistent, the LEARNER constructs the DFA, by taking all rows with varying patterns and interprets it as states, the pattern with the next letter in 'S.A' represents the state where this transition is leading. Closeness now ensures that this pattern is also represented by another state in 'S', so no new state must be created. This DFA is now vetted by the TEACHER.

the next letter of what?

how?
no info about teacher

When the TEACHER confirms it, the algorithm terminates. Nonetheless, when the TEACHER produced a counterexample, this counterexample is added to ‘S’ and also all its prefixes. And the algorithm reiterates.

Initialize S and E to $\{\lambda\}$.
Ask membership queries for λ and each $a \in A$.
Construct the initial observation table (S, E, T).

Repeat:
  While (S, E, T) is not closed or not consistent:
    If (S, E, T) is not consistent,
      then find s_1 and s_2 in S, $a \in A$, and $e \in E$ such that
      $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$,
      add $a \cdot e$ to E,
      and extend T to $(S \cup S \cdot A) \cdot E$ using membership queries
    If (S, E, T) is not closed,
      then find $s_1 \in S$ and $a \in A$ such that
      $row(s_1 \cdot a)$ is different from $row(s)$ for all $s \in S$,
      add $s_1 \cdot a$ to S,
      and extend T to $(S \cup S \cdot A) \cdot E$ using membership queries

  Once (S, E, T) is closed and consistent, let $M = M(S, E, T)$.
  Make the conjecture M.
  If the Teacher replies with a counter-example t, then
    add t and all its prefixes to S
    and extend T to $(S \cup S \cdot A) \cdot E$ using membership queries.
Until the Teacher replies *yes* to the conjecture M.
Halt and output M.

just copy frm paper, not considered

Fig. 2. L algorithm [2]*

B. Background on the Implementation of the Inference Tool

The later examined inference tool consistence of the pylstar python implementation of the algorithm and a simulated server stack. Both programs are isolated in docker containers and talk to each other over a network connection.

The pylstar implementation is also using smart datastructures e.g. a QueryTree implementation. This safes time in the actually inference faces. Many words and querys in the original algorithm are sent multiply times. In theory this is not relevant. However, this can further boost the performance of the implementation.

The pylstar module gets an input alphabet with abstract word like ‘EmptyCertificate’ and treats it like a letter. It then crafts potential words and follows the LSTAR algorithm. Since the TLS implementation is not able to understand this abstract artifacts there is an additional mapper in place to translate the outgoing messages into real TLS messages and the incoming server responses back into abstract words.

absolutely not explained how the inference tool

III. Related Work

This paper builds on the work of Rasoamanana, Levillain and Debar [3]. They used a framework in python, combining the python-based module scapy, with an python implementation of the L* algorithm called pylstar. Scapy is a python module that is able to forge and decode packages. They used cryptographic material from OpenSSL and showed different behavior for different TLS implementations. While looking into implementations like OpenSSL, GnTLS, mbedtls, wolfssl, matrixssl, NSS erlang and fizz, the team found vulnerabilities in the form of unexpected loops, authentication bypasses and padding oracles. Jansen \cite{b3} implemented similar tools for java with additional focus on the fast identification. They looked into the heuristic decision tree and the adaptive distinguishing graph. However they only focus on versions prior to TLS 1.3, which makes it less attractive for work with the newest implementations. Husak, Cermak, Jirssik and Celeda [4] looked into the more traditional view of fingerprinting, by analyzing the usage of different cipher suites in the non-encrypted handshake. They mapped the used ciphers to User-Agent, containing OS and browser versions. They found out that a sophisticated mapping is possible and were able to make strong predictions on the User-Agents.

IV. Analyzing the Constructed Fingerprints

We builded the stacks for TLS implementations:

Openssl (3-0-1, 1-1-1n), *NSS* (3-41, 3-39), *Matrixssl* (4-0-0, 4-3-0). They all used the same Inputalphabet:

[No-Renegotionation, CertificateValid, ClientHello, EmptyCertificate, ApplicationData, CertificateVerify, ChangeCipherSpec, CloseNotify, Finished, InvalidCertificateVerify]

All of them except Matrixssl-4.3.0 yielded the same DFA structure. Namely a DFA with 4 states having the same transitions.

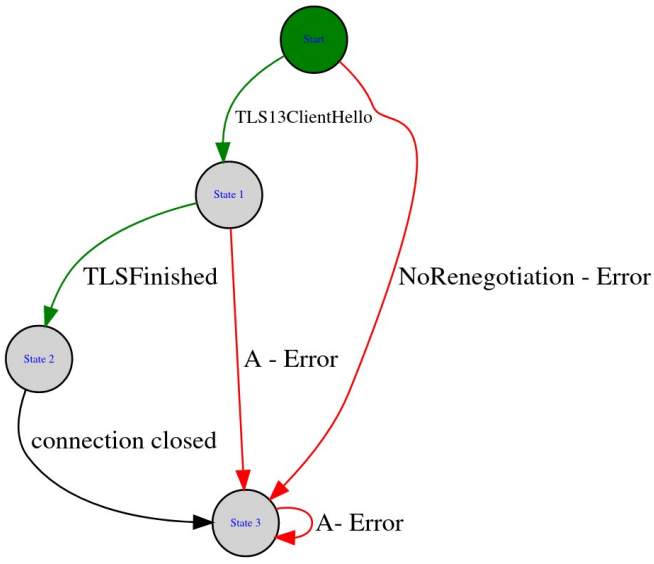

Fig. 3 Infereed DFA with PYLSTAR

As displayed in this general 4-state DFA (combining the inference of 5 inferences) the tool found out, that there is a path to successfully establish the connection and if there is deviating behavior it leads to the drop of the connection. The

whole input alphabet is denoted with ‘A’. In this graph there are only the input messages displayed to give a more general insight.

However, there is more information regarding the inference. When comparing the DFAs, one can discover differences in single error-responses. The inference tool keeps track of all received messages and can list them accordingly. The analysis in the next part is about the received error messages in certain states of the DFA.

When comparing versions of the implementations the differences are smaller than when comparing between different implementations. Despite having all of them implemented TLS1.3, one can detect these differences. However, all the transmitted messages are quite overwhelming, therefore we decieded to select only the responses showing differences in behavior and comparing them directly.

The tables assumes that two different implementations are holding a connection and being currently in the state qi. Then they both sends the same messages m going therefore both into state qj, but receive different replies.

Diverse response messages are marked in red, missing responses are marked with a grey background.

A. Compare versions of the same implementation:

1) Comparing the two Openssl versions

			Openssl 3.0.1	Openssl 1.1.1n
qi	qj	Message	Response	Response
2	3	AppData	AppData+Warning(close_notify)	AppData
2	3	CloseNotify	Warning(close_notify)	EOF

Fig 4. Selected Error responses of OpenSSL versions

qi and qj denote the start and end state this message is sent.

One can see that the earlier versions just sended an EndOfFile error, the newer implementation responses with a close-notification warning.

.

2. Comparing the two NSS versions:

			Nss 3.41	Nss 3.39
qi	qj	Message	Response	Response
1	3	Client Hello	FatalAlert(unexpected_msg)	FatalAlert(bad_record_mac)
2	3	Client Hello	FatalAlert(unexpected_msg)	FatalAlert(bad_record_mac)
2	3	CloseNotify	Warning(close_notify)	EOF

Fig 5. Error responses of NSS versions

For a failed Client hello inserted at the wrong position in the protocol, NSS changed the error code the way it responded. And the same behavior with EOF and close notify like with the OpenSSL versions above.

The two matrix ssl versions are no longer comparable, because they appear completely different with different states.

B. Compare versions of the different implementation:

1) Comparing Openssl 3.0.1 and NSS 3.41

qi	qj	Message	Openssl 3.0.1	Nss 3.41
0	1	Client Hello	ServerHello, ChangeCipherSpec,	ServerHello, […]
1	3	ChangeCipher Spec	FatalAlert(unexpected_msg)	FatalAlert(decode_error)
1	2	Finished	newticket, newticket	No Response
2	3	No-Renegotionation	FatalAlert(handshake_failure)	Warning(close_notify)
2	3	ChangeCipher Spec	FatalAlert(unexpected_msg)	FatalAlert(decode_error)

Fig 6. Selected Error responses of OpenSSL and NSS

In the first client hello response, the openssl server response with an extra changecipherspec. One can see, that the differences are greater when comparing different implementations. One can also note that OpenSSL uses completely different error codes then NSS for example the after the NoRenegation.

2) Comparing Openssl 3.0.1 and Matrix 4.0.0 (version with 4 states)

			Openssl 3 0 1	Matrixss 4 0 0
qi	qj	Message	Response	Response
0	1	Client Hello	ServerHello, ChangeCipherSpec,	ServerHello, […]
1	3	Client Hello	FatalAlert(unexpected_msg)	FatalAlert(bad_record_mac)
1	3	AppData	FatalAlert(unexpected_msg)	Warning(close_notify)
1	3	ChangeCipher Spec	FatalAlert(unexpected_msg)	No Response
1	2	Finished	newticket, newticket	newticket
2	3	No-Renegotionation	FatalAlert(handshake_failure)	EOF
2	3	Client Hello	FatalAlert(unexpected_msg)	FatalAlert(bad_record_mac)
2	2	AppDataEmpty	No Response	FatalAlert(unexpected_msg)
2	3	ChaCiphSpec	FatalAlert(unexpected_msg)	No Response

Fig 7. Selected Error responses of OpenSSL and MatrixSSL

The additional changechipherspecs occures once more, and one noticies that the implementaions of OpensSSL usually responses with ‘FatalAlert-unexpected messages’, and the MatrixSSL implementation with either dropping the connection or a ‘bad record mac’ Fatal Alert.

C. MatrixSSL 4.3.0

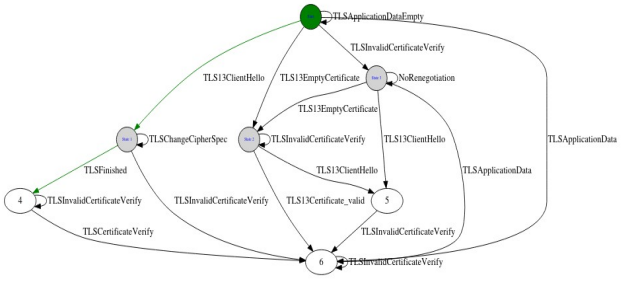

Fig 8. Infereed DFA of MatrixSSL 4.3.0

The implementation MatrixSSL 4.3.0 resulted in an larger DFA. In this case it is even simpler to find a different identifiying sequence and we are not forced to find only error messages that charakterize this implementaion, but we can just differentiaete it after how many number of messages it will drop the connection.

D. Evaluation of the Inference Process

The developers of the inference tools provided some additional data about the conducted operations. Namely the time spend during the execution and querys and actaully send queries.

		time spend [sec]	states	queries	sub_queries
Openssl	1.1.1n	43.7371833324432	4	120544	60025
	3.0.1	49.9824593067169	4	120544	60025
Matrixssl	4.0.0	91.9034194946289	4	120544	60025
	4.3.0	12647.0953195095	7	208753	103948
NSS	3.39	50.9587998390198	4	120544	60025
	3.41	53.0763297080994	4	120544	60025

Fig 9. Table of Meta-Data about the inference procedure

We noticed that the inference takes quite long. So the longest inference time was over 3 hours. There is also a dependence on the number of inferred states and duration. 4 state-dfas are around a minute, the 7 state-dfa took around 3.5 hours. In addition, there is a difference between queries and actual submitted queries. That is the case because the underlying LSTAR implementation uses a smart data structure that first checks whether the word has already been query and only if that is not the case and actually query gets sent. We also saw that all the 4-state dfas have the same number of states, queries and submitted queries, but slightly different times. The reason for this could be dependent on the machine, on docker, or on the implementations.

E. Evaluation of the Fingerprinting

The fingerprints have a huge implication for the security of the TLS stacks. An attacker is now able to pin down a server with a vulnerable implementation. On the other hand, it is also valuable for administrators and security researchers to get an insight into the underlying implementation. Patches and updates can be narrowed down to target servers and the updated versions can be easily verified by the servers behavior. This is all possible with a lightweight implementation, namely just sending the specific sequences of inputs on which the servers are responding differently. This can be further enhanced, when researchers are computing these sequences beforehand and making it and the tools needed to craft these specific packages available to the public. So any administrator could fingerprint all the servers in his network and check for their TLS implementations or any target server. This is especially useful if a Bot net is using a specific vulnerability in an TLS implementation. [quote]. Researchers have a tool to fastly identify potential Command-and-Control (C2) servers.

F. Problems

The biggest pitfall of the fingerprinting is the huge number of queries the LEARNER has to send to achieve the full image of the DFA. This can take up a large amount of time and was also a problem in our research. There were also some unresolved error-codes about an ‘Unknown TLS version’ that could not be resolved. The complete examination of the source code might also be a field of future work.

V. Fingerprinting Usage of Ciphersuite

A. Motivation

The presented methods of fingerprinting yielded some interesting results and we were able to differentiate underlying TLS stacks. With the following approach we want to find a way to expand the parameters to achieve higher precision in the generated fingerprints. Thereby expanding the input alphabet and the possible pathways the dfa can have. This will further increase the probability of finding bugs and security vulnerabilities and sharpen the fingerprinting technology. At the end, it will produce more states and more possibilities for arbitrary behavior of the implementations. We focused on the usage of ciphersuite and thereby on the symmetric ciphers. This can be a pathway for further development of fingerprinting.

B. System environment

what is the appr??

We are not able to integrate this new approach into the software environment of [quote]. Our experiment only comes down to local testing for doability and has to be the starting point for future work. Thereby it also needs to be further generalized, automated and isolated from any host

system to have more validity. We concentrated on the newest openssl version 3.0.5 and connected to a simple openssl server, making use of the openssl library. With the help of generated self-signed certificate we were able to simulate an handshake on a low level and could configure low level settings accordingly. The client set up is based on the python ssl module and the python socket module.

C.) Background on the handshake

As mentioned in the Background, the TLS1.3 handshake consistent of multiple messages.

```
struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* remaining bytes in message */
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:    NewSessionTicket;
        case key_update:            KeyUpdate;
    };
} Handshake;
```

Fig 10. Datastructures used in the Handshake [6]

We are now interested in the first ClientHello message, which has a specific structure:

```
Structure of this message:

    uint16 ProtocolVersion;
    opaque Random[32];

    uint8 CipherSuite[2];    /* Cryptographic suite selector */

    struct {
        ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
        Random random;
        opaque legacy_session_id<0..32>;
        CipherSuite cipher_suites<2..2^16-2>;
        opaque legacy_compression_methods<1..2^8-1>;
        Extension extensions<8..2^16-1>;
    } ClientHello;
```

Fig 11. Datastructures used in the Client Hello Message [6]

In the experiment we wanted to change the parameter ciphersuite cipher_suites.

C. Cipher Suites in TLS1.3

In TLS1.3 the list of symmetric encryption algorithms has been shorted. All remaining are authenticated encryption with associated data algorithms. the most important being: [rfc]

background!

1) AES GCM

Advanced Encryption Standard with the operation modus: Galois Counter Mode counter mode combined with finite field arithmetic. The encrypted text then contains the IV, ciphertext, and authentication tag.

2) AES CCC

Advanced Encryption Standrad with operation modus: Counter mode with Cipher Block Chain Message Authenticated Code:

combines counter mode for confidentiality and cipher block chaining for authentication.

3) ChaCha20 - Poly1305

ChaCha20 is a stream cipher build on a pseudorandom function and based on add-roate XOR operations. This is combined with the message authentication code poly1305

D. Results

With only the ssl python module we were not able to change the parameters of the cipher suit list. There is a method (SET_CIPHERS) for changing the ciphers, however it is not as easily applicable to TLS1.3, because the standard demands the presents of the above listed AEAD algorithms. So the server will always automatically chose the AES_GCM algorithm. And no different behavior can be enforced. The configuration possibilities were too highlevel so we examined the received messages for the standard handshake at the simulated server-side, and could extract the binarys of the message. By debugging and mapping the entries of the hexdump, one could easily understand the structure of the messages and modify it.

```
{0x13, 0x02} TLS_AES_256_GCM_SHA384
{0x13, 0x03} TLS_CHACHA20_POLY1305_SHA256
{0x13, 0x01} TLS_AES_128_GCM_SHA256
{0xC0, 0x2C} TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
{0xC0, 0x30} TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
```

Fig 12. Hexdump of the cipher suit lists

The forced exclusion of the above listed standard algorithms lead to the termination of the connection, however we were able to change the priority of these standard algorithms to convince the server to encrypt our messages with ChaCha20-Poly1305-sha256.

E. Further research

With additional research one is able to automate this procedure and apply this to various tls stacks like the paper [...] did and apply the LSTAR algorithm. One could include several forced mistakes, message drops or even mathematical errors into the connections and thereby enriching the input alphabet. With addition of further cryptographic and message parameters and error handlings ~~one surely~~ is able to differentiate the stacks even further and can increase the benefits of more precise fingerprinting.

F. Evaluation

The results showed that it is also achievable to do some small configuration changes that could be further utilized to fingerprint a TLS stack. However, the set up must be isolated into a docker or a VM like the reachers showed before to avoid interactions with other applications. It is also important to keep an eye on the number of states. We have seen in the other evaluation that the number of states strongly increase the queries and therefore the performance of the method. One can argue that for the actual fingerprinting, only a sequence needs to be tested and not the inferred dfa. However, if the runtime is increased at the shown rate, it is doubtful that the algorithm will be used by other researchers.

VI. Discussion

A. Situation of our work

This paper follows the work of Rasoamana, Levillain and Debar on Fingerprinting the TLS Stack by presenting the core concepts of the Lstar algorithm, examining an extract of the discovered fingerprints and detecting new possible ways to broaden the fingerprinting approach.

These other opportunities for further and more extensive fingerprinting could be also combined with the findings of Martin, Milan, Tomás and Pavel. [4]. They researched user-identification via the set of cipher suites.

Furthermore new possibilities would also be the combination of identification tools presented by Jansen [2]. For example a more heuristic approach for identification of DFAs could be also applied to TLS1.3.

B. Practical Implications

what is this?

1) Fast identifying

The availability of a tool that quickly fingerprint TLS implementations of a server or client by just one TCP connection and one identifying sequence of TLS protocol elements

2) Storing fingerprints

Proposition of a different way of storing tls implementations in the form of finite automaton and simple storage of the images of the DFA, instead of the whole source code.

3) Finite automata algorithms

The applicability of a range of algorithms from the field of theoretical informatics. Namely

finding counterexamples, finding identifying sequences, comparing two implementations in the form of a finite automaton. All of them provable properties of run time and accuracy.

4) Proposing formal definitions for standardization

Future standards can take fingerprints and the results of fingerprinting to develop new and firm definitions for important standards to avoid deviations that lead to vulnerabilities

5) Documentations of usage

Examine the distribution of TLS implementations and sub-versions more in depth.

Study the propagation of new versions by exploring received fingerprints and assess how fast they are accepted in the communities.

6) Finding vulnerable version

Server that are still using versions with known vulnerabilities can be identified much faster, and targets for botnets and compromising computing power is eliminated at an much early state

7) Finding patterns in encrypted traffic

Being able to spot abnormal behavior in a network can add an additional perspective for administrators without breaking encryption and user privacy.

Changes of fingerprints or certain fingerprints that can be associated with malicious traffic [cloudflare resaerch md5 hash and so on]

There might also be an application in the fight of certain botnets and Command and Control servers which possibly have a unique version or fingerprint they are using.

8) Further material for general fingerprinting

Adding new possibilities for fingerprinting hosts and servers in addition to previously developed fingerprinting methods to broaden the picture for the identification of endpoints

C. Limitations

1) Overlapping fingerprints

This is the largest limitation for fingerprints. In often cases there are no distinguishable differences in behavior of the acting server or client, even on the level of implementation, this happens often in strongly connected versions, like follow ups where the developers made sure the behavior is in fact not changing

2) Open connections

To achieve results with the presented approach, there has to be a way to communicate directly with the server, and this is not yet applicable to a traffic based methodology where there is just access to encrypted packages

A way would be to limit the fingerprinting to the unencrypted Client Hello, like in the work of Martin, Milan, Tomás and Pavel [5]. If on is able to change the content of the package and actively taking over the handshake, one might also get inside in the implemented versions

3) Unifying behavior

There might be counter measures to avoid the possibility of fingerprinting by identifying the deviations and losing the gap in the behaviors of the implementation to avoid creating fingerprints of implementations and being able to hide as much information from the outside world. However, there are so many implementation and this ongoing process of developing is in most cases yielding different results in the behavior.

4) Areas of future study

There are various areas for further study like fingerprinting only on client hello and server hello messages,

fingerprinting implementations on botnets if there are occurring patterns,

fingerprinting on the record protocol and how this is implemented and used.

form??

Furthermore, interesting ares might also be fingerprinting on error messages and proposing algorithms for standardizing with formal definitions to avoid bugs and to declare a uniform behavior for all versions.

VII. Future Study

There are wide fields of future study for TLS fingerprinting. The first step would be to further expand the input alphabet like described above. It is also possible to inclue wrong values, timestamps, purposely mathematically miscalculation and examine how the different implementations react to it. Especially in cases that are prone to errors,there might be a variety of error-handling installed. Furthermore, there is also the opportunity to fingerprint the client TLS stacks to find out more about the used implementations on the client side.

Another important field to further examine is the fast recognition of identifying sequences. This is the core to fingerprinting, because a fast recognition algorithm results in performant toolings that can be used in combination with precalculated fingerprints. In this situation the user of this technology does not have to worry about speed of the actual inference. This can be computed beforehand and is not time critical.

Finally, TLS fingerprinting could be a strong call for more formalism in standardizations. The undefined behavior the fingerprinting is based on will always be a problematic. However, a DFA present in a standardization might be the path to a clearer and more unambiguous error handling and unique error encoding.

VIII. Conclusion

This paper showed core concept of the construction of finite automata in order to fingerprint implementations. We gave an insight into a new form of extracting information of an underlying TLS stack by making use of model learning, namely the LSTAR algorithm. We analyzed the previously discovered fingerprints and examined their implications. These fingerprints reveal differences in the error handling of major TLS stack implementations. In this paper we compared the produced prints graphically and showed varying error messages in the Handshake protocols. This method can help researchers to gain more information about the underlying TLS functionality without compromising the server. We also exhibit new ways to create more thorough and precise fingerprints. We therefore examine the different usage of symmetric cipher algorithms and the doability of this approach. Limitations to the process are mainly overlapping fingerprints, the initiative for unifying the behavior of differences in tls implementations.

References

v few sources?

[1] https://etherealmind.com/percentage-of-https-tls-encrypted-traffic-on-the-internet/ Accessed 27.12.2022

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

[3] Rasoamanana, A.,Levillain O., Debar H.: Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks (2022).

[4] Janssen, E., Vaandrager, F., de Ruiter, J., Poll, E.: Fingerprinting TLS implementations using model learning. Master's thesis. Radboud University (2021)

[5] Martin, H.; Milan, Č.; Tomáš, J.; Pavel, Č. HTTPS Traffic Analysis and Client Identification Using Passive SSL/TLS Fingerprinting. EURASIP J. Inf. Secur. 2016, 1, 1–22

[6] https://www.rfc-editor.org/rfc/rfc8446#section-4