DissecTLS: A Scalable Active Scanner for TLS Server Configurations, Capabilities, and TLS Fingerprinting

Markus Sosnowski(✉), Johannes Zirngibl, Patrick Sattler, and Georg Carle

Technical University of Munich, Munich, Germany
{sosnowski,zirngibl,sattler,carle}@net.in.tum.de

Abstract. Collecting metadata from Transport Layer Security (TLS) servers on a large scale allows to draw conclusions about their capabilities and configuration. This provides not only insights into the Internet but it enables use cases like detecting malicious Command and Control (C&C) servers. However, active scanners can only observe and interpret the behavior of TLS servers, the underlying configuration and implementation causing the behavior remains hidden. Existing approaches struggle between resource intensive scans that can reconstruct this data and light-weight fingerprinting approaches that aim to differentiate servers without making any assumptions about their inner working. With this work we propose DissecTLS, an active TLS scanner that is both light-weight enough to be used for Internet measurements and able to reconstruct the configuration and capabilities of the TLS stack. This was achieved by modeling the parameters of the TLS stack and derive an active scan that dynamically creates scanning probes based on the model and the previous responses from the server. We provide a comparison of five active TLS scanning and fingerprinting approaches in a local testbed and on toplist targets. We conducted a measurement study over nine weeks to fingerprint C&C servers and analyzed popular and deprecated TLS parameter usage. Similar to related work, the fingerprinting achieved a maximum precision of 99 % for a conservative detection threshold of 100 %; and at the same time, we improved the recall by a factor of 2.8.

Keywords: Active scanning · TLS · Fingerprinting · C&C servers

1 Introduction

Transport Layer Security (TLS) is currently the *de facto* standard for encrypted communication on the Internet [18]; thus, providing a good common base to analyze, compare, and relate servers. The protocol is influenced by libraries, hardware capabilities, custom configurations, and the application build on top, resulting in an a server specific TLS configuration. A large amount of metadata from this configuration can be collected because in the initial TLS handshake clients and servers must exchange their capabilities such that a mutual

© The Author(s) 2023
A. Brunstrom et al. (Eds.): PAM 2023, LNCS 13882, pp. 110–126, 2023.
https://doi.org/10.1007/978-3-031-28486-1_6

cryptographic base can be found. There are at least two possibilities to collect this metadata: on the one hand, TLS server debugging tools like testssl.sh [33] or SSLyze [10] perform resource intensive scans that dynamically adapt to the server and can reconstruct a human-readable representation. On the other hand, active TLS fingerprinting approaches like JARM [4] or Active TLS Stack Fingerprinting (ATSF) [31] use a small set of fixed requests that are designed to be good in differentiating TLS server configurations. Their light-weight approaches enable them to be used for Internet-wide scans; *e.g.*, `censys.io` already provides JARM fingerprints [8].

Related works have shown that collecting and analyzing TLS configurations from a large amount of servers enables further use cases, *e.g.*, monitoring a fleet of application servers [4] or detecting malicious Command and Control (C&C) servers [4,31]. To be able to collect this data, the respective scanning approach needs to be efficient, to both reduce the time it takes to collect the data and the impact the scan has on third parties.

However, using a fixed set of probes will always leave open the possibility for redundant data to be collected and for useful information to be overlooked; therefore, the performance of subsequent applications (*e.g.*, detecting C&C servers) might not reach their full potential. An alternative is to exhaustively scan a server until the full TLS configuration can be reconstructed. However, current tools are not efficient enough to be used on a large scale.

This work investigates whether a dynamically adapting scan can be implemented efficient enough to be used on a large scale and if this provides a benefit over existing work and tools. We propose DissecTLS as an efficient tool to collect TLS server configurations and provide the following contributions:

(i) a model of the TLS stack on a server that explains its behavior towards different requests and that can be used to craft TLS Client Hellos (CHs) on a per-server level to reconstruct its underlying configuration;
(ii) a comparison of five popular TLS scanners regarding their capabilities to detect different configurations and their scanning costs performed both in a controlled testbed environment and on toplist servers;
(iii) a measurement study of one top- and two blocklists over nine weeks comparing a C&C server detection using fingerprinting tools and this work, complimented with an overview of common TLS parameters; and
(iv) published measurement data [29], scanner, and comparison scripts [30].

2 Methodology

During the initial handshake of the TLS protocol, clients and servers share several pieces of information related to their capabilities to negotiate a mutual encryption base. Part of this can be configured by the user (*e.g.*, ciphers the server is allowed to select), only limited by the actual capabilities of the software and hardware. However, TLS servers only react to clients; therefore, reveal only a portion of their internal configuration with every response (*e.g.*, the server

Table 1. Model of TLS configuration properties on a server and their representations.

Property	Representation
Supported TLS versions	set
Cipher suites	priority list / set[a]
Supported groups	priority list / set[a]
ALPNs	priority list / set[a]
Selected extension values[b]	map(id → value)
Inappropriate Fallback support	bool
Order of TLS extensions	DAG[c] / set
Error behavior	one of {TCP, TLS, Ignore}

[a] Only sets can be collected if the server uses the client preferences.
[b] EC Point Formats, Signature Algorithms (Cert), and Heartbeat.
[c] A directed acyclic graph (DAG) is used if the server responds consistently.

selects only a single cipher from the list of proposed ciphers). This means, multiple requests (i.e., CHs) must be sent to collect the full amount of information hidden in the TLS stack. It is not feasible (regarding time and resources) to send every possible CH to a server. Thus, every active TLS scanner uses a strategy to select CHs depending on the information it wants to collect. With DissecTLS we aim to reconstruct the configuration that cause the observed TLS behavior in a scalable manner that can be used even for Internet-wide scans. Therefore, we need to reduce the number of requests as far as possible. This is achieved by defining a general model of the TLS configuration on a server and use the minimum number of requests necessary to learn the parameters of the model. Additionally, we defined the output such that it can be used for fingerprinting; *i.e.*, exclude session, timing, and instance related data. Depending on the previous responses from a TLS server we use the model to craft the most promising CH that should reveal new information about the server.

The following sections will explain our model of the TLS stack, how we represent its features, and how our scanner is implemented on an abstract level.

2.1 Modeling the TLS Configuration on Servers

To design a scan that is able to extract the parameters of a TLS stack configuration, these parameters need to be defined first. We analyzed popular web server configurations (*e.g.*, provided by Mozilla [23]), TLS server debugging tools (testssl.sh [33] and SSLyze [10]), passively captured TLS handshakes, and the TLS 1.2 and 1.3 specification [27,28] to derive the model from Table 1. This model reflects our understanding of TLS and how it is applied in the Internet. It is not complete as discussed in Sect. 6.

TLS servers support a set of versions and either answer with the correct version, abort the handshake, or attempt a downgrade to a lower version. There are three priority lists used in the handshake where the client offers a list of

options and the server selects one according to its internal preferences. Iteratively removing each parameter from new requests that was previously selected by the server, the full list of length n can be scanned with $n + 1$ requests. This is the optimal approach using the "lowest number of connections necessary [...] for one host", explained by Mayer *et al.* [20]. However, if the server prefers client preferences, only a set of supported parameters can be acquired instead of a priority list. Clients can inform the server about their own priorities through the order of parameters in the CH. We tested whether servers respect this priority as follows: after learning at least two parameters, we also learned which one the server selected first. Then, we send a new CH where the order of the two is reversed; we know a server prefers its own preferences if this had no influence on the selection. We scan cipher suites, supported groups, and Application Layer Protocol Negotiations (ALPNs) with the currently 350, 64, and 27 possible values listed by IANA [17], respectively. Some servers provide the full list of supported groups [27] directly as extension, in these cases we do not explicitly scan them. However, the presence of a pre-computed key share can influence the priorities of the supported groups; hence, we collect the preference without a pre-computed key share and afterwards test whether the presence influenced the decision. Support of most TLS extensions is indicated by their presence or absence and does not need a particular logic, they just need to be triggered in the CH with their presence. Others need specific logic because they modify the encryption (Encrypt Then Mac and Extended Master Secret), are mutually exclusive with other extensions (Record Size Limit and Max Fragment Size), or multiple values can be send (Heartbeat). Sometimes, the content of extensions is of interest because it reveals information about the server capabilities, and in these cases we store the raw byte content. The man-in-the-middle inappropriate fallback protection needs special logic because it only makes sense to send the signaling cipher [21] if multiple TLS versions are detected. Lastly, servers can respond differently in cases of problems, some report an error on the Transmission Control Protocol (TCP) layer, some send TLS alerts, and others just ignore the problematic part of the handshake (*e.g.*, using a default value). An example is shown in Appendix A.

In summary, this model is an abstract and human-readable representation of the TLS stack on a server that can explain its behavior in TLS handshakes.

2.2 Representing Multiple Observations of Extension Orders

The order of extensions is not defined in the TLS standard; however, we argue most servers have a consistent order as result how they are implemented in the code. We confirmed this by checking the source code of the Golang TLS library we modified to implement our tool. Moreover, in Sect. 4.2 we found that more than 99% of the servers in the study responded with a consistent order.

The presence of extensions depends on the request and not all extensions can be observed at the same time (*e.g.*, the key share extension is only present in a TLS 1.3 handshake). This means, every response from the server reveals part of the order and multiple observations can be combined to reconstruct the

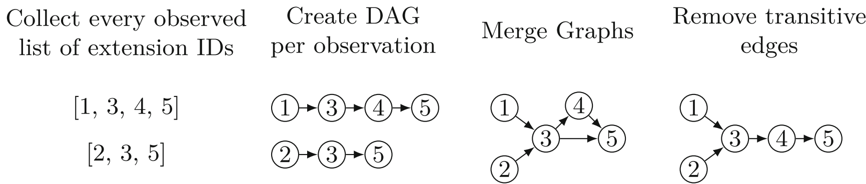


Fig. 1. Example for merging multiple observations of TLS extensions into a single format. If the graph contains cycles after merging, the extension order is inconsistent.

internal order on the server as close as possible. We created a DAG for each observation, merged these graphs and removed duplicate and transitive edges. If the graph contains cycles after merging; this means, the observations were inconsistent and the extension order cannot be reconstructed. An example for this process is illustrated in Fig. 1.

In conclusion, a DAG allows to represent multiple observations of extensions in a compact format that is as close as possible to the internal server order.

2.3 Implementation of DissecTLS

DissecTLS is implemented as feature of the TUM goscanner [15], which is a TLS scanner for Internet-wide measurements. It is based upon a modified version of the Golang TLS library that can send custom CHs and extract handshake data.

We designed DissecTLS to use as few requests as possible. To achieve this, the logic of the scan is divided into several scan tasks and each task is responsible for one or a few related parameters. The tasks are designed to collect information in parallel. Every task modifies the next CH depending on its current state and, after receiving the response, updates the server parameters with the new information. This is repeated until an error occurs; then, each task that could have caused the error is toggled on or off until one remains and the cause is found (*e.g.*, whether a missing cipher or the wrong TLS version was responsible). The task causing the error must resolve it (*e.g.*, mark the cipher scan as complete or a TLS version as unsupported) and the scan is continued normally. In general, the more specific a server was (*e.g.*, it sent a "protocol version" TLS alert instead of a TCP reset), the faster the cause can be identified. Some servers did not respond with error messages but let the TCP connection time out, we treat these cases not as an error in favor of reducing the load in case of real timeouts.

In summary, with help of our model we implemented a scanner that uses a minimal amount of requests that allow us to reconstruct the TLS configuration.

3 Comparison of TLS Scanners and Their Ability to Detect Different TLS Stack Configurations on Servers

Active TLS scanners are designed to extract information from servers. We compared their performance doing so by measuring their ability to distinguish differ-

Table 2. Detected number of Nginx configurations for each test case. An ideal scanner detects every alteration made on the server and finds "Goal" number of configurations.

Test Case	DissecTLS	DissecTLS (lim.)	ATSF	JARM	SSLyze	testssl.sh	Goal
TLS Versions	15	15	13	11	15	14	15
Cipher Suites	1 956	359	115	11	63	1 956	1 956
ALPNs	2	2	2	2	1	2	2
Preferences	2	2	2	2	1	2	2
Session Tickets	2	2	2	2	2	2	2
Used CHs per Server							
Minimum	8.0	8.0	9.0	9.0	423.0	9.0	
Average	14.3	10.0	10.0	10.0	450.1	132.7	
Maximum	42.0	15.0	12.0	10.0	455.0	224.0	

ent TLS server configurations. Without analyzing the scanner output we argue that whenever a scanner is able to differentiate two different TLS configurations, the scanner has detected the relevant piece of information. The more configurations it can differentiate, the more valuable is its output. However, we also measured the costs of the scanner by counting the amount of requests it needed to perform the scan. The lower the costs are, the more servers can be scanned in the same time and the lower the impact is on individual servers. An ideal scalable approach collects a high amount of information with low costs.

We compared testssl.sh [33], SSLyze [10], JARM [4], ATSF [31], and this work. We selected them because from our knowledge they are the relevant representatives that either fingerprint or reconstruct the TLS configuration. We configured our approach in two versions, one tries to fully reconstruct the TLS configuration (DissecTLS), the other completes using 10 handshakes (DissecTLS lim.). We interpret the textual output of each scanner as its representation of the server. If two outputs are equal, they detected no difference in the configuration. We were able to directly use the output of JARM, ATSF, and this work. We had to remove information regarding timing (*e.g.*, scan time), sessions (*e.g.*, cryptographic keys), and server instances (*e.g.*, the domain name) from the output of testssl.sh and SSLyze to get stable results for the same TLS configuration. Additionally, we disabled the vulnerability detection of these tools.

3.1 Scanner Comparison in a Local Testbed

In our local testbed we compared the TLS scanners based on a ground truth. We challenged them in different scenarios where we systematically made alterations to a server and checked whether the scanners were able to detect it.

The experiment was designed as follows: we selected a parameter we could configure on the TLS server (Test Case), launched an Nginx 1.23 docker container for each configuration we could generated for this parameter, and scanned the containers with every scanner. We used tcpdump [32] to measure the number of CHs the scanners were using. The results can be seen in Table 2. An ideal

scanner is able to differentiate all variations we have configured (listed under "Goal"). Nginx allowed to configure four TLS versions, resulting in 15 working combinations ($2^4 - 1$). We only used six TLS 1.2 ciphers because this number was still scannable in a reasonable time. These six ciphers resulted in 1 956 configurations (every permutation of every combination of the six ciphers). ALPNs, Server Preferences, and Session Tickets were scanned either en- or disabled. The table shows that only DissecTLS and testssl.sh were able to detect every alteration we made on the server. DissecTLS (lim.) tried to detect the TLS versions, then data in extensions, and lastly the ciphers; hence, it usually detected only the first few ciphers from the server and could not detect configurations that differ in the lower cipher priorities. Testssl.sh could not detect one case where only TLS 1.3 was enabled because at the time of the experiment it included an OpenSSL version that was not TLS 1.3 capable. SSlyze was not able to detect the order of ciphers, therefore, could not detect any permutation we performed on the ciphers. The two fingerprinting approaches ATSF and JARM were not able to detect every alteration on the servers. This was expected as they use a fixed number of requests. However, as this experiment is artificial, it is possible that the obtained fingerprints are still good enough for fingerprinting use cases. Regarding the scanning costs, the picture is reversed. The fingerprinting tools and the limited version of DissecTLS used the least number of requests, DissecTLS slightly more, and testssl.sh and SSlyze being the most costly. We expect testssl.sh and SSLyze to be used on a small scale where scanning costs do not matter; however, we can see that the former is more optimized and uses fewer requests to collect more information. We can see a difference in JARM and ATSF regarding the maximum number of used CHs: both initially use 10 CHs, but the latter completes handshakes; therefore, we sometimes observe an additional CH from the scanner as response to a Hello Retry Request (servers can send them to request a different key share from the client). DissecTLS makes use of this TLS feature to reconstruct the supported group preferences of the server in case no key share is present because its presence might influence the decision. Therefore, we observe up to 15 CHs for the maximum of 10 handshakes.

To conclude, DissecTLS competes both with testssl.sh regarding the amount of collected information and with active TLS fingerprinting tools regarding their low scanning costs. However, this analysis only includes a single TLS implementation and artificial test cases; therefore, to get a more complete view the next section compares the scanners in a more realistic setting on toplist servers.

3.2 Scanner Comparison on the Top 10k Toplist Domains

This section compares the five TLS scanners on the top 10k domains from the Tranco [19] toplist. Because the ground truth is unknown, only their performance to differentiate servers can be compared. The scan took 6 days to complete because of the low request rate testssl.sh and SSLyze were able to achieve.

The number of configurations each tool was able to detect and the number of requests necessary to collect this information can be seen in Table 3. DissecTLS was able to detect the most configurations, followed by Testssl.sh. However, this

Table 3. Comparison of TLS scanners regarding the number of detected configurations and Client Hello usage on the resolved (IPv4 and IPv6) top 10k Tranco domains.

	DissecTLS	DissecTLS (lim.)	ATSF	JARM	SSLyze	testssl.sh
Configurations	3 450.0	1 839.0	1 956.0	1 325.0	2 738.0	3 235.0
Total CHs	530.6k	235.6k	238.5k	209.4k	9.5M	3.4M
Average CHs	24.0	10.7	10.8	9.5	430.0	154.9
Total Scanned Targets						22 075

Table 4. Overview of the collected data for the Top- and Blocklist study.

Input source	Total Samples	Distinct Targets	Unique Domains	Successful Scans DissecTLS	ATSF	JARM
Tranco Toplist	15.8M	2.8M	1.1M	14.4M	13.1M	14.3M
abuse.ch Feodo Tracker	4 040.0	812.0		1 725.0	2 126.0	768.0
abuse.ch SSLBL	1 034.0	223.0		27.0	42.0	26.0

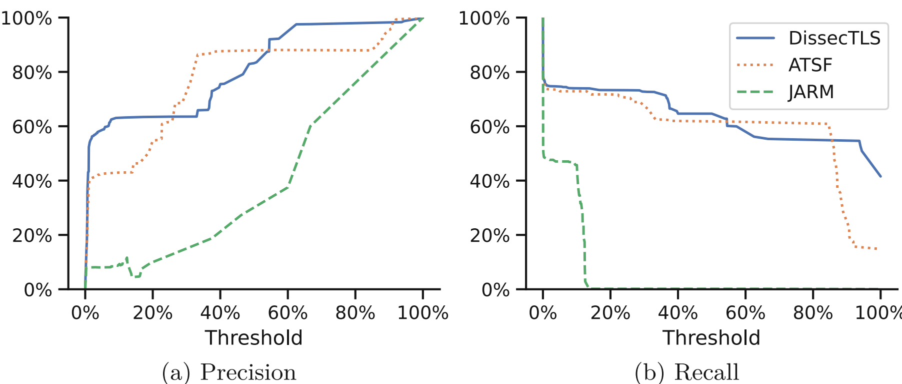
does not mean a scanner collected only a super-set from another, as discussed in Sect. 6. This work uses just a sixth of the requests compared to testssl.sh, with 24 CHs on average. JARM used less than 10 requests on average because sometimes the TCP connection failed and no CH was sent. In contrast to the last section, the limited version of DissecTLS performed a bit worse than ATSF. Apparently, our approach only detects the finer details that help to differentiate TLS configurations when it completes the scan.

This sections showed that the dynamic scanning approach from testssl.sh, DissecTLS, and SSLyze is superior to the fixed selection of CH regarding collected data. However, this comes with increased scanning costs. We argue that only JARM, ATSF, and DissecTLS are resource efficient enough to be used for large-scale measurements. Additionally, in the following we refrain from limiting the number of requests of DissecTLS. While roughly doubling the scanning costs it provides a more complete; hence, a more useful view on the TLS stack.

4 Measurement Study on Top- and Blocklist Servers

This section transfers the findings from the previous section to a larger scale where we collected more than 15 Million data samples with each scanner (data available under Ref. [29]). However, we only used DissecTLS, ATSF, and JARM for this study because testssl.sh and SSLyze did not scale well enough for this use case.

We scanned servers from the complete Tranco [19] toplist and two C&C server blocklists: the abuse.ch Feodo Tracker [1] and the abuse.ch SSLBL [2]. We collected nine weekly snapshots starting from July 01, 2022. Table 4 presents an overview of these measurements. We resolved domains from the toplist and



(a) Precision (b) Recall

Fig. 2. Precision and Recall for classifying C&C servers each week based on the data collected in previous weeks. Using the fingerprints from the respective scanner as input.

scanned each combination of IPv4 and IPv6 address together with the domain as Server Name Indication. We call each IP and domain name combination a target. The two blocklists only list IP addresses; hence, the number of targets is equal to the number of entries on these lists. We count a "success" if the respective scanner produced an output. For DissecTLS and ATSF this is the case if a TCP connection could be established. JARM additionally needed the server to respond at least once with a Server Hello. DissecTLS and JARM implement a retry mechanism on failed TCP handshakes, together with the different success definition, this can explain the variations in the success rates.

4.1 Fingerprinting C&C Servers

Althouse *et al.* [4] and Sosnowski *et al.* [31] describe the fingerprinting of malicious C&C servers as one of the major use cases of their respective approach. Like the fingerprinting tools, DissecTLS collects data from the TLS stack and its output can be used for fingerprinting. In the following, we performed a C&C server classification based on the data collected with JARM, ATSF, and DissecTLS from the weekly top- and blocklist measurements. This allowed us to compare the different data collection approaches regarding a C&C server detection.

Each scanned target is labeled as C&C server (positive) or toplist server (negative) based on whether the IP address was listed on one of the blocklists in the respective week. In case of ambiguity, the blocklist took preference. Then, we made a prediction for each target based on the data and labels from previous weeks. The prediction worked as follows: each week n, we calculated the rate how often a fingerprint was observed from C&C servers versus toplist servers during weeks $[1..n-1]$ and predicted a "C&C server" if this rate was above a configurable threshold. A threshold of 50% means that at least every second server with a specific fingerprint would need to be labeled as C&C server so that this fingerprint results in a "C&C server" prediction. We performed this classification with the fingerprints from JARM, ATSF, and DissecTLS. Figure 2 compares the precision and recall defined as $\frac{TP}{TP+FP}$ and $\frac{TP}{TP+FN}$ ($TP :=$ "true positives", $FP :=$

"false positives", $FN :=$ "false negatives"), respectively. Intuitively, precision is the rate of correct classifications and recall the fraction of C&C servers we were able to identify. It shows that the classification based on ATSF or DissecTLS performed quite similar with a high precision that reached the maximum at the most conservative detection threshold of 100%. However, the high precision is only achieved through a lower recall of 15% and 42%, respectively. JARM fingerprints were not descriptive enough to identify the C&C servers on our two blocklists and together with the lower success rate this resulted in a low recall.

In conclusion; DissecTLS, ATSF, and JARM collect TLS data that can be used to detect C&C servers. DissecTLS and ATSF achieved a precision that is more than 99% for the 100% threshold. Moreover, DissecTLS achieved a 2.8 times higher recall than ATSF for said threshold. Additionally, we argue that DissecTLS collects more valuable data because it provides a human-readable representation of the TLS stack as described in the next section.

4.2 Human-Readable TLS Server Configurations

Until this section this work analyzed TLS configurations as a single unit; however, DissecTLS produces an output (see example in Appendix A) that can be used to understand how a server is configured. This can help to explain why fingerprinting was possible. In Appendix B we present statistics from the top- and blocklist servers that can deepen our understanding of TLS parameter usages on the Internet. We have analyzed the support for different TLS versions; computed a popularity ranking of cipher suites, supported groups, and ALPNs; analyzed whether servers prefer client preferences or not; and looked how many servers supported deprecated cipher categories.

In conclusion, an exhaustive TLS scanning approach can be used for fingerprinting but additionally provides valuable insights into the TLS ecosystem.

5 Related Work

Fingerprinting TLS clients in passive network traces is a well established discipline, shown by multiple related works [3,5–7,16]. This concept has been adapted by Althouse *et al.* [4] and Sosnowski *et al.* [31] through active scanning to be able to fingerprint servers. Both approaches use a fixed set of 10 requests that "have been specially crafted to pull out unique responses in TLS servers" [4] and "empirically optimized to provide as much information as possible" [31], respectively. They capture variations of the TLS configuration in their fingerprints; however, they do not actively search for them; additionally, the explainability of their output, or fingerprint, is low and it is difficult to understand what has caused the specific fingerprint. Both works show that they can find malicious C&C servers on the Internet. A fundamentally different approach is proposed by Rasoamanana *et al.* [26], they define a State Machine describing TLS handshakes and argue that the transitions between states can be used to fingerprint specific

implementations; especially, if these transitions are not conform to the TLS specification and, sometimes, even pose a security vulnerability. Their focus on the behavior of the library in the context of erroneous input does not consider the parameters that are the cause of the non-erroneous behavior. Dynamically scanning TLS servers is a common practice in the context of analyzing and debugging servers with tools like testssl.sh [33] or SSLyze [10]. Both make assumptions how the TLS on the server works and adapt their scanning to this model. However, they focus on the configurable part of the server, do not export every fingerprintable information, and are not optimized for Internet-wide usage (*e.g.*, use more than 100 requests to scan a single server). Mayer *et al.* [20] showed that cipher suite scanning can be optimized to use 6% of the connections compared to related works. However, they ignore the rest of the TLS configuration.

6 Discussion

This work proposes an exhaustive but optimized TLS scanning approach that can be used for large-scale Internet measurements and for TLS fingerprinting. The following paragraphs discuss several aspects we found worth mentioning.

C&C TLS Configurations. In general, configurations we could relate to C&C servers had just slight alterations in their parameters compared to common configurations (*e.g.*, the position of a single cipher). However, we collected interesting results (see Appendix A) from several servers labeled as Trickbot, according to the Feodo Tracker [1]. These servers supported TLS 1.0 and downgraded higher versions, which is already a rare behavior. In contrast to the low TLS version, the ciphers were strong and some used a modern key agreement, *i.a.*, X25519 (standardized 2016 [14] - 8 years after TLS 1.2 [28]). This led us to the conclusion that this was a modern server where some modern features were disabled.

Completeness of the Testbed. Every TLS scanner from Sect. 3.1 was capable to detect more configurations than the ones we have tested, *e.g.*, TLS versions prior to TLS 1.0 or other cipher suites. We selected the tested values because they were configurable on the Nginx server. Some features, *e.g.*, the extension order, cannot be configured. Our choice of the six ciphers was arbitrary and it is possible that there are combinations of ciphers where the performance of the scanners is different. However, our tool sends 350 different ciphers and the analysis shows that it can effectively identify permutations of those on the server.

Completeness of the TLS Server Model. Sections. 3.2 and 3.1 showed that DissecTLS and testssl.sh were able to detect the most TLS configurations. However, looking into their output, no scanner provided a super-set of the other; hence, our proposed model cannot be complete. We manually investigated cases where testssl.sh was able to differentiate configurations while DissecTLS was not, and vice versa. Both scanners rely on consistent server responses; however, Sosnowski *et al.* [31] reported inconsistent behaviors for 1% of their fingerprinted targets. If servers behave inconsistently, both scanners might have collected an incomplete view of the TLS stack and reported different configurations on each

connection attempt. We expect testssl.sh is a bit more resilient to this behavior due to its excessive but thoroughly scanning in contrast to limiting the amount of CHs as possible. DissecTLS was able to find more configurations than testssl.sh; *i.a.*, through differentiating the error behavior and by merging the observed extensions into a single DAG. However, testssl.sh detected more details sometimes: *e.g.*, it was able to detect variations for non-elliptic TLS 1.2 Diffie-Helman Key Exchange Mechanism (KEM) key sizes, collected cipher priorities per TLS version, detected typical server failures like being unable to handle certain CH sizes, differentiated whether a session resumption was implemented through IDs (legacy) or tickets, and used a service detection (*e.g.*, detecting Hypertext Transfer Protocol). To support these cases with DissecTLS, we would need to increase the number of sent CHs and implement the missing TLS features in the library. Whether the additional data would provide a benefit for use cases like the C&C detection is an open question for future work because we could not include testssl.sh in our C&C server detection study (Sect. 4.1), due to its limited scalability. To conclude, neither scanner collected a super-set from the other and we argue that it is impossible to build an ideal scanner without knowledge about every TLS implementation and how TLS will evolve in the future.

Ethical Considerations All our active Internet measurements are set up following best scanning practices as described by Durumeric *et al.* [12]. We used rate limiting (overall and per-target), dedicated scan servers with abuse contacts, informative reverse DNS entries and websites that inform about our research, maintained a blocklist, and provided contact information for further details or scan exclusion. Our work does not harm individuals or reveal private data as covered by [11,24] and focuses on publicly reachable services. The core design principle of our approach was to reduce the impact on third parties by minimizing the number of requests while maintaining an useful level of data quality.

7 Conclusion

This work proposes a scalable active scanning approach to reconstruct the TLS configuration on servers. The approach is compared with four active TLS scanners and fingerprinting tools. While we are able to collect a comparable amount of information to single server TLS debugging tools, we also keep up with the performance of scalable active TLS fingerprinting tools using around twice the number of requests. Our approach collects more data than the fingerprinting tools and produces human-readable representations of a TLS configuration, improving the explainability of the approach. We performed a nine week measurement study of top- and blocklists, analyzed common TLS parameter usages, and fingerprinted potentially malicious C&C servers. Similar to related work, the fingerprinting achieved a precision of more than 99% for the most conservative detection threshold of 100%; however, at the same time DissecTLS achieved a recall 2.8 times higher than the related ATSF [31]. This was achieved by a scan that dynamically adapts based on a TLS stack model and previously learned information. The model was used to explain server responses and to craft new

requests that should reveal new data. This paper shows that an exhaustive TLS parameter scanner can be implemented efficiently enough to be used on a large scale. Moreover, it can replace existing active TLS fingerprinting approaches because it provides a similar fingerprinting performance but additionally produces a valuable dataset. In the future, it can help to acquire a global view on the TLS parameter usage to deepen our understanding of the TLS ecosystem.

A Example DissecTLS Output

Table 5 shows an example output we have collected from a TLS server that was labeled as Trickbot on the Feodo Tracker [1]. For better readability the output was enhanced with the parameter names from IANA [17]. We could determine the order of the ciphers and supported groups; therefore, the shown values are a priority list. We could not determine the ALPN preferences from the server because it supported only one option. The extension order was consistent across all observations and the resulting DAG had no branches.

Table 5. Example DissecTLS output obtained from a server labeled as Trickbot.

Property	Value
supported TLS versions	TLS 1.0 support
	TLS 1.1 downgrade
	TLS 1.2 downgrade
	TLS 1.3 downgrade
cipher suites (priority list)	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
	TLS_RSA_WITH_AES_256_CBC_SHA
	TLS_RSA_WITH_AES_128_CBC_SHA
	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
cipher Preference	server
supported groups (priority list)	x25519
	secp256r1
group preference	server
with key share	client
ALPN (set)	http/1.1
ALPN preference	unknown
extension data	EC Point Format → [uncompressed
	ansiX962_compressed_prime,
	ansiX962_compressed_char2]
order of TLS extensions (DAG)	renegotiation_info → max_fragment_length →
	ec_point_formats → session_ticket → ALPN →
	encrypt_then_mac → extended_master_secret
version error behavior	Ignore
cipher error behavior	TLS Alert
groups error behavior	Ignore
ALPN arror behavior	Ignore

B Additional TLS Server Parameter Statistics

This section adds several statistics we have obtained while analyzing the data from the measurement in Sect. 4.2. We find them interesting; however, not necessary to support the findings of the paper.

Table 6. Support for different TLS versions from successfully scanned targets.

	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3
Success	62.83%	65.24 %	99.57%	74.57%
Abort	37.61%	35.03%	0.32%	0.00%
Downgrade	0.02%	0.21%	0.15%	25.90%

Support for the TLS versions can be seen in Table 6. Although TLS 1.0 and 1.1 is deprecated since 2021 [22], we saw a high amount servers supporting it. Some servers even downgraded the handshake by responding with a lower version than the one we requested. This was expected for TLS 1.3 because the TLS 1.3 CHs is basically a TLS 1.2 CH with special extensions. A server that does not understand these extensions should continue with a TLS 1.2 handshake. However, we rarely observed this behavior also for other versions.

We collected cipher suites, supported groups, and ALPNs as priority lists. This enables combining them to get the overall most popular values as shown in Table 7 (full list available under [30]). This problem is similar to a voting problem where multiple individuals can vote with a list of descending preference and can be solved with scoring rules as discussed by Fraenkel *et al.* [13]. We decided to use the Dowdall rule, which favors parameters with top preferences. This way parameters of a low priority, usually only kept for backward compliance, are given a low score. The ranking worked as follows: from each priority list the parameters $[p_1, \ldots, p_n]$ are scored with $[1, \frac{1}{2}, \ldots, \frac{1}{n}]$, the scores for each parameter are summed up, and ranks based on the highest scores are computed. We analyze the parameters independent of the TLS version; hence, the TLS 1.3 ciphers are ranked above the others because, in general, higher versions are preferred over ciphers.

Some servers selected the cipher suites, supported groups or ALPNs based on the preference of the client. This leaves security decisions open to the client but can be beneficial to the user if the client has limited hardware capabilities. However, in our measurements we saw this was rarely the case and most servers preferred their own priorities as shown in Table 8. This is different if a client already pre-computed a TLS 1.3 key share for one of the supported groups; then, 29% of the servers used the key share to avoid an additional round trip.

An important security feature on servers is the support against version downgrade attacks. If this is not given, even when security issues are fixed in a newer TLS version, a downgrade can reopen these attack vectors. Such a downgrade could be achieved, *e.g.*, by a man-in-the-middle attacker blocking connections

Table 7. Most preferred TLS parameters (IANA names [17]) ranked separately with the Dowdall rule and total distinct values. Each scanned target was used as vote.

Rank	Cipher Suites	Supported groups	ALPNs
1	aes_256_gcm_sha384	x25519	h2
2	chacha20_poly1305_sha256	secp256r1	http/1.1
3	aes_128_gcm_sha256	secp384r1	http/1.0
4	ecdhe_rsa_with_aes_128_gcm_sha256	secp521r1	spdy/3
5	ecdhe_rsa_with_aes_256_gcm_sha384	x448	spdy/2
6	ecdhe_ecdsa_with_chacha20_poly1305_sha256	brainpoolP512r1	http/0.9
7	ecdhe_ecdsa_with_aes_128_gcm_sha256	brainpoolP384r1	acme
8	ecdhe_rsa_with_aes_256_cbc_sha384	secp256k1	tls/1
9	ecdhe_rsa_with_aes_128_cbc_sha256	brainpoolP256r1	h2c
10	ecdhe_ecdsa_with_aes_128_cbc_sha	sect571r1	h3
Total	152	45	13

Table 8. Server preferences and downgrade protection in relation to number of targets where the parameter could be successfully determined.

	Parameter Determined	Values	
Cipher suite preference	2334678	4.63%	(client)
Supported Group preference	2125081	5.58%	(client)
with Key Share	1661825	28.60%	(client)
ALPN preference	1899399	0.03%	(client)
TLS downgrade protection	2101497	98.68%	(protcted)

for a higher TLS version expecting the client will attempt to reconnect with a lower version. Table 8 shows most servers were protected.

Several servers still support categories of deprecated ciphers [9,25] as shown in Table 9. These ciphers are known to be insecure; however, they are not per-se a security vulnerability because an attacker would still need to force a client and server to agree on them.

Table 9. Servers supporting at least one deprecated cipher suite per category. Percentages are in relation to the successfully scanned targets.

	Null	Export	Anonymous	RC4	Any
Targets	376	1403	1606	36281	36694
	0.02%	0.06%	0.07%	1.56%	1.58%

References

1. abuse.ch: Feodo Tracker. https://feodotracker.abuse.ch/. Accessed 28 Oct 28 (2022)
2. abuse.ch: SSL Certificate Blacklist. https://sslbl.abuse.ch/. Accessed 28 Oct 2022
3. Althouse, J., Atkinson, J., Atkins, J.: TLS Fingerprinting with JA3 and JA3S (2019). https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967
4. Althouse, J., Smart, A., Nunnally Jr., R., Brady, M.: Easily identify malicious servers on the internet with JARM (2020). https://engineering.salesforce.com/easily-identify-malicious-servers-on-the-internet-with-jarm-e095edac525a
5. Anderson, B., McGrew, D.: OS fingerprinting: new techniques and a study of information gain and obfuscation. In: 2017 IEEE Conference on Communications and Network Security (CNS) (2017). https://doi.org/10.1109/CNS.2017.8228647
6. Anderson, B., McGrew, D., Kendler, A.: Classifying Encrypted Traffic With TLS-Aware Telemetry. FloCon (2016)
7. Anderson, B., McGrew, D.A.: Accurate TLS fingerprinting using destination context and knowledge bases. CoRR (2020). https://doi.org/10.48550/arXiv.2009.01939
8. Censys: JARM in Censys Search 2.0 (2022). https://support.censys.io/hc/en-us/articles/4409122252692-JARM-in-Censys-Search-2-0. Accessed 14 Oct 2022
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (2006). https://doi.org/10.17487/RFC4346
10. Diquet, A.: SSLyze. https://github.com/nabla-c0d3/sslyze. Accessed 13 Oct 2022
11. Dittrich, D., Kenneally, E., et al.: The Menlo Report: Ethical principles guiding information and communication technology research. US Department of Homeland Security (2012)
12. Durumeric, Z., Wustrow, E., Halderman, J.A.: ZMap: fast internet-wide scanning and its security applications. In: Proceedings of the USENIX Security Symposium (2013)
13. Fraenkel, J., Grofman, B.: The Borda Count and its real-world alternatives: comparing scoring rules in Nauru and Slovenia. Aust. J. Pol. Sci. (2014). https://doi.org/10.1080/10361146.2014.900530
14. Friedl, S., Popov, A., Langley, A., Emile, S.: Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (2014). https://doi.org/10.17487/RFC7301
15. Gasser, O., Sosnowski, M., Sattler, P., Zirngibl, J.: Goscanner (2022). https://github.com/tumi8/goscanner
16. Husák, M., Cermák, M., Jirsík, T., Celeda, P.: Network-based HTTPS client identification using SSL/TLS fingerprinting. In: 2015 10th International Conference on Availability, Reliability and Security (2015). https://doi.org/10.1109/ARES.2015.35
17. IANA: Transport Layer Security (TLS) Parameters. https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml. Accessed 13 Oct 2022
18. Labovitz, C.: Internet traffic 2009–2019. In: Proceedings of the Asia Pacific Regional Internet Conference on Operational Technologies (2019)
19. Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., Joosen, W.: Tranco: a research-oriented top sites ranking hardened against manipulation. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium (2019). https://doi.org/10.14722/ndss.2019.23386

20. Mayer, W., Schmiedecker, M.: Turning active TLS scanning to eleven. In: De Capitani di Vimercati, S., Martinelli, F. (eds.) SEC 2017. IAICT, vol. 502, pp. 3–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58469-0_1
21. Moeller, B., Langley, A.: TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507 (2015). https://doi.org/10.17487/RFC7507
22. Moriarty, K., Farrell, S.: Deprecating TLS 1.0 and TLS 1.1. RFC 8996 (2021). https://doi.org/10.17487/RFC8996
23. Mozilla: SSL configuration generator (2022). https://ssl-config.mozilla.org. Accessed 13 Oct 2022
24. Partridge, C., Allman, M.: Addressing ethical considerations in network measurement papers. In: Proceedings of the 2015 ACM SIGCOMM Workshop on Ethics in Networked Systems Research. Association for Computing Machinery (2016). https://doi.org/10.1145/2793013.2793014
25. Popov, A.: Prohibiting RC4 Cipher Suite. RFC 7507 (2015). https://doi.org/10.17487/RFC7465
26. Rasoamanana, A.T., Levillain, O., Debar, H.: Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks. In: Computer Security - ESORICS (2022). https://doi.org/10.1007/978-3-031-17143-7_31
27. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (2018). https://doi.org/10.17487/RFC8446
28. Rescorla, E., Dierks, T.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (2008). https://doi.org/10.17487/RFC5246
29. Sosnowski, M., Zirngibl, J., Sattler, P., Carle, G.: DissecTLS Measurement Data. https://doi.org/10.14459/2023mp1695491
30. Sosnowski, M., Zirngibl, J., Sattler, P., Carle, G.: DissecTLS: Additional Material (2023). https://dissectls.github.io/
31. Sosnowski, M., et al.: Active TLS stack fingerprinting: characterizing TLS server deployments at scale. In: Proceedings of the Network Traffic Measurement and Analysis Conference (TMA) (2022)
32. The Tcpdump Group: tcpdump. https://www.tcpdump.org. Accessed 27 Oct 2022
33. Wetter, D.: Testing TLS/SSL encryption. https://testssl.sh/. Accessed 27 Oct 2022

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.