Categorization of TLS Scanners

Youssef Jemal, Tim Betzer*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: youssef.jemal@tum.de, betzer@net.in.tum.de

***Abstract*—The Transport Layer Security (TLS) protocol is a cornerstone of modern web security. In the TLS ecosystem, the configurations of servers are mostly concealed from clients, as servers only react to the clients' proposals during the handshake process. Scanning and fingerprinting approaches address this gap by performing several handshakes with the server in order to extract as much information as possible about its TLS configuration. This information can be helpful in improving the network security by discovering misconfigurations and identifying malicious servers. This paper classifies existing TLS scanners into three categories. We introduce every category and discuss its characteristics while highlighting its advantages and drawbacks. We also introduce a local testbed to compare the performance of various scanners based on their ability to distinguish between different TLS configurations.**

***Index Terms*—Active scanning, Fingerprinting, TLS, SSL**

1. Introduction

Throughout the last years, Transport Layer Security (TLS) has become a widely used security protocol and the standard for encrypted communication over the Internet [1]. It guarantees the integrity and confidentiality of the data as well as the authentication of the involved parties. The TLS protocol begins with a handshake where the client and the server negotiate a common cryptographic base. In the handshake, the client shares all their capabilities with the server. The latter, however, only chooses from the client's proposals according to its internal configuration. As a result, the server's TLS capabilities remain unclear for external parties.
A possible way to uncover this information is by passively listening to the server's TLS communications. Based on the content of the captured data packets, we attempt to reconstruct the server's TLS configuration.This approach does not generate any additional traffic and does not strain either the target server or the network. It is, however, inherently inefficient with protocols that implement encryption mechanisms since a third-party listener has no access to the session's cryptographic keys. For example, in TLS 1.3, the server already encrypts several fields in the *Server Hello* message, which makes them inaccessible for analysis by passive monitoring tools [2]. This is where active scanning tools become relevant. The idea is to craft and transmit several *Client Hello* messages and then observe the server's responses to attempt to reconstruct its TLS configuration. Such scans are beneficial not only because they provide us with a detailed overview of the server's configuration, but also because they are powerful tools for finding vulnerabilities and misconfigurations. Furthermore, scanning approaches can help detect malicious Command & Control (C&C) servers as they usually have the same, or similar, configurations [3].
This paper categorizes existing TLS scanners into three types: Elementary TLS Scanners such as TUM goscanner [4] or zgrab2 [5]. In their basic mode, these scanners only initiate one TLS handshake and return the server's response. They are practical for Internet-wide measurements [5] and can provide us with invaluable insights into the TLS ecosystem. The second type are server debugging tools that are able to reconstruct a detailed and comprehensive representation of the server's internal configuration, such as DissecTLS [6], SSLyze [7], and testssl.sh [8].
The third and last type are fingerprinting approaches. The most relevant representatives are JARM [9] and Active TLS Fingerprinting (ATSF) [10]. These approaches only send a small fixed number of *Client Hello* messages and generate a unique fingerprint for every TLS configuration, which is primarily used to differentiate and compare servers.
The remainder of this paper is organized as follows: We start by explaining the methodology of the TLS protocol in section 2. Section 3 summarizes the most important TLS configuration parameters from a scanner's perspective. Section 4 analyzes different TLS scanning approaches and discusses their advantages and limitations. In section 5, we compare the performance of TLS scanners across different categories in a local environment before we conclude in section 6.

2. Methodology

A basic understanding of the TLS protocol, especially the TLS handshake, is required to understand the scanners' work system. This paper only focuses on TLS 1.2 and TLS 1.3, as all previous versions are deprecated [11]. They are also the most relevant and widely used versions within the TLS ecosystem [6]. TLS 1.2, standardized in 2008 [12], has been the backbone of secure Internet communication for over a decade. Its successor, TLS 1.3, was standardized in 2018 [13] and introduced several significant improvements. The faster handshake process is one of the most important enhancements. TLS 1.3 only takes 1 Round-Trip Time (RTT) to complete the handshake instead of 2 RTTs in TLS 1.2. The newest version is also inherently more secure as it only supports AEAD cipher suites [13] that simultaneously provide privacy and authenticity [14].

 doi: 10.2313/NET-2025-05-1_16

The following figure captures the most important steps of the TLS 1.3 handshake:

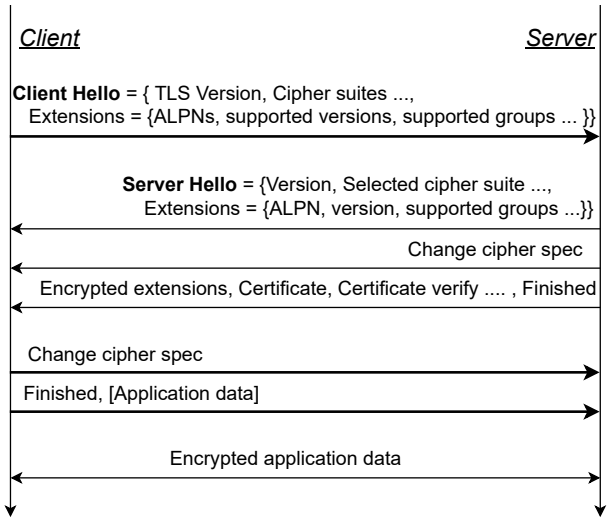

Figure 1: TLS 1.3 Handshake [15]

2.1. Client Hello

The client initiates the handshake by sending a *Client Hello* message to the server [13]. This message specifies the client's supported TLS versions and suggests several cipher suites for the server to choose from. The *Client Hello* message can potentially contain more than ten extensions [13] with the majority of them not relevant for our work. This paper only focuses on the most important extensions from a scanning perspective. As part of the handshake, the client also generates multiple key pairs (public and private) based on the proposed key exchange algorithms (in the *supported groups* extension) and includes the public keys in the *key share* extension of the CH message [13]. These keys are necessary for generating the symmetric encryption key.

2.2. Server Hello

In response to the *Client Hello* message, the server replies with a *Server Hello* message [13]. This message selects a cipher suite from those proposed by the client and specifies the server's extensions. After generating its key pair and receiving the client's public key, the server can calculate the symmetric encryption key. It also includes its public key in the *key share* extension to allow the client to calculate the same cryptographic material. Furthermore, in order to authenticate itself, the server sends its *Certificate* and a *Server Certificate Verify* message to prove ownership of the private key associated with the certificate. To ensure the integrity of the handshake, the server applies a hash function on all the previously exchanged data and sends the result in the *Finished* message (encrypted). The client also performs the same operation and compares the output with server's hash. If both hashes are equal, then it means that the handshake messages were transmitted correctly. To conclude the handshake, the client sends a *Change Cipher Spec* message followed by a *Finished* message. From this point, the client and the server can start exchanging application data in an encrypted and secure manner. In earlier TLS versions, the *Change Cipher Spec* record was used as an indication that all subsequent records will be encrypted [12]. To avoid misbehavior of middleboxes, implementations try to make the TLS 1.3 handshake similar to TLS 1.2 [13]. This includes the transmission of the *Change Cipher Spec* record, which does not serve any function in TLS 1.3.

3. TLS Configuration parameters

It would be impractical and costly for scanners to extract all the server's TLS properties since it would require a lot of time and resources. Depending on the information it wants to gather, each scanner prepares a specific strategy to extract this information [6]. The table below outlines some of the TLS configuration parameters that can be extracted by scanning tools [6].

TABLE 1: The typically extracted TLS configuration properties

Parameters	Representation
Supported Versions	set
Cipher Suites Supported Groups ALPNs	priority list/set[1]
Deflate compression	bool

[1] A set in case the server uses the client preferences.

The Supported Versions field lists all the TLS versions supported by the server. These versions include TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3. Although TLS 1.0 and TLS 1.1 were formally deprecated in RFC 8996 due to security concerns [11], they continue to be accepted by servers within the TLS ecosystem [6]. One of the most important properties of a TLS server is its supported cipher suites. A cipher suite in TLS 1.3 specifies the AEAD (Authenticated Encryption with Associated Data) cipher mode that will be used for bulk encryption and a hash algorithm. This is different from previous TLS versions, where the cipher suite also contained information about the key exchange algorithm. In TLS 1.3, the key exchange algorithm is negotiated through extensions as discussed in the previous section. A server's supported cipher suites are defined in relation to a specific TLS version. For example, TLS 1.3 only defines and supports 5 cipher suites [13], meaning that any server using this version should only accept a subset of these 5 cipher suites [13].

The supported groups parameter specifies the cryptographic groups that the server supports for the key exchange protocol used for the generation of the symmetric encryption key.

The Application-Layer Protocol Negotiation (ALPN) extension optimizes the communication. In the *Client Hello* message, the client provides a list of the supported application protocols, and the server selects at most one protocol based on its capabilities. The server then includes

 doi: 10.2313/NET-2025-05-1_16

it in the ALPN extension of the *Server Hello* message [16]. This extension enables the negotiation of the application layer protocol during the TLS handshake. It, therefore, reduces the overhead that would have otherwise occurred if the negotiation were to be completed after the handshake.
In versions prior to 1.3, the TLS protocol included support for compression methods, with the most used being the *deflate compression* method. Clients and servers would negotiate a commonly supported compression method through the *compression methods* field. However, TLS 1.3 obsoleted this feature entirely due to known security vulnerabilities, such as the CRIME attack [17]. TLS 1.3, however, still implements the *compression methods* field in the handshake messages, with its value always set to 0 (indicating no compression). This is done to maintain backward compatibility with previous versions.

4. Capabilities of TLS Scanners

This section explores the three categories of TLS scanners: Elementary scanners, fingerprinting approaches, and server debugging tools. In the following subsections, we introduce each type and provide examples of different implementation approaches.

4.1. Elementary Scanners

Elementary scanners initiate a single handshake with the server and return the server's response. This approach provides very limited information about the server's configuration, which makes such tools impractical for scanning in their base mode. They are utilized as foundations for more complex implementations, where their elementarity and scalability can be leveraged as part of a deeper analysis. A prominent example is zgrab2, which can perform a TLS handshake over HTTP with the entire IPv4 address space in less than 6 hours and 20 minutes [5]. This tool is highly effective for conducting Internet-wide surveys. *Censys.io* [5] is a search engine powered by zgrab2 that automates Internet-wide scanning. It offers a REST API and a web interface for querying an up-to-date database of the public address space gathered through continuous scanning with zgrab2. The search interface supports advanced search features, including "full-text searches, regular expressions, and numeric range queries" [18]. For example, a command such as `services.tls.versions.tls_version = "TLSv1_3" and location.country_code = "DE"` returns all IPv4 hosts in Germany that support TLS 1.3, which are currently around 3.7M hosts. Durumeric *et al.* [5] argue that this approach democratizes the internet-wide scanning process by making the scanning of the TLS ecosystem accessible to researchers and users in general without concerns about legal permissions or network infrastructure.
Another example is TUM goscanner [4] which serves as a foundation for the implementation of DissecTLS.

4.2. TLS Debugging Tools

Scanners of this type provide a detailed overview of the server's TLS configuration. This is typically achieved by sending a large number of *Client Hello*s to exhaustively test all possible configurations. As there are over 370 ciphers that can be used across different TLS versions, the number of *Client Hello*s required is primarily determined by the *Cipher Suites* parameter [19]. SSLyze [7] is a scanning tool that implements a "naive" algorithm. It performs a full cipher suite scan by conducting one TLS handshake for each cipher suite. This results in approximately 543 transmitted *Client Hello* messages [19]. A main advantage of this stateless approach is that it allows for parallelization of requests, trading the high generated traffic for a speedup in the scan execution [19]. testssl.sh [8], while more efficient and optimized than SSLyze [6], also remains impractical for big-scale measurements due to the high number of sent requests. Such tools should only be used on a small scale, where we're only interested in fast and precise results and where the scanning costs can be neglected. The most prominent debugging tool that combines the lightweight nature of fingerprinting approaches with the precision and completeness of debugging tools is DissecTLS. Integrated as a feature of TUM goscanner [4], DissecTLS divides the scanning process into multiple subtasks, each responsible for collecting information about specific parameters [6]. Each task maintains a state and dynamically crafts the next *Client Hello* based on this state. This approach reduces redundant handshakes that do not yield new information.

4.3. Fingerprinting Approaches

Fingerprinting approaches, unlike the previously discussed categories, do not produce a human-readable representation of the server's configuration. Instead, they aim to minimize the number of sent *Client Hello*s while extracting as much information as possible. The collected data is then stored in a concise format called a "fingerprint", which is primarily used for comparing servers [3]. The reduced traffic footprint and the lightweight output make these methods highly scalable and well-suited for large-scale scanning. The most significant use case for these tools is the identification of Command and Control (C&C) servers, which are a fleet of computers that have the same or similar configurations and are used to transmit malicious commands and steal data. By comparing the similarity between a fingerprint of an unknown server and that of a known C&C server, fingerprinting tools provide a high probability indication on whether the server is a C&C server. Althouse *et al.* [3] argue that their fingerprinting tool JARM can identify "most, if not all Cobalt Strike C2 servers" on the IPv4 address space on port 443.
JARM [9] sends 10 *Client Hello*s specifically designed to extract as much information as possible about the server's TLS configuration. It then uses the received *Server Hello*s to produce a 62-character fingerprint with the following structure [3]:

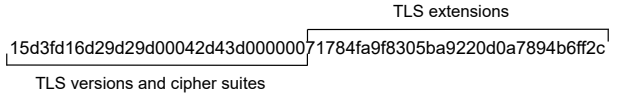

Figure 2: JARM Output Example

Each three characters of the first 30 provide information about the TLS version and cipher suite chosen by the

 doi: 10.2313/NET-2025-05-1_16

server as a response to each of the ten *Client Hello*s. A "000" indicates that the server refused the connection [3]. The remaining 32 bits are constructed using a truncated SHA256 hash on the TLS extensions sent by the server in the *Server Hello*.
An even more advanced fingerprinting tool is Active TLS Fingerprinting (ATSF). Sosnowski *et al.* [10] suggest that this tool is superior to JARM in identifying C2 Servers and generally better at distinguishing server configurations [10]. Similar to JARM, ATSF sends ten distinct *Client Hello* messages. However, it provides a more precise fingerprint by leveraging additional TLS handshake messages. While JARM only deals with the extensions present in the *Server Hello* message (Plus the ALPN extension), ATSF also incorporates extensions from *Encrypted Extensions*, *Certificate Request*, *Hello Retry Request* and *Certificate* TLS messages [10]. ATSF, however, produces a longer output than JARM since it employs a different technique for generating fingerprints. In fact, for every TLS handshake, it produces an independent fingerprint and then concatenates all these outputs to generate the final server's fingerprint [10]. This is an example of an elementary handshake fingerprint:

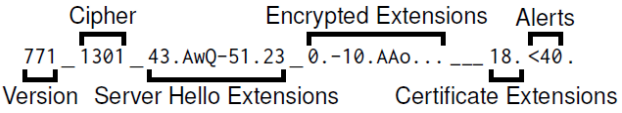

Figure 3: ATSF handshake fingerprint example

5. Comparison of Scanners in a Local Testbed

In our local testbed, we compared the TLS scanners based on two important criteria: their capability to correctly distinguish between different TLS configurations, and the amount of traffic they produce throughout the scanning process. The main test parameter for this experiment is the *Cipher suites* parameter. We selected this parameter because it allows for a wide range of possible configurations, matching our experiment's needs. We selected 5 TLS 1.2 cipher suites, resulting in 325 unique configurations. We can compute this number by calculating the number of all permutations of all the possible combinations of the selected cipher suites. The use of permutation was necessary because server preferences were enabled throughout the experiment (except when scanning the *Preferences* property). This means that the server keeps an internal priority list of the supported cipher suites and always picks the highest-priority cipher suite available. For each configuration, we deployed a Docker container running an Nginx Version 1.27.3 server. We ran each scanner against all 325 servers and counted the unique scan outputs generated. This value reflects the number of configurations successfully identified by the scanner. For ATSF and JARM, we were able to use the output directly. For the other scanners, we used a Python script that extracts the scanned cipher suites from the scan results, which we can then use as a basis for the differentiation. This measure was necessary to avoid unstable results, as some parameters, such as *scan time*, can vary even for the same configuration. We also extracted other simple parameters, such as *Preferences*, and *Session Tickets* which can be either en- or disabled. To assess the scanning costs, We used tcpdump to capture the number of *Client Hello*s sent by the client during each scan, and then computed the average number across all scans.
The following table summarizes the results of the experiment.

TABLE 2: Identified number of Nginx configurations for each scanner.

Test Case	ATSF	JARM	SSLyze	testssl.sh	DissecTLS	Total
Cipher Suites	22	17	31	325	325	325
Preferences	2	2	1	2	2	2
Session Tickets	2	2	2	2	2	2
Average CHs	10	10	447	128.1	11	-

As shown in the table, only dissecTLS and testssl.sh were able to completely identify all the servers' cipher suites configurations. ATSF and JARM, however, were only able to differentiate around 20 unique configurations. This result is expected, as these fingerprinting tools only send a fixed number of *Client Hello*s. Their lower level of precision is offset by significantly reduced scanning costs, making them ideal tools for large-scale deployments. SSLyze had a relatively poor performance, as it was only able to differentiate 31 unique configurations. The reason behind this is that SSLyze does not consider the order of the cipher suites [6], meaning it could only differentiate between the different combinations of the 5 cipher suites, which amount to 31 configurations. On top of that, it was by far the most costly scanner in terms of generated traffic, with an average of 447 *Client Hello*s. While testssl.sh was able to distinguish all configurations, its high average scanning cost of 128 *Client Hello*s makes it unsuitable for large-scale use cases. DissecTLS is the best-performing scanner in this experiment. It effectively combines the low cost typically associated with fingerprinting approaches while maintaining the accuracy and precision of TLS debugging tools. The average of approximately 11 sent *Client Hello*s, along with the successful identification of all the TLS configurations, clearly demonstrates this.

6. Conclusion and Future Work

In this paper, we introduced the concept of TLS scanning. We presented three different scanning categories: Elementary scanners, fingerprinting techniques, and debugging tools. We explained each category's methodology and introduced its most relevant representatives. Additionally, we discussed each type's advantages, limitations, and use cases. We then compared the performance of different scanners in a controlled environment by evaluating their ability to identify different TLS configurations while monitoring the scanning costs. The conclusion was that DissecTLS is the most efficient scanner, providing the precision of debugging tools while generating minimal traffic, typically a feature of fingerprinting tools. It should be mentioned, however, that the experiment was conducted in an artificial environment. The results could be slightly

 doi: 10.2313/NET-2025-05-1_16

different from reality. As part of future work, we could scan actual servers within the TLS ecosystem and then compare the scanners' performance. This would provide more realistic and precise results.

References

[1] R. Holz, J. Hiller, J. Amann, A. Razaghpanah,T. Jost, N. Vallina-Rodriguez, O. Hohlfeld, "Tracking the deployment of TLS 1.3 on the Web: A story of experimentation and centralization," *ACM SIGCOMM Computer Communication Review, Volume 50, Issue 3*, 2020.

[2] R. Holz, J. Amann, A. Razaghpanah, and N. Vallina-Rodriguez, "The era of tls 1.3: Measuring deployment and use with active and passive methods," 2019. [Online]. Available: https://arxiv.org/abs/1907.12762

[3] J. Althouse, A. Smart, RJ Nunnally, M. Brady, "Easily identify malicious servers on the internet with jarm." [Online]. Available: https://engineering.salesforce.com/easily-identify-malicious-servers-on-the-internet-with-jarm-e095edac525a/

[4] O. Gasser, M. Sosnowski, P. Sattler, J. Zirngibl. (2022). [Online]. Available: https://github.com/tumi8/goscanner

[5] Z. Durumeric, D. Adrian, A. Mririan, M. Bailey, J. Alex Haldermann, "A search engine backed by internet-wide scanning," *CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[6] M. Sosnowski, J. Zirngibl, P. Sattler, and G. Carle, "Dissectls: A scalable active scanner for tls server configurations, capabilities, and tls fingerprinting," in *Passive and Active Measurement*, A. Brunstrom, M. Flores, and M. Fiore, Eds. Cham: Springer Nature Switzerland, 2023, pp. 110–126.

[7] A. Diquet. [Online]. Available: https://github.com/nabla-c0d3/sslyze

[8] Dr. Wetter. Testing tls/ssl encryption. [Online]. Available: https://testssl.sh/

[9] J. Althouse, A. Smart, RJ Nunnally, M. Brady. [Online]. Available: https://github.com/salesforce/jarm

[10] M. Sosnowski, J. Zirngibl, P. Sattler, G. Carle, C. Grohnfeldt, M. Russo, D. Sgandurra, "Active TLS Stack Fingerprinting: Characterizing TLS Server Deployments at Scale," 2022. [Online]. Available: {https://arxiv.org/abs/2206.13230v1}

[11] K. Moriarty, S. Farrell. (2021) Deprecating tls 1.0 and tls 1.1. [Online]. Available: https://www.rfc-editor.org/info/rfc8996

[12] I. E. T. Force. (2008) The transport layer security (tls) protocol version 1.2. [Online]. Available: https://www.rfc-editor.org/info/rfc5246

[13] ——. (2018) Rfc 8446: The transport layer security (tls) protocol version 1.3. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446#page-60

[14] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 98–107. [Online]. Available: https://doi.org/10.1145/586110.586125

[15] M. Driscoll. The illustrated tls 1.3 connection. [Online]. Available: https://tls13.xargs.org/

[16] I. E. T. Force. (2014) Transport layer security (tls) application-layer protocol negotiation extension. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7301#page-3

[17] P. Sirohi, A. Agarwal, and S. Tyagi, "A comprehensive study on security attacks on ssl/tls protocol," in *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, 2016, pp. 893–898.

[18] [Online]. Available: https://search.censys.io/

[19] W. Mayer and M. Schmiedecker, "Turning active tls scanning to eleven," in *ICT Systems Security and Privacy Protection*, S. De Capitani di Vimercati and F. Martinelli, Eds. Cham: Springer International Publishing, 2017, pp. 3–16.