ZTLS: A DNS-based Approach to Zero Round Trip Delay in TLS handshake

Sangwon Lim
Seoul National University
Seoul, Republic of Korea
sangwonlim@snu.ac.kr

Hyeonmin Lee
Seoul National University
Seoul, Republic of Korea
min0921110@snu.ac.kr

Hyunsoo Kim
Seoul National University
Seoul, Republic of Korea
wayles@snu.ac.kr

Hyunwoo Lee
Korea Institute of Energy Technology (KENTECH)
Naju, Republic of Korea
hwlee@kentech.ac.kr

Ted "Taekyoung" Kwon
Seoul National University
Seoul, Republic of Korea
tkkwon@snu.ac.kr

ABSTRACT

Establishing secure connections fast to end-users is crucial to online services. However, when a client sets up a TLS session with a server, the TLS handshake needs one round trip time (RTT) to negotiate a session key. Additionally, establishing a TLS session also requires a DNS lookup (e.g., the `A` record lookup to fetch the IP address of the server) and a TCP handshake. In this paper, we propose ZTLS to eliminate the 1-RTT latency for the TLS handshake by leveraging the DNS. In ZTLS, a server distributes TLS handshake-related data (i.e., Diffie-Hellman elements), dubbed `Z-data`, as DNS records. A ZTLS client can fetch `Z-data` by DNS lookups and derive a session key. With the session key, the client can send encrypted data along with its `ClientHello`, achieving 0-RTT. ZTLS supports incremental deployability on the current TLS-based infrastructure. Our prototype-based experiments show that ZTLS is 1-RTT faster than TLS in terms of the first response time.

CCS CONCEPTS

• **Security and privacy** → **Web protocol security**; **Security protocols**.

KEYWORDS

Transport Layer Security, TLS, Performance, Latency

ACM Reference Format:
Sangwon Lim, Hyeonmin Lee, Hyunsoo Kim, Hyunwoo Lee, and Ted "Taekyoung" Kwon. 2023. ZTLS: A DNS-based Approach to Zero Round Trip Delay in TLS handshake. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30–May 04, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583516

WWW '23, April 30–May 04, 2023, Austin, TX, USA

ACM ISBN 978-1-4503-9416-1/23/04…$15.00
https://doi.org/10.1145/3543507.3583516

1 INTRODUCTION

As online service users are sensitive to latency in responses from servers [17, 59], it is crucial for content providers to provide low-latency services. According to [50], Amazon finds that every 100ms of latency costs them a 1% drop in sales, and Google finds that an extra 0.5 seconds for generating search page drops traffic by 20%. Also, it is reported that users tend to visit a website less often if it is slower than a close competitor by more than 250 milliseconds [42].

With the growing concerns about security and privacy in online communications, Transport Layer Security (TLS) [45] has become the *de facto* standard to protect user privacy and prevent tampering [2]. However, TLS requires one round-trip to establish a secure session between a client and a server, in addition to a DNS [8] lookup and a TCP handshake [26]. Thus, it is challenging to reduce the delay in setting up a TLS session for user satisfaction [18].

To address such a challenge, many approaches have been proposed to reduce the latency of the TLS handshake. After a comprehensive analysis, we find that the approaches in the literature are classified into three categories. First, some approaches [27, 45] reduce the number of round-trips of the TLS handshake. The approaches [38, 39] in the second category attempt to send the application data with the handshake messages simultaneously. Third, Bohannon [15] seeks to reduce the length of a round-trip time (RTT) by placing a proxy near a client.

In this paper, we propose a novel approach, ZTLS, that leverages the DNS to *reduce 1-RTT* in the TLS handshake. As a DNS lookup by a client is required before the TLS handshake, our main idea is that if a server delivers its cryptographic information (e.g., Diffie-Hellman elements for a key negotiation) simultaneously with its IP address through DNS records, 1-RTT can be reduced in the TLS handshake.

To this end, we design `Z-data` that contains a server's cryptographic information, which can be published in advance to the server's authoritative DNS server. `Z-data` also includes information such as a signature and a certificate, to provide authentication for its issuer (i.e., domain) and integrity of itself without additional mechanisms such as DNSSEC [48][1]. Before initiating the TLS handshake, a client fetches a server's IP address as well as `Z-data`

[1] As of December 2016, less than 1% of .com, .org, and .net sites have deployed DNSSEC, among which about one-third are not working properly [28].

simultaneously[2]. With `Z-data`, a client can generate a session key to encrypt its data and send "encrypted" application data to the server with a 0-RTT delay. In this way, ZTLS effectively reduces the latency to establish a secure channel compared to TLS, which provides users with a faster response.

Furthermore, we design ZTLS to be *backward compatible*. A ZTLS client determines whether a connecting server supports ZTLS by whether the `Z-data` of the server exists in the DNS, and a ZTLS server decides whether it conducts ZTLS or not, based on the presence of an extension for ZTLS in the received `ClientHello`.

We summarize the contributions of this paper as follows.

- We design a novel technique to enable clients can send encrypted data with a 0-RTT delay. ZTLS is the first approach that leverages the DNS to reduce network latency in the TLS handshake while supporting backward compatibility with the standard TLS protocol.
- We implement a prototype of ZTLS and publicly release the source codes[3].
- To show the feasibility of ZTLS, we conduct a comprehensive evaluation and show that ZTLS reduces 1-RTT[4] of the latency required for a client to get the first application data from a server.

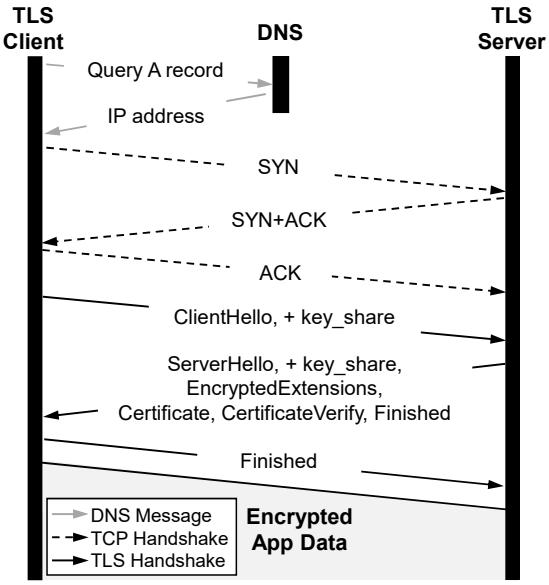
2 BACKGROUND

When a TLS client (e.g., a browser) intends to send encrypted application data to a TLS server, the client should take several steps before sending the data (see Figure 1). First, the client sends a DNS query to a DNS resolver to obtain the IP address of a server of interest. Next, the client and the server make a TCP connection by performing the three-way TCP handshake. Finally, on top of the TCP connection, a TLS session is established through the TLS handshake.

Domain Name System (DNS). The DNS is a globally distributed database that maps domains to their associated information (using DNS records [8]), and the information (i.e., DNS records) is provided through its authoritative name server. Usually, the authoritative name server of a domain is close to the domain's other servers like web and mail servers; however, the name server may not be close to its potential clients. On the client side, DNS resolvers send DNS queries recursively from the root name server down to the authoritative name server on behalf of client applications. They also cache the results of the DNS queries for a certain period.

One of the most important DNS records is an `A` record that contains the IPv4 address of a domain (i.e., its server). As shown in Figure 1, a TLS client first fetches the server's `A` record to obtain its IP address. In addition to `A`, there are many other DNS records with their own purposes. For instance, a `TLSA` record [31] contains a server's certificate or public key, and a `TXT` record [8] was originally intended for human-readable notes; however, it is also used to store machine-readable data such as security information [33, 36, 37, 43].

[2]DNS lookups can be conducted in parallel.
[3]ZTLS library — https://doi.org/10.5281/zenodo.7597964. ZTLS client and server — https://doi.org/10.5281/zenodo.7597982.
[4]Averaged round trip times toward Alexa top 1M sites over wired networks are Eastern N. America: 51.7ms, Western N. America 61.2ms, South America: 102.9ms, Western Europe: 40.5ms, South Africa 138.1ms, East Asia: 120.9ms, South East Asia: 136.2, and Oceania: 126.6 [40]. In wireless environments such as 3G, RTTs are usually longer [44].



+ Indicates noteworthy extensions sent in the previously noted message.

Figure 1: A client first gets a server's IP address from DNS. Next, the client and the server establish a TCP connection. Finally, they establish a TLS session over the TCP connection.

When sending a query/response for a DNS record, the 'DNS-over-UDP/53' protocol [8] is used by default. The maximum message size of this protocol was 512 bytes at the time of its design but became 4,096 bytes after EDNS [23] appeared in 1999. However, the maximum transmission unit (MTU) (The default is 1,500 bytes[5]) [7] is the practical limit. This is because if a DNS response size exceeds this limit, IP fragmentation [6] occurs and the response is retransmitted with the 'DNS-over-TCP/53' protocol [8], which results in a significant delay.

Transport Layer Security (TLS). TLS provides a secure channel between two communicating endpoints: a TLS client and a TLS server. Security properties of TLS include server authentication[6], confidentiality, and integrity. First, server authentication means that a client should be able to authenticate a server during a TLS handshake. A TLS server sends its `Certificate` and its signature (`CertificateVerify`) over the TLS handshake. By verifying the signature, the client authenticates the server. Second, confidentiality and integrity mean that no one other than the two endpoints can read/modify/write data exchanged between them. To this end, both TLS endpoints negotiate the encryption method (e.g., AES-256) and hash method (e.g., SHA-256) through the TLS handshake. They also establish a session key employing an HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [34] during the handshake. From TLS 1.3 [45], it is mandatory for TLS endpoints to use the Ephemeral Diffie-Hellman Key Exchange method [30] to provide forward secrecy[7] [19]. Overall, a TLS handshake entails 1-RTT overhead in TLS 1.3 (see Figure 1).

[5]In practice, the recommended maximum payload size value is 1,280-1,410 bytes [23].
[6]Client authentication is optional.
[7]It guarantees the session key is not compromised even if the related long-term key is compromised.

3 RELATED WORK

TLS handshake latency reduction. The TLS 1.3 handshake [45] reduces the number of RTTs to one in contrast to two RTTs in the TLS 1.2 handshake [24]. There is also a simplified TLS handshake to establish a subsequent session in a shorter time compared to a full handshake. There are a couple of simplification mechanisms. First, in Session ID Resumption [24], a client uses the session ID, provided by a server in the first session, to resume the session. The server should keep the state mapped to the session ID. Second, in Session Ticket Resumption [27, 49], a client employs the session ticket encrypted by a key known only to a server. This ticket contains the session key and other information necessary to resume the session. Thus, the server does not need to maintain session states in its storage. TLS endpoints can arrange a Pre-Shared Key (PSK), which includes a database lookup key like Session ID Resumption or a self-encrypted and self-authenticated value like Session Ticket Resumption [45]. The PSK mode supports 0-RTT Data, which means a client can transmit encrypted application data (so-called "early data") along with a PSK on the first flight [45].

In TLS Snap Start [38], a client sends encrypted application data without a PSK on the first flight. For subsequent TLS connections, a client generates a `ServerHello` value based on the previous `ServerHello` and a specific rule, derives a session key using a `ClientHello` and the `ServerHello`, encrypts the application data based on the key, and transmits the data along with the `ClientHello` and the `ServerHello` on the first flight.

In TLS False Start [39], a client transmits encrypted application data 1-RTT faster than the standard TLS 1.2 handshake. In TLS 1.2, a client sends application data after the TLS handshake is completed, while in TLS False Start, a client sends the data after receiving `ServerHelloDone` before Server's `Finished` arrives.

In addition, there is a study that utilizes a proxy for TLS handshakes. A patent from Facebook [15] proposes a mechanism that a proxy device located near a user negotiates cipher suites on behalf of its principal server. In this way, the delay of the TLS 1.2 handshake can be reduced.

TCP handshake latency reduction. TCP Fast Open (TFO) [21] removes the 1-RTT required for the TCP handshake; thus, it can be used to accelerate TLS session establishment [10, 20, 47]. In TFO, a server shares a TFO cookie in the initial TCP connection with a client, and the client utilizes the cookie in the subsequent connections.

QUIC [32] runs the TLS handshake over UDP [55]; that is, it initiates the cryptographic negotiation without the TCP handshake. Thus, it can also accelerate the completion of the TLS handshake.

Takeaway. We get the following lessons from the literature:

- Several approaches such as 0-RTT Data and TLS Snap Start send encrypted application data with information indicating a pre-shared key or some information from which a key can be derived.
- It would be desirable if we can accelerate setting up the first TLS session (not to mention subsequent sessions). Most of the approaches [24, 27, 38, 45, 49] cannot be applied for establishing the first TLS session. Thus, their usage can be limited as the majority of HTTPS connections are reported to be the first sessions, not session resumptions [53].
- A proxy near clients can be used to reduce the setup latency [15]. This approach achieves its goal by reducing the physical distance for communications; however, it requires extra entities, which is costly and difficult to deploy and maintain.

Based on the above lessons, we decide to find a way to pass information to a client so that the client can derive a session key even before contacting the server. To this end, we turn our attention to the DNS as (1) the DNS lookup is usually performed before the TLS protocol, (2) there are many DNS record types, some of which can be flexibly used, and (3) the DNS infrastructure can disseminate server-related data at some points close to clients.

TLS enhancement leveraging the DNS. Several techniques [31, 46] have been proposed employing the DNS to make TLS more privacy-preserving and security-hardening. TLS Encrypted Client Hello (ECH) [46] is devised to protect the *privacy* of sensitive information (i.e., domain name) in the `ClientHello`. DANE [31] is introduced to mitigate the issue of fraudulent certificates. However, no approaches are presented to enhance the performance of the TLS handshake.

4 ZTLS HANDSHAKE DESIGN

This section describes ZTLS. We first present goals and a threat model that we consider in designing ZTLS. Then, we provide an overview of ZTLS, followed by its details.

4.1 Design considerations

Design goals. In designing ZTLS, we take into account the following goals:

- **Reduced first response latency:** A ZTLS client receives the first response faster than TLS 1.3 with a ZTLS server, and it has the same delay as TLS 1.3 with a TLS 1.3 server.
- **Backward compatibility:** Both a ZTLS client and a ZTLS server should be able to fall back to the TLS protocol if their counterpart does not support ZTLS.

Threat model. We assume a Dolev-Yao attacker [25] that fully controls the network. That is, the attacker can receive all the messages from the participating parties, and can modify, drop, reorder, or inject messages. However, the attacker is computationally bounded. For instance, the attacker can decrypt encrypted messages only if it has the corresponding encryption key. The attacker can also launch DNS-related attacks such as DNS poisoning [57]. We assume all the related parties are available; that is, we do not consider resource-exhaustion attacks [41].

4.2 Overview

We design ZTLS based on the TLS 1.3 handshake. ZTLS leverages the DNS system to remove the 1-RTT of TLS handshake (see Figure 2).

① A ZTLS server uploads information dubbed `Z-data` to its authoritative name server [8].

[8] Another server with one of the domain's certificates and its private key can also make and upload `Z-data` to the authoritative name server.

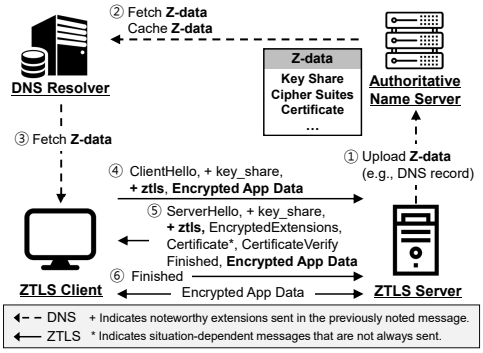


Figure 2: An overview of ZTLS operations is shown. A ZTLS client sends the first encrypted data along with its `ClientHello` with 0-RTT.

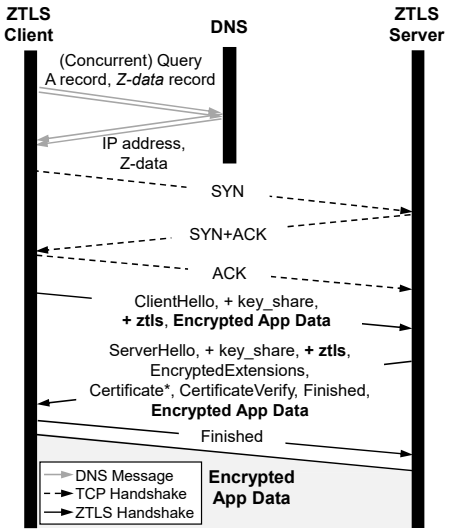
② DNS resolvers fetch and cache `Z-data`.
③ A ZTLS client obtains `Z-data` from a DNS resolver.
④ The client generates a session key from the `Z-data` and the information in its `ClientHello`, and hence sends its encrypted application data along with the `ClientHello`. In the `ClientHello`, there is an extension: {extension_type=ztls; extension_data=*Zdata_id*}. In this way, the ZTLS client sends its encrypted application data to the ZTLS server with *0-RTT delay*.
⑤ The server responds to the `ClientHello` with a `ServerHello` and transmits its encrypted application data along with the other messages. The `ServerHello` includes a "ztls" extension to confirm the selected *Zdata_id* or to handle exceptions (e.g., *Zdata_id* in the `ClientHello` is expired). If the server uses the same certificate as the `Z-data`, it may omit the certificate transmission.
⑥ If a `Z-data`-related exception occurs, the ZTLS client falls back to the standard TLS handshake. Otherwise, the ZTLS handshake is finalized with exchanging `Finished` messages.

The overall interactions among ZTLS-related entities are shown in Figure 3. First, a ZTLS client queries a DNS resolver to obtain the IP address and `Z-data` of a server of interest simultaneously using multi-threads. Once the `A` record arrives (i.e., even if the `Z-data` has not arrived yet), the ZTLS client sends a TCP SYN to the server. When the ZTLS client receives both the TCP SYN/ACK (from the server) and `Z-data` (from the DNS resolver), it initiates a ZTLS handshake. At the beginning of the ZTLS handshake, the ZTLS client has a key to encrypt its application data derived based on its `ClientHello` and `Z-data`. Obviously, the ZTLS client sends encrypted data 1-RTT faster than the TLS client as shown in Figure 1.

Reduced first response latency. In TLS 1.3, before a client sends initial encrypted data (say, an HTTP GET request) to a particular server, it obtains the IP address of the server through the DNS system, performs the TCP handshake, and exchanges `ClientHello` and `ServerHello` with the server to derive an encryption key and agree on the cipher suite as shown in Figure 1.

This procedure illustrates an anti-pattern, called *dependency on other computation* [14], which causes performance degradation. we adopt two tactics to counter this anti-pattern as follows[9].

[9]See Appendix A for details on why these two tactics were chosen.



+ Indicates noteworthy extensions sent in the previously noted message.
* Indicates situation-dependent messages that are not always sent.

Figure 3: We illustrate the message flow among the ZTLS-related entities.

(i) *Introducing concurrency*: We propose to simultaneously obtain the following information of a server: its IP address (`A` record), Diffie–Hellman (DH) elements, and certificate, all of which are required for the ZTLS client to establish a secure channel with the ZTLS server. This allows the ZTLS client to send encrypted data to the ZTLS server without waiting for 1-RTT for the key exchange.

(ii) *Maintaining multiple copies of data*: We design `Z-data` to be valid for a certain period so that ZTLS can take advantage of the caching effect of DNS resolvers. For this, the DH element in `Z-data` can be cached for a pre-defined period. To provide the same level of forward secrecy as TLS 1.3, *key_shares* (i.e., DH elements) of both peers are exchanged during the ZTLS handshake in the same way as the resumption procedure of TLS 1.3. The valid period of the same DH element should be determined by considering the industry practice. Thus, it does not exceed one hour at most [56].

Backward compatibility. To support the incremental deployability of ZTLS, we devise a mechanism by which a ZTLS client and a ZTLS server can identify whether its counterpart supports ZTLS or not. First, ZTLS clients can efficiently scan whether a server supports ZTLS by checking the presence of its `Z-data` in the DNS. Second, we propose a new TLS extension type, "ztls" to indicate ZTLS support. ZTLS servers can thus figure out whether a client supports ZTLS with the TLS extension in the `ClientHello`.

Changes required to clients, servers, and DNS to adopt ZTLS are detailed in Appendix D.

4.3 Detailed ZTLS design

`Z-data`. It has similar information as `ServerHello` but has additional features as it is delivered through a different channel. Table 1 shows the structure of `Z-data`[10]. Regular characters (e.g., 'v=') and spaces are used as delimiters, and italics are used as variables and

[10]An instance of `Z-data` is shown in Appendix B.

Table 1: Structure of `Z-data` is illustrated.

`Z-data` structure
v=*Ztls_version Validity_period_not_before Validity_period_not_after Max_early_data_size Zdata_id Key_share_named_group_enum Key_share_key_exchange* [*Not_supported_cipher_suites*] B_CERTIFICATE[2] *Certificate* E_CERTIFICATE[2] *Client_certificate_request Signature_scheme Signature_value*

are to be replaced with actual values. '[' and ']' indicate that what is written between them can be omitted.

`Z-data` contains the following fields: First, `Z-data` includes a server's cryptographic parameters. TLS 1.3 makes a TLS client and a TLS server exchange their cryptographic parameters using the Diffie–Hellman key exchange during a handshake for key derivation. ZTLS works almost the same way, but the difference is that a ZTLS client obtains cryptographic parameters of a ZTLS server through its `Z-data`. To this end, the *Key_share_named_group_enum* and *Key_share_key_exchange* fields are included in `Z-data`. The former indicates a DH group defined in TLS 1.3 and the latter is a DH public key. To agree on a particular cipher suite, a ZTLS server lists cipher suites that it does not support, which is specified in the *Not_supported_cipher_suites* field of `Z-data`. A ZTLS client chooses a cipher suite and informs the server of the chosen one through `ClientHello`.

Second, `Z-data` includes the *Certificate* and the *Signature_value* fields. The former is a ZTLS server's certificate, and the latter is the server's signature over `Z-data` to guarantee its integrity and authenticate its owner. We also add the *Signature_scheme* field to indicate which signature algorithm is used. *B_CERTIFICATE* and *E_CERTIFICATE* are inserted as delimiters before and after the *Certificate* field, respectively. To indicate the RawPublicKey [58] certificate type, a character '2' after the two delimiters can be followed. Otherwise, the certificate type is X.509 [22]. Note that a TLS server can optionally request a TLS client to send its authentication information. The *Client_certificate_request* field is for this.

Third, to prevent replay attacks, `Z-data` specifies its validity period by the *Validity_period_not_before* and *Validity_period_not_after* fields, which are expressed in the ISO 8601 [13] format.

Lastly, we add more fields for operational purposes. We introduce the *Ztls_version* field to indicate the version of ZTLS, which is needed for backward and forward compatibility. Also, as a ZTLS server may operate multiple entries of `Z-data` for temporal updates and spatial constraints, we add the *Zdata_id* field. Similar to TLS 1.3, a ZTLS server declares the maximum length of 0-RTT data to mitigate DoS attacks and not to use too much memory, which is specified in *Max_early_data_size*.

When a DNS response exceeds a certain limit, 'DNS-over-UDP/53' is reset to 'DNS-over-TCP/53' To prevent such an increase in setup latency, we split the `Z-data` into two records. The server's certificate is transmitted by a `TLSA` record[11], and the rest of the fields are by a `TXT` record[12].

[11] CERT record is also fine.

[12] Exploiting the `TXT` and `TLSA` records is a tentative solution, and there can be a new record type for the latter.

ZTLS server. For backward compatibility, a ZTLS server supports both ZTLS and TLS protocols. When the ZTLS server receives a `ClientHello`, it checks if there is an extension field whose *extension_type* value is 'ztls'. If so, it runs the ZTLS protocol; otherwise, it falls back to the standard TLS protocol. In the ZTLS protocol, the server first checks whether the application data does not exceed the maximum size. Then, it retrieves the corresponding Diffie-Hellman (DH) elements based on the *Zdata_id*, reads the client's DH elements (`key_share`), and derives the secret. Employing the HKDF, the session key is generated based on the secret and the handshake transcript. If a `Z-data`-related exception arises, the server notifies the client of the exception through the "ztls" extension in the `ServerHello` and ends the handshake. Otherwise, the server decrypts the encrypted application data with the session key. If the *Client_certificate_request* field is set, the server authenticates the client through the client's certificate (`Certificate`) and the client's signature (`CertificateVerify`). Next, the ZTLS server derives a new session key using HKDF with the `ClientHello` and its `ServerHello` for forward secrecy and sends its `ServerHello`, `EncryptedExtensions (EEs)`, and encrypted application data to the ZTLS client. As shown in Figure 3, the ZTLS server sends its certificate (`Certificate`) and signature (`CertificateVerify`) to prove that it is the owner of the domain. The server may omit to send its `Certificate` if it uses the same certificate as the `Z-data`. After that, the ZTS server transmits its `Finished` to confirm the integrity of the ZTLS handshake. Similarly, when the ZTLS client receives these messages, it derives the new session key and responds with its `Finished`. Last, the ZTLS server verifies the client's `Finished` to check the integrity of the handshake.

A ZTLS server is only required to additionally handle secrets (e.g., x's in g^x's) corresponding to *Zdata_ids* compared to a TLS server. We recommend performing ZTLS only on truly idempotent requests to prevent replay attacks following TLS and QUIC practices [29, 53] for 0-RTT Data.

ZTLS client. A ZTLS client runs three threads. The first thread queries the `TLSA` record, and the second one queries the `TXT` record and verifies the received `Z-data`. At the same time, the third thread queries the `A` record and makes a TCP handshake. If the server supports ZTLS, it makes a ZTLS handshake. If the server supports TLS only, it falls back to a TLS handshake. This is similar to the strategy used by Google Chrome to support QUIC [9, 11].

In the ZTLS protocol, after validating `Z-data`, the client chooses a cipher suite based on the *Not_supported_cipher_suites* and derives a session key by using HKDF with the *Key_share_named_group_enum*, the *Key_share_key_exchange*, and its own `ClientHello`. Next, the client encrypts its application data with the session key and sends the encrypted application data along with the `ClientHello` including a "ztls" extension containing the *Zdata_id* to a ZTLS server. If the *Client_certificate_request* field is set, the ZTLS client transmits its `Certificate` and signature (`CertificateVerify`) over its `ClientHello` along with its `ClientHello` and its encrypted data. After that, the client receives encrypted application data along with the other messages shown in Figure 3 from the server. The client verifies the `CertificateVerify` to check that the server that sent the `ServerHello` is the owner of the domain. The client derives the

Table 2: Security properties of data transmitted in ZTLS.

Data	Confidentiality	Integrity	Authentication
Z-data		✓	✓
ClientHello		✓	(✓)
First App data	✓	✓	✓
ServerHello		✓	✓
EEs	✓	✓	✓
Rest App data	✓	✓	✓

EEs: EncryptedExtensions, ✓: Required, (✓): Conditionally required

new session key using HKDF based on its ClientHello, the ServerHello, and the handshake transcript. With the key, it verifies the Finished (from the server) to check the integrity of the handshake and decrypts the encrypted data. Likewise, the client sends its Finished to the ZTLS server for the integrity of the handshake.

4.4 Security analysis

In this section, we review how ZTLS supports the security properties of the transmitted data (i.e., assets) against the threats considered in Subsection 4.1. Next, we check whether ZTLS can defend against attacks involving the DNS infrastructure.

Security properties. In this analysis, confidentiality means only authorized endpoints can access data, and integrity is guarding against improper data modification or destruction [3], and authentication is verifying the identity of the creator of data. We organize in Table 2 what security properties are required for the data transmitted in ZTLS and review whether ZTLS provides them.

(i) Z-data: Since Z-data contains only publicly available information, confidentiality is not required. The certificate and signature in Z-data guarantee the subject of the certificate is the domain's owner and that the information is not tampered with. The *Validity_period_not_before* and the *Validity_period_not_after* fields in Z-data indicate the validity of the Z-data at a given time.

(ii) ClientHello: A client's Finished, which is an HMAC [35], confirms the integrity of its ClientHello. Depending on the request of a ZTLS server, a ZTLS client sends its Certificate and signature (CertificateVerify) over its ClientHello, along with its ClientHello, to prove that the client is the owner of the certificate.

(iii) First App data: As mentioned earlier, a ZTLS client and a ZTLS server securely share a secret key using the Diffie–Hellman key exchange method for encrypting/decrypting the First App data. This cryptographic information is exchanged between both endpoints through Z-data and ClientHello. With the secret key (derived by each entity) and the encryption method (from the agreed cipher suite), the ZTLS client encrypts and sends data to the ZTLS server with 0-RTT. Similar to session resumption of TLS 1.3, ZTLS slightly sacrifices forward secrecy[13] to enhance performance. Note that we can control the risk level of compromising forward secrecy with the validity period. TLS 1.2 [24] and TLS 1.3 [45] each have a lifetime field in a session ticket to control their validity period. Similarly, in ZTLS, the lifetime of Z-data can be controlled through the *Validity_period_not_before* and the *Validity_period_not_after* fields. Thus, if an attacker obtains a server's private key (its DH element), the attacker can only decrypt 0-RTT data created within the validity period. This weakness is the same as that of the 0-RTT data in TLS 1.3 and is practically allowed[14]. To mitigate the risk, ZTLS recommends operating Z-data with a lifetime of less than one hour considering the industry practice[15] for 0-RTT data. Additionally, attackers cannot decrypt the following data because the key used to encrypt the first data (0-RTT data) is changed to an ephemeral key by the TLS/ZTLS handshake protocol. Thus, only the first data (typically HTTP GET) sent by clients within an hour is exposed to this risk in ZTLS. The integrity of the First App data and the authentication of the data creator are guaranteed through the negotiated hash algorithm and the derived secret symmetric key. As a countermeasure to replay attacks, we recommend performing ZTLS only on idempotent requests[16], which is the same as 0-RTT data practice in TLS[17] and QUIC [29, 53]. Fortunately, the first request is typically something idempotent like HTTP GET [56].

(iv) ServerHello, EEs, and Rest App Data: The way in which the security properties of the rest of the data are supported is the same as that of TLS 1.3. The Finished of a server ensures the integrity of its ServerHello and EncryptedExtensions (EEs), and the server's signature (CertificateVerify) guarantees that the server (who sent the ServerHello and the EEs) is the owner of the domain. The secret key for the EEs is derived in the above manner based on the ClientHello and the ServerHello and the handshake traffic. The Rest App Data is guaranteed its confidentiality, integrity, and authentication by the cipher suite and the secret key, which are negotiated in the handshake. Since the secret keys are derived based on ephemeral elements such as a ClientHello and a ServerHello, forward secrecy is satisfied for these encrypted data. As nonce, which is derived from the sequence number, is maintained independently at both sides, the non-replayability of the Rest App Data is provided.

Attacks to the DNS. As ZTLS exploits the DNS, attacks targeting the DNS need to be considered. Here, we do not consider attacks that aim to make DNS unavailable since they have the same effect on TLS and ZTLS, in which a client is unable to fetch DNS records (e.g., A record). Instead, we focus on attacks that forge or replay DNS messages.

DNS poisoning (also known as DNS spoofing) is an attack, in which an attacker tampers with DNS caches, causing a client to receive an incorrect response [51]. Recall that Z-data includes a signature to guarantee its integrity. Thus, a ZTLS client can easily detect if Z-data is altered. The ZTLS client that detects the manipulation simply falls back to the standard TLS protocol.

[13] Forward secrecy ensures that encrypted communications and sessions recorded in the past cannot be decrypted even if long-term secret keys are compromised in the future [52].

[14] 0-RTT Data is considered "too big a win not to do" [5].

[15] Cloudflare allows the session ticket keys for PSK encryption (which is used for 0-RTT data) for an hour [56].

[16] We recommend ZTLS servers reject a non-idempotent request as the first data. Then, a client falls back to the standard TLS handshake with a server. Note that this exception also occurs with 0-RTT DATA in TLS 1.3 practices and is handled in the same fashion [56].

[17] According to TLS 1.3 RFC [45], recording the random number of ClientHello can be a countermeasure against replay attacks depending on the server load and service characteristics.

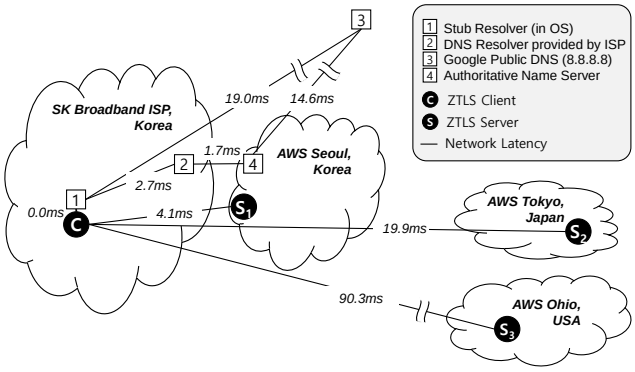


Figure 4: Experiment settings to compare ZTLS and TLS.

DNS replay attack is another attack that needs to be considered. In this attack, an attacker stores valid DNS responses and reuses them to impersonate a legitimate server (or a website). The purpose of this attack may be to exploit a leaked private key corresponding to the previously used *Key_share_key_exchange*. However, the attack cannot be successful due to the *Validity_period_not_before* and the *Validity_period_not_after* fields in `Z-data`. If `Z-data` is not currently valid, ZTLS clients fall back to the standard TLS protocol.

5 EVALUATION

We implement a prototype of ZTLS to evaluate its performance. We try to see how ZTLS effectively reduces the latency of the first response compared to TLS 1.3.

5.1 Prototype implementation

The implementation consists of three modules—ZTLS protocol library, ZTLS client, and ZTLS server. The ZTLS protocol library provides functions for the ZTLS handshake, which is implemented based on the OpenSSL [4]. Also, we implement a ZTLS client and a ZTLS server, which are applications that use the ZTLS protocol library. The ZTLS client has two modes: ZTLS mode and TLS mode. We publicly release all the source codes (ZTLS library, client, and server)[18].

5.2 Experiment setup

To measure the first response latency of ZTLS/TLS with various network settings among a client, a server, and a DNS resolver, we set up a testbed as shown in Figure 4. First, we deploy ZTLS servers at three locations employing Amazon Web Services (AWS) — `(S1)`: Seoul (AWS Asia Pacific), `(S2)`: Tokyo (AWS Asia Pacific), and `(S3)`: Ohio (AWS US East) — to measure the latency depending on the server location. Each server operates Ubuntu 20.04.3 LTS on an AWS EC2 t2.micro machine[19]. The ZTLS client runs on Ubuntu 20.04.3 LTS with systemd v251-rc1 that fixes the EDNS0 bug[20], which operates on a laptop that has a 7th Gen Intel(R) CPU i5-7500 @ 3.40GHz and 8G RAM. The client is located in Seoul. When the client sends DNS queries, EDNS0 is applied, but DNSSEC is not applied. Also, we set up an authoritative name server that sends `Z-data` by installing Bind9 [1] on Ubuntu 20.04.3 LTS operating on an AWS EC2 t2.micro machine. We test the three cases of DNS resolvers—a stub resolver, a local resolver in the same LAN as the client, and a resolver outside LAN (e.g., Google public DNS)—to measure the latency depending on the location of a resolver that caches the `Z-data`. The number right above each straight line between two entities in Figure 4 is the one-way latency; for example, the latency between the ZTLS client and the ZTLS server in `(S1)`: Seoul is 4.1ms. In the experiments, a client alternately performs the ZTLS connection setup after 1-second sleep and the TLS connection setup after 1-second sleep, respectively, for 210 times. This setting is to minimize the effect of Internet traffic fluctuation on the performance comparison. Here, the response is the application data of five bytes long sent by the ZTLS/TLS server without using any additional protocol like HTTP.

5.3 The first response latency

Using the above environment, we conduct experiments to evaluate the first response time of ZTLS/TLS. Here, the first response time is the interval from the moment the client starts the DNS lookup for the domain name of the server to the moment that the server's response (i.e., the five-byte app data) arrives at the client over a ZTLS/TLS connection.

For numerical evaluation, there are three cases depending on the cached place of `Z-data` that a ZTLS client fetches.

- **Case 1. Stub resolver:** ([1] in Figure 4), It is located on an OS. This case may happen when a user closes her Internet browser and then she visits the same ZTLS/TLS server again.
- **Case 2. Local DNS resolver:** ([2] in Figure 4), The resolver is in the same LAN as the client. This case may happen when a user opens his Internet browser for the first time after turning on a laptop.
- **Case 3. Public DNS resolver:** ([3] in Figure 4), it is located outside the LAN. Google Public DNS is an example. This case may happen when a user selects a public DNS resolver.

Table 3 shows the experiment results[21]. Looking at the first response times of ZTLS and TLS of case 1 in Table 3, the client receives the responses from the server in Seoul `(S1)` with ZTLS earlier than with TLS (average 3.7 ms, and median 3.3 ms faster). In the case of Tokyo `(S2)`, ZTLS is faster than TLS by 35.1 (34.3) ms[22]. With the farthest server in Ohio `(S3)`, ZTLS is faster than TLS by 177.8 (178.0) ms. Clearly, the results show that ZTLS is about 1-RTT faster than TLS. When a local resolver is employed (case 2), it shows a similar result to case 1; thus, we omit the detailed explanation. If the client uses a public DNS resolver (case 3), the overall delays are increased due to the distance to the public DNS resolver from the client. Nevertheless, the performance advantage of ZTLS is similarly observed. The differences between ZTLS/TLS response time are Seoul: 3.9(4.0) ms, Tokyo: 29.3 (33) ms, and Ohio: 172.8 (175.2) ms. As Internet connectivity becomes pervasive, there would be increasingly more networking environments with long RTTs.

[18] Please refer to section 1 to see the DOIs of the source codes.
[19] AWS EC2 t2.micro machine has an Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz and 1G RAM.
[20] See Appendix C for details.
[21] Please refer to Appendix E to see the tendency of the full results.
[22] Unless otherwise stated, the numbers represent the average and median values, the median value is shown in parentheses.

Table 3: Delays of the first responses for each case of DNS resolvers and ZTLS server locations (unit: ms).

Case	Stub Resolver						Local (ISP) DNS Resolver						Public DNS Resolver					
	Seoul		Tokyo		Ohio		Seoul		Tokyo		Ohio		Seoul		Tokyo		Ohio	
	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS	ZTLS	TLS
Median	26.1	29.4	88.4	122.7	366.5	544.5	33.2	38.2	95.9	132.9	367.4	544.2	63.8	67.8	127.5	160.5	405.2	580.4
Average	25.3	29.0	88.6	123.7	364.3	542.1	33.0	38.4	95.0	133.3	367.5	540.6	66.6	70.5	134.3	163.6	405.4	578.2

Table 4: DNS lookup times for each record (unit: ms).

	TLS	ZTLS		
	A	A	TXT	TLSA
Public DNS Resolver	40.4 (37.8)	43.7 (38.9)	42.3 (38.1)	41.2 (39.4)
Local DNS Resolver	6.1 (5.8)	7.4 (8.0)	6.3 (6.0)	7.9 (7.5)
Stub Resolver	0.5 (0.5)	0.5 (0.5)	0.3 (0.3)	0.6 (0.5)

average (median)

We believe the results of the server in Ohio (S3) are of importance for applications with low latency requirements.

Next, to observe ZTLS performance more clearly, we analyze the performance of each part of ZTLS (in Figure 3).

(i) *DNS lookup time*: Table 4 shows the average and median of DNS record lookup times for each record required for TLS (i.e., A record) and ZTLS (i.e., A, TXT, TLSA record) handshakes, respectively. Let us first analyze the time difference between single DNS queries (for TLS) and parallel DNS queries (for ZTLS). Although parallel DNS queries are slower by an average of 0~3.3ms (median 0~2.2ms), this does not affect the overall delay noticeably. The next item for analysis is the time it takes to look up a DNS record that is larger than the A record, such as a TXT record or a TLSA record[23]. We design the structure of Z-data in such a way that the records for ZTLS do not suffer truncated responses. Thus, the lookup speed to obtain the three records (in parallel) for ZTLS is similar to that of TLS which fetches only the A record.

(ii) *Handshake time*: Let us now see how fast the ZTLS handshake is performed compared to the TLS one. To answer this, we measure the handshake time of each protocol. When a client uses its local DNS resolver, the ZTLS handshake is faster than the TLS handshake across all server locations; Seoul: 8.1 (8.2) ms, Tokyo: 38.5 (38.4) ms, Ohio: 174.2 (177.1) ms. This result reveals that the ZTLS handshake is about 1-RTT (Seoul: 8.2ms, Tokyo: 39.8ms, Ohio: 180.6ms, which is shown in Figure 4) faster than the TLS handshake. The relative ZTLS handshake gains of the other cases are as follows— the public DNS Resolver case (Seoul: 4.5 (5.2) ms, Tokyo: 28.5 (33.3) ms, Ohio: 176.1 (176.5) ms) and the stub DNS Resolver case (Seoul: 3.2 (3.8) ms, Tokyo: 35.4 (34.8) ms, and Ohio: 177.0 (177.5) ms).

The above results show that leveraging the DNS (i.e., fetching TXT and TLSA records for Z-data) does not impose additional delays to ZTLS. Also, by exploiting the DNS, the ZTLS handshake reduces 1-RTT compared to TLS one, which effectively reduces the first response latency.

[23]Each response message size in the experiment is as follows. A record: 63 bytes, TXT record: 555 bytes, and TLSA record: 759 bytes.

6 DISCUSSIONS

DNS Time To Live (TTL) modification. Although it is a minor operational issue, we learn that some aberrant DNS resolvers perform cache updates not diligently when the TTLs of the cached records expire [16]. We also know that since the ECH protocol, which allows endpoints to share public keys through the DNS, cannot easily discard expired keys, it struggles with control of forward secrecy [12]. Thus, we add a validity period into Z-data. ZTLS clients can check the validity of Z-data, and if it is not valid, it falls back to standard TLS. Therefore, in ZTLS, expired keys can be easily ignored, and forward secrecy can also be controlled. To avoid meaningless verification failures, ZTLS recommends setting the TTL slightly shorter than the end of the validity period and employing multiple Z-data entries in an overlapping fashion.

Burden to DNS. For ZTLS, DNS resolvers have to respond to clients with TXT and TLSA records in addition to A records, which incurs a 200% increase in the number of packets at worst. However, TTL values of A records of popular sites (e.g., google.com) is 5 minutes and our recommendation for TTL values is 60 minutes. Thus, the increase in the number of packets exchanged between stub resolvers and DNS resolvers is 16.7% at best. Furthermore, DNS resolvers and authoritative name servers should additionally exchange TXT and TLSA records. The increase would be 16.7% as they would be exchanged by the TTL cycles. Note that the burden of ZTLS is similar to techniques leveraging DNS like ECH [46].

Z-data prefetching. Since Z-data is designed to be disseminated by any system regardless of its trustworthiness, there is no confidential information in it. Thus, to enhance performance, Z-data can be freely cached and prefetched by clients and DNS resolvers as similar to [54].

7 CONCLUSION

Most online services rely on TLS for secure communications. The security features of TLS require 1-RTT latency when establishing an encrypted session between a client and a server. In this paper, we propose ZTLS, which is the first approach that exploits the DNS to enable clients to send encrypted data with 0-RTT delay. For this purpose, we devise a new data structure, dubbed Z-data, to securely and efficiently disseminate a server's cryptographic information to its clients over the DNS infrastructure. Also, to support incremental deployment, ZTLS is designed to satisfy backward compatibility with the standard TLS protocol. We implement a prototype of ZTLS to demonstrate the feasibility of our design. Our prototype-based experiments show that ZTLS effectively reduces the 1-RTT delay for the first response from a server compared to TLS.

ACKNOWLEDGMENTS

The authors would like to appreciate the anonymous reviewers for their comments to improve our paper. Ted "Taekyoung" Kwon and Hyunwoo Lee are the co-corresponding authors.

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-0-02048) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation), and the KENTECH Research Grant (202200048A). Also, this research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No.2022R1A6A3A01087260). Lastly, this work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (NRF-2022R1A2C2011221).

REFERENCES

[1] [n. d.]. BIND9. https://www.isc.org/bind/. Retrieved: 2022-10-12.
[2] [n. d.]. Google Transparency Report. https://transparencyreport.google.com/https/overview?hl=en. Retrieved: 2022-10-12.
[3] [n. d.]. integrity - Glossary | CSRC - NIST Computer Security Resource Center. https://csrc.nist.gov/glossary/term/integrity. Retrieved: 2022-10-11.
[4] [n. d.]. OpenSSL. https://www.openssl.org/source/. Retrieved: 2022-10-12.
[5] [n. d.]. Rescorla, E.: TLS 1.3 (2015). http://web.stanford.edu/class/ee380/Abstracts/151118-slides.pdf. Retrieved: 2022-10-12.
[6] 1981. Internet Protocol. RFC 791. https://doi.org/10.17487/RFC0791
[7] 1984. A Standard for the Transmission of IP Datagrams over Ethernet Networks. RFC 894. https://doi.org/10.17487/RFC0894
[8] 1987. Domain names - implementation and specification. RFC 1035. https://doi.org/10.17487/RFC1035
[9] 2015. About enabling QUIC in android. https://groups.google.com/a/chromium.org/g/proto-quic/c/4fjpJ7hUtgg. Retrieved: 2022-10-12.
[10] 2016. Building a faster and more secure web with TCP Fast Open, TLS False Start, and TLS 1.3. https://blogs.windows.com/msedgedev/2016/06/15/building-a-faster-and-more-secure-web-with-tcp-fast-open-tls-false-start-and-tls-1-3/. Retrieved: 2022-10-12).
[11] 2017. QUIC fallback to TCP scenario. https://groups.google.com/a/chromium.org/g/proto-quic/c/zo7--OQLQBo. Retrieved: 2022-10-12.
[12] 2018. Encrypt it or lose it: how encrypted SNI works. https://blog.cloudflare.com/encrypted-sni/. Retrieved: 2022-10-12.
[13] 2019. ISO8601. https://www.iso.org/standard/70907.html. Retrieved: 2022-10-12.
[14] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.
[15] Philip Lewis Bohannon. 2017. Transport layer security latency mitigation.
[16] Guillaume Bonnoron, Damien Crémilleux, Sravani Teja Bulusu, Xiaoyang Zhu, and Guillaume Valadon. 2016. *Survey and analysis of DNS infrastructures*. Research Report. CNRS. https://hal.archives-ouvertes.fr/hal-01407640
[17] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. 2017. Why Is the Internet so Slow?!. In *Passive and Active Measurement (PAM)*, Mohamed Ali Kaafar, Steve Uhlig, and Johanna Amann (Eds.). Springer International Publishing, Cham, 173–187.
[18] Bob Briscoe, Anna Brunstrom, Andreas Petlund, David Hayes, David Ros, Ing-Jyh Tsang, Stein Gjessing, Gorry Fairhurst, Carsten Griwodz, and Michael Welzl. 2016. Reducing Internet Latency: A Survey of Techniques and Their Merits. *IEEE Communications Surveys & Tutorials* 18, 3 (2016), 2149–2196. https://doi.org/10.1109/COMST.2014.2375213
[19] Ran Canetti, Shai Halevi, and Jonathan Katz. 2003. A Forward-Secure Public-Key Encryption Scheme. In *Advances in Cryptology — EUROCRYPT 2003*, Eli Biham (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 255–271.
[20] Shan Chen, Samuel Jero, Matthew Jagielski, Alexandra Boldyreva, and Cristina Nita-Rotaru. 2019. Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC. In *European Symposium on Research in Computer Security*. Springer, 404–426.
[21] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. 2014. TCP Fast Open. RFC 7413. https://doi.org/10.17487/RFC7413
[22] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and William Polk. 2008. Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280.
[23] Joao da Silva Damas, Michael Graff, and Paul A. Vixie. 2013. Extension Mechanisms for DNS (EDNS(0)). RFC 6891. https://doi.org/10.17487/RFC6891
[24] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. https://doi.org/10.17487/RFC5246
[25] Danny Dolev and Andrew C. Yao. 1983. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (1983), 198–208.
[26] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. https://doi.org/10.17487/RFC9293
[27] Pasi Eronen, Hannes Tschofenig, Hao Zhou, and Joseph A. Salowey. 2008. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077. https://doi.org/10.17487/RFC5077
[28] Pouyan Fotouhi Tehrani, Eric Osterweil, Jochen H. Schiller, Thomas C. Schmidt, and Matthias Wählisch. 2021. Security of Alerting Authorities in the WWW: Measuring Namespaces, DNSSEC, and Web PKI. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2709–2720. https://doi.org/10.1145/3442381.3450033
[29] Alessandro Ghedini. 2019. Even faster connection establishment with QUIC 0-RTT resumption. https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/. Retrieved: 2022-10-12.
[30] Daniel Kahn Gillmor. 2016. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919. https://doi.org/10.17487/RFC7919
[31] Paul E. Hoffman and Jakob Schlyter. 2012. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698. https://doi.org/10.17487/RFC6698
[32] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000
[33] Scott Kitterman. 2014. Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208. https://doi.org/10.17487/RFC7208
[34] Dr. Hugo Krawczyk and Pasi Eronen. 2010. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869. https://doi.org/10.17487/RFC5869
[35] H. Krawczyk, M. Bellare, and R. Canetti. 1997. RFC2104: HMAC: Keyed-Hashing for Message Authentication.
[36] Murray Kucherawy, Dave Crocker, and Tony Hansen. 2011. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376. https://doi.org/10.17487/RFC6376
[37] Murray Kucherawy and Elizabeth Zwicky. 2015. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489. https://doi.org/10.17487/RFC7489
[38] Adam Langley. 2010. *Transport Layer Security (TLS) Snap Start*. Internet-Draft draft-agl-tls-snapstart-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-agl-tls-snapstart/00/ Work in Progress.
[39] Adam Langley, Nagendra Modadugu, and Bodo Moeller. 2016. Transport Layer Security (TLS) False Start. RFC 7918. https://doi.org/10.17487/RFC7918
[40] Hyunwoo Lee, Doowon Kim, and Yonghwi Kwon. 2021. TLS 1.3 in Practice:How TLS 1.3 Contributes to the Internet. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 70–79. https://doi.org/10.1145/3442381.3450057
[41] U. Lindqvist and E. Jonsson. 1997. How to systematically classify computer security intrusions. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*. 154–163. https://doi.org/10.1109/SECPRI.1997.601330
[42] Steve Lohr. 2012. For Impatient Web Users, an Eye Blink Is Just Too Long to Wait. https://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html. Retrieved: 2022-10-12.
[43] Daniel Margolis, Mark Risher, Binu Ramakrishnan, Alex Brotman, and Janet Jones. 2018. SMTP MTA Strict Transport Security (MTA-STS). RFC 8461. https://doi.org/10.17487/RFC8461
[44] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 133–140. https://doi.org/10.1145/2674005.2674991
[45] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446
[46] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. 2022. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-14. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-14 Work in Progress.
[47] Florentin Rochet, Emery Assogba, Maxime Piraux, Korian Edeline, Benoit Donnet, and Olivier Bonaventure. 2021. TCPLS: Modern Transport Services with TCP and TLS. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 45–59. https://doi.org/10.1145/3485983.3494865
[48] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. 2005. DNS Security Introduction and Requirements. RFC 4033. https://doi.org/10.17487/RFC4033
[49] Joseph A. Salowey, Hao Zhou, Hannes Tschofenig, and Pasi Eronen. 2006. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507. https://doi.org/10.17487/RFC4507

[50] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. 2014. The Internet at the Speed of Light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (Los Angeles, CA, USA) *(HotNets-XIII)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/2670518.2673876
[51] Sooel Son and Vitaly Shmatikov. 2010. The Hitchhiker's Guide to DNS Cache Poisoning. In *Security and Privacy in Communication Networks*, Sushil Jajodia and Jianying Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483.
[52] Drew Springall, Zakir Durumeric, and J. Alex Halderman. 2016. Measuring the Security Harm of TLS Crypto Shortcuts. In *Proceedings of the 2016 Internet Measurement Conference* (Santa Monica, California, USA) *(IMC '16)*. Association for Computing Machinery, New York, NY, USA, 33–47. https://doi.org/10.1145/2987443.2987480
[53] Nick Sullivan. 2017. Introducing Zero Round Trip Time Resumption. https://blog.cloudflare.com/introducing-0-rtt. Retrieved: 2022-10-12.
[54] Srikanth Sundaresan, Nazanin Magharei, Nick Feamster, and Renata Teixeira. 2012. Accelerating Last-Mile Web Performance with Popularity-Based Prefetching. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Helsinki, Finland) *(SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 303–304. https://doi.org/10.1145/2342356.2342421
[55] Martin Thomson and Sean Turner. 2021. Using TLS to Secure QUIC. RFC 9001. https://doi.org/10.17487/RFC9001
[56] Filippo Valsorda. 2016. An overview of TLS 1.3 and Q&A. https://blog.cloudflare.com/tls-1-3-overview-and-q-and-a/. Retrieved: 2022-10-12.
[57] Zheng Wang. 2014. POSTER: On the Capability of DNS Cache Poisoning Attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1523–1525. https://doi.org/10.1145/2660267.2662363
[58] Paul Wouters, Hannes Tschofenig, John IETF Gilmore, Samuel Weiler, and Tero Kivinen. 2014. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250. https://doi.org/10.17487/RFC7250
[59] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. 2017. Where has my time gone?. In *International Conference on Passive and Active network measurement (PAM)*. Springer, 201–214.

A ARCHITECTURAL TACTICS CONSIDERED WHEN DESIGNING ZTLS

We discover a *dependency on other computation* anti-pattern [14], which causes performance degradation, in the necessary procedure before sending an initial HTTPS request.

Common tactics to solve the anti-pattern are *controlling resource demands* and *managing resources*. The former is a method of reducing resource demands through algorithm optimization or degradation of the quality of service, which is not relevant to the problem setting of this paper. Thus, We focus on *managing resources*.

Tactics typically used in the *managing resources* category include *increasing resources, introducing concurrency, running multiple copies of computations, maintaining multiple copies of data, bounding queue sizes, and scheduling resources efficiently* [14]. Among these approaches, *increasing resources* and *running multiple copies of computations* cannot be used due to the cost of operating additional machines. Also, *bounding queue sizes* and *scheduling resources* are not suitable for the problem domain of designing a handshake protocol. We adopt two techniques: *introducing concurrency* and *maintaining multiple copies of data*.

B AN INSTANCE OF Z-DATA

Among the fields of `Z-data` in ZTLS, the *Certificate* is delivered as a `TLSA` record, and the rest of the data is delivered as a `TXT` record. Table 5 is an instance of `Z-data` excluding the *Certificate* field in a `TXT` record.

Table 5: An instance of `Z-data` in a `TXT` record.

`Z-data` instance (`TXT` record)
" v=ztls1 " " 20211228035822z " " 20220108035822z " " 128000 " " 10000 " " 29 " " MCowBQYDK2VuAy EANQ9MK/3Cm4igzj+cdzQLzzEwRAOcGs jpjbGF/+yVzlY= " " N " " 2052 " " ozJjh2jihq2wWVdNLbwP6yISuvv 5pX5zfyZp6XZBjawp/LIv9oSKRMkghcKPHYWk MGpLBx5dw/ol4aBb g+/0DavS9HmeNB0YyHEoou37qLKnHBKh/fp8Tu7NeEXJxG2I lnIAn6 0ITnd3v/X7dEDmUEeB/y1c7A4XQjgIn3nNYES3O8EMbi4SEMyU1h9Y ds2V " " c94cvKaxyYK80k02h9oPN6iiO5HVtDXmgPmYEFRQUHDGnG TORLXSJShhKl6fODZH BZdjh+PYTfda3Xp/IohmjHUylf9aBasSqirzX6 4HeNwOTn5yxDGacHRGITGsqIwB tJ1iQ6kMw5NALq9LQFA 6Bg== "

C THE EDNS0 BUG IN UBUNTU OS

During the experiment setup, we found a bug in *systemd-resolved* of Ubuntu 20.04.3 LTS. The bug is that the maximum size of a UDP payload is set to 512 bytes. We found out that the bug had been reported to the *systemd-resolved* development community and had been fixed in a pre-release version, v251-rc1. We adopted the fix in our experimental settings. You can view the patch contents and source at the following link — https://github.com/systemd/systemd/commit/526fce9 and download the relevant update from the following link — https://github.com/systemd/systemd/releases/tag/v251-rc1.

D CHANGES REQUIRED TO CLIENTS, SERVERS, AND DNS

Note that ZTLS is incrementally deployable as it is designed in a backward-compatible fashion. Furthermore, changing clients, servers, and DNS is simple. First, the client and server can use ZTLS by upgrading the TLS library, where ZTLS is implemented, with a minor change on the client/server applications. The only change required for the client application is to query DNS for the IP address and `Z-data` together and provide the `Z-data` to the TLS library. The server application needs interfaces to access Zdata-related keys, which are similar to the ones to access Session Ticket Keys. Note that there are no changes required to DNS applications. Authoritative name servers only need to provide `TLSA` and `TXT` records.

E THE TENDENCY OF THE FIRST RESPONSE TIMES IN THE EXPERIMENT

Despite some jitters, we can observe that ZTLS provides a tendency for shorter response time than TLS in the experiment results which are plotted in Figure 5.

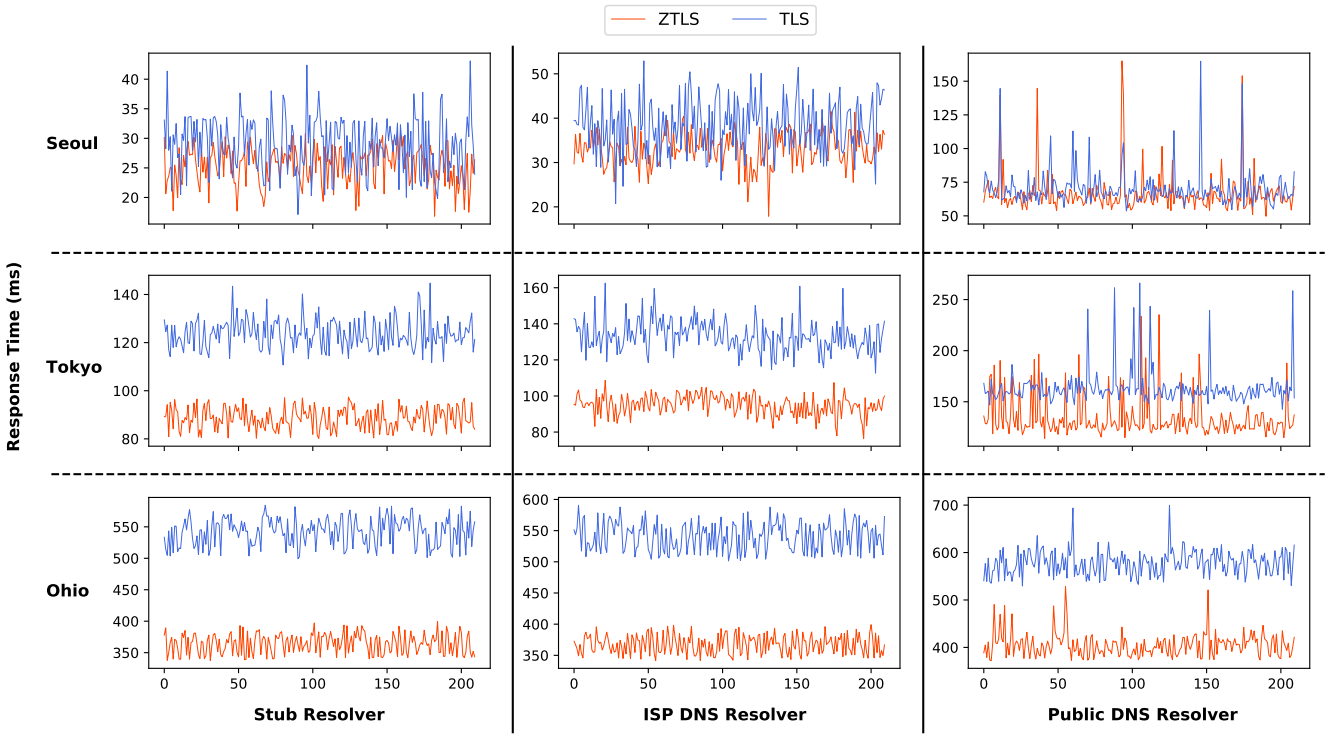


Figure 5: The tendency of the first response time for each case of DNS resolvers and ZTLS server locations.