# Delay Measurement of 0-RTT Transport Layer Security (TLS) Handshake Protocol

Danylo Goncharskyi*, Sung Yong Kim*, Ahmed Serhrouchni*, Pengwenlong Gu*, Rida Khatoun* and Joel Hachem†
* LTCI, Telecom Paris, Institut Polytechnique de Paris, France
† Rochester University, MI 48307, USA

*Abstract*—**Transport Layer Security (TLS) 1.3 was normalised in 2018, in which an efficient 0-rtt handshake protocol was proposed. For future 5G networks, the 0-RTT handshake will be a more suitable choice for both secrecy and efficiency. However, 4 years after it was proposed, the 0-rtt handshake protocol is still not widely accepted by network service providers due to concerns about its ability to resist replay attacks. In order to address this issue, many solutions have be proposed in the past few year but all of them will increase the complexity and overhead of the 0-RTT protocol. In this paper, we focus on testing whether the 0-RTT handshake protocol is supported by service providers, and testing its performance in a real network environment to verify whether it can withstand continuous optimization in terms of security. Test results show that with 0-RTT, the server received the first application data up to 37 time faster than the 1-RTT and up to 83 time faster than 2-RTT. However, at the client side, the performance of 0-RTT protocol is virtually the same as 1-RTT, as predicted.**

*Index Terms*—**Transport Layer Security, 0-rtt handshake protocol, Early data, Performance measurement, Benchmarking**

## I. Introduction

Transport Layer Security (TLS) is widely used for user authentication and encrypted data transmission. Its first version, TLS 1.0 [1] was published in 1999, which is originated as Secure Sockets Layer (SSL). In the following years, TLS 1.1 [2] and TLS 1.2 [3] were published in 2006 and 2008 respectively. Both of these versions use 2-rtt handshake. But compared to TLS 1.1, TLS 1.2 allows using more efficient authenticated encryption modes such as Galois/counter mode (GCM) to replace the traditional cipher block chaining (CBC) mode. The current version of TLS is TLS 1.3 [4], which was published in 2018. Compared to the TLS 1.2, the encryption algorithm is simplified in TLS 1.3 handshake by reducing the number of supported encryption protocols to 3, reduced the handshake complexity from 2-RTT to 1-RTT and even provide a 0-RTT handshake protocol for low delay networks.

As we know, in the future 5G era, a large number of ultra-reliable low latency communications (URLLC) applications have extremely high requirements for transmission delay. Therefore, the 0-RTT handshake will be a more suitable choice for both secrecy and efficiency. However, due to the simplified process of the 0-rtt handshake protocol, it also faces some serious security challenges in actual deployment, which leads to a result that 0-RTT handshake is still not widely supported by web service providers. The replay attack is the most dangerous one. If the resumption secret or the session ticket is captured by a malicious node, the adversary can launch large numbers of retries or duplicates them on multiple connections, which may lead to leakage of user information or denial of legitimate services [5].

In order to propose replay-resistant 0-RTT protocol, great efforts have been made in the past few years. In [6], a dynamic hash chain based authentication method is proposed to address the replay attack issues in smart grid networks. In which the key for each device is renewed at each data collection round based on hash-chain concept. In [7], the authors changed the 0-RTT handshake protocol proposed in RFC8446 by adding an "identity share" extension. In their proposed scheme, the shared key can be dynamically generated based on the identity (ie. the public key of client). In [8], a PPRFs based forward security and replay resilience protocol is provided, which can immediately be used in TLS 1.3 0-RTT and deployed unilaterally by servers, without requiring any changes to clients or the protocol. And in [9] the authors suggests to set a short wait time (200 ms for example) to verify that the packet isn't forged by interacting with the server multiple times. It can be observed that the most effective way to resist replay attacks during the 0-rtt handshake process is to constantly change the user's authentication information so that the previously leaked information cannot be repeatedly accepted by the server. However, the above solutions all increase the complexity and overhead of the 0-RTT protocol while enhancing its security.

In this paper, we focus on testing whether the 0-RTT handshake protocol is supported by service providers, and testing its performance in a real network environment to verify whether it can withstand continuous optimization in terms of security. Specifically, we did two different types of tests. Firstly, we chose 12 well-known service providers to test their 0-RTT usage and we found out that only cloudfare.com, google.com and youtube.com support early data. Then, in order to test the performance of the 0-RTT handshake protocol in a real network environment, we set up a OpenSSL server who support early data at Lyon and launch several tests from Palaiseau. We tested two widely used OpenSSL commands: *s_client* and our modified *s_time* to evaluate the performance of 0-RTT handshake protocol and compared it with 1-RTT and

2-RTT protocols. Test results show that with 0-RTT, the server received the first application data up to 37 time faster than the 1-RTT and up to 83 time faster than 2-RTT. However, at the client side, the performance of 0-RTT protocol is virtually the same as 1-RTT, as predicted.

The rest of the article is organized as follows: Section II presents in detail the 1-RTT and 0-RTT handshake protocols proposed in TLS 1.3. Section III provides the description of the test environment and our test methods. Test results and analysis are given in Section IV. We conclude this article in Section V.

## II. 0-RTT Handshake and Security Issues

In this section, we will briefly introduce the new mechanisms provided in the TLS 1.3 protocol

### A. *TLS 1.3 Handshake*

A TLS 1.3 full handshake is given in Fig. 1. As always, a *ClientHello* is sent to the server to kick off the handshake process. Note that in TLS 1.3, using the extensions ”supported groups” and ”key share”, the client sends the list of supported cypher suites and its key share in the *ClientHello*, which allows to reduce the handshake process from 2-RTT to 1-RTT. After received the *ClientHello*, the server reply the client a *ServerHello*, in which the chosen key agreement protocol, the server's key share as well as its certificate are comprised. After the exchange of keys, both sides have the following information: Client Random, Server Random, Client Params and Server Params. Then both sides can compute the ”Master Secret” using the DH and HKDF algorithms separately. Afterwards, the server sends a *ChangeCipherSpec* message to the client, and the following messages like certificate will all be encrypted. Then, the server sends its finish message to the client, which is a hash of the entire handshake up to this point. Finally, the client replies with its *ChangeCipherSpec* and finish messages to finish the handshake process.

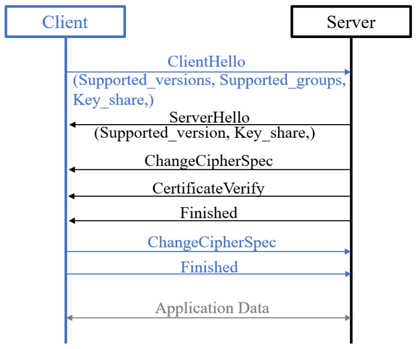


Fig. 1. 1-RTT handshanke of TLS 1.3.

Compared to the TLS 1.2, the encryption algorithm is simplified in TLS 1.3 handshake. The key exchange and signature algorithms can be negotiated by using extensions like ”supported groups”, ”key share” and ”signature algorithms” in the first flight message of both sides. This enables TLS 1.3 to start encrypted communication after only one round trip, which not only speeds up connections but also greatly improves the security of the handshake process.

Besides the improvements in standard handshake process, TLS 1.3 also support 0-rtt handshake for session resumptions or pre-shared key (PSK) based connections. In order to activate the 0-RTT handshake, the following three conditions must be satisfied:

- The session ticket received by the client in the previous handshake has ”max early data size” extension.
- In the process of PSK session resumption, the ”early data” extension is configured in the *ClientHello*.
- The ”early data” extension is configured in the server's *EncryptedExtensions* message.

If all these three conditions cannot be satisfied at the same time, session resumption will be done in 1-RTT mode.

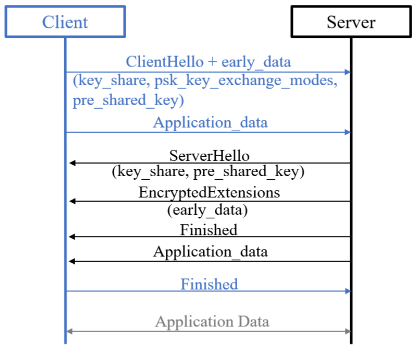


Fig. 2. 0-RTT handshanke of TLS 1.3.

The 0-RTT handshake process is illustrated in Fig. 2. During this process, the client firstly sends the early data along with the *ClientHello* message which contains the PSK identity information to resume the connection. Then the server verifies the PSK information (eg. SNI value, ticket age and cipher suites). If verified as correct, the server replies a *EncryptedExtensions* message with acceptance of early data connection. Note that in 0-RTT mode, the server can either request or not request for client certificate. If not requested, the server would drop ”Certificate” and ”CertificateVerify” in its finished message. At this stage, the 0-RTT handshake has been completed.

Normally, a full 1-RTT handshake is used in the first authentication between a server and a client and both sides can derive a resumption secret. Then, for session resumption, there are two standard ways that the server can verify the resumption secret provided by a client.

- Session Caches: In this way, the server stores all resumption secrets of recent session and issues each client a unique identity. Whenever a user requests to reconnect, the server compares the identity and resumption secret provided by the user in the 0-RTT messages with the information stored in its own database. If the verification is passed, the user is allowed to reconnect. One major drawback of this method is that for massive access scenarios, the server needs to store a large amount of user information, which leads to high storage cost and high look up delay.

- Session ticket: Using session ticket, the server does not need to cache the user information. Instead, at the beginning of each new session, the server issues a ticket to the client which contains the resumption secret and encrypt the ticket with a session ticket encrypt key (STEK)[1]. The encrypted ticket is stored by the client but cannot decrypt it. Thus, for session resumption, the client only need to send back the session ticket to the server in its 0-RTT messages, which allows the server to recover the resumption key and verify the identity of the user.

For the above two schemes, the common safety hazard is that they are neither forward secure nor replay resistant, especially in wireless scenarios since wireless channels are open to all users.

In TLS 1.3, the forward secrecy issue can be addressed using the Elliptic-curve Diffie–Hellman (ECDH) protocol, in which the session key generated upon the negotiated DH parameters is unique for one session and does not rely on the server's long-term keys. In this way, even the private key of client or server is leaked or compromised by an adversary, it is not possible to decrypt the previously sniffed traffic data. Even in 0-RTT mode, only the early data is encrypted by the resumption secret which guarantees a delayed forward secrecy. Moreover, several methods like puncturable pseudorandom functions (PPRFs) [10] are also proposed to enhance the forward secrecy in 0-RTT mode.

## III. Test Methods

Since not all service providers support TLS 1.3, we designed two tests in this article. First, we selected 12 widely used service providers to test whether they support 0-RTT TLS. Then, we tested the time it takes to establish a connection with different versions of TLS. To ensure fair testing, we set up a server allows TLS1.2, TLS1.3 and 0-RTT in Lyon and accessed it from Paris (The distance between the server and the client is about 400km). The architecture of the test platform is shown in Figure 3.

### A. 0-RTT TLS by Services Providers

For security reason, 0-RTT is disabled by many service providers since the replay attack is still a huge threat. Specifically, if the resumption secret or the session ticket is captured by a malicious node, the adversary can resend a large number of retries or duplicate them on multiple connections, which may lead to leakage of user information or denial of legitimate services.

For this reason, it is interesting to test which of the commonly used websites support 0-RTT and which do not. We launched the tests on the aforementioned 12 websites using the following script: *python -m sslyze -early_data*, the results are given in Table. I. It can be observed that most of the servers do not support early data. Among these 12 websites, only cloudfare.com, google.com and youtube.com support early data.

[1]The same STEK is used for many sessions and clients

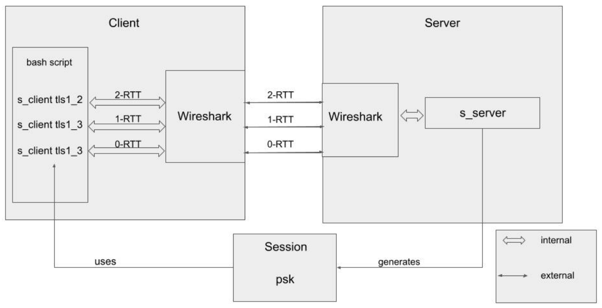


Fig. 3. The architecture of the test platform.

TABLE I
Status of Websites Supporting 0-RTT TLS

| Website | Early Data | Website | Early Data |
|---|---|---|---|
| cloudfare.com | support | google.com | support |
| youtube.com | support | amendes.gouv.fr | not support |
| telecom-paris.fr | not support | microsoft.com | not support |
| apple.com | not support | amazon.com | not support |
| facebook.com | not support | wikipedia.com | not support |
| github.com | not support | paypal.com | not support |

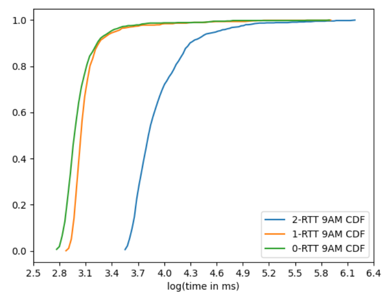
### B. Evaluation of Connection Time

In this part, we test the specific time it takes for different versions of TLS to establish a connection. In order to ensure the reliability of the test, we use two commands proposed in the OpenSSL: *s_client* and our modified *s_time*.

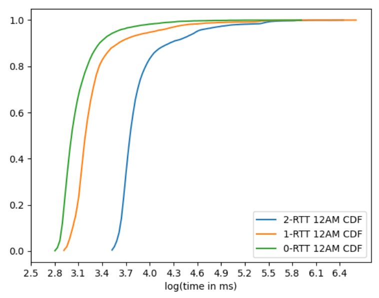
As illustrated in Fig 3, at the server side, we implemented the *s_server* of OpenSSL with the following options: *-accept 443 -cert cert.pem -key key.pem -early_data -no_anti_replay*, which forced the server to listen to the port 443 and enables early data. And with *-no_anti_replay* option, the anti-replay protection is switched off so that we can send more than two early data messages in a row.

At the client side, we use firstly the command *s_client* to measure the time for 100 connections with TLS 1.2, TLS 1.3 1-RTT and TLS 1.3 0-RTT then calculate the average time for each of them. Specifically, we use the *s_client* with the following options: *-connect address, -tls1_2 for TLS1.2* and *connect address, -tls1_3* for TLS1.3. And we use the function *time* to measure the time taken for 100 connections of each TLS version.

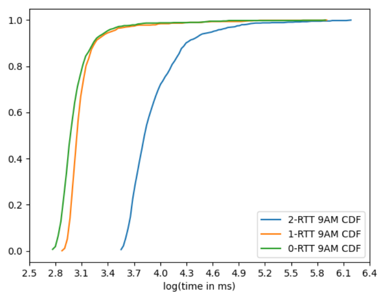
In order to enable the 0-RTT with OpenSSL, *s_client* needs to read two files: *early_data.txt* and *sessions*. In this way, we take the resumption secret firstly with a 1-RTT handshake using the following script: *sleep 1 — OpenSSL s_client -sess_out sessions -connect*. Then, we put the obtained



(a) Test results at 9AM

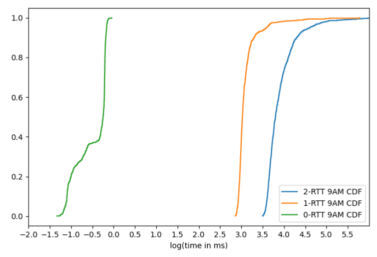


(b) Test results at 12AM

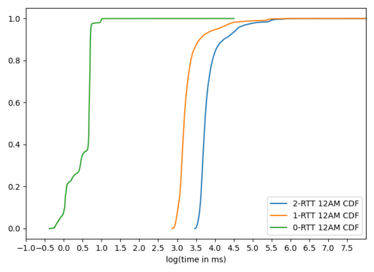


(c) Test results at 9PM

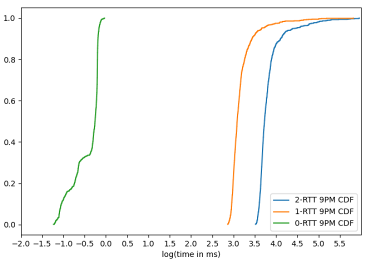
Fig. 4. The CDF of the handshake time at the client side.



(a) Test results at 9AM



(b) Test results at 12AM



(c) Test results at 9PM

Fig. 5. The CDF of the handshake time at the server side.

resumption secret in the file *sessions* and using the script *-sess_in sessions -early_data early_data.tx* to reuse the session and launch the 0-RTT handshake.

However, one major drawback of this method is that we cannot get the transmission delay of the first application data. What We have is the time the server takes to accept the first application data. It may impact the accuracy of our tests. In this way, the results obtained are given as follows: 107 ms per connection for 2-RTT, 85 ms for 1-RTT and 83 ms for 0-RTT.

Besides using the *s_client*, we also modified the *s_time* module of OpenSSL to enable 0-RTT handshake[2]. Specifically, the first problem is that the session should be saved either in the ram or in a file, which calls an *SSL_read* to receive the session information from the server. However, the first time usage of *SSL_read* might be block at the client side until it receives data from the server, which might not happen. To address this issue, we let the first *SSL_read* be called in a separate process which will be killed after use. We then need to save the session containing the psk in a file for the following 0-RTT handshake. Since the server sets a psk for one session, each new connection must reuse the previous session. This process forces us to redo the *SSL_read* and save the current session for the subsequent one. Moreover, it also requires that the server replies the early data. In our tests, although we stored the new psk for each round, 0-RTT is still 30 to 900% faster than 1-RTT (30% with fast internet, 900% with slow one).

[2]The modified *s_time* has been uploaded to the gitlab: https://gitlab.com/dang21/openssl-s_time.

Note that the *s_time* only measures the transmission speed at the client side but not at the server side. As aforementioned, according to the RFC 8446 [4], the only difference between the 0-RTT and the 1-RTT is that with 0-RTT, application data can be directly transmitted with the *ClientHello* as early data. This also motivates us to measure the time the server takes to accept the first application data, which will be discussed in next section.

## IV. Test Results

In order to measure the handshake time for different versions of TLS correctly, we propose to use the Wireshark to capture the traces generated by the test methods presented in Section III and calculate the transmission and process time for both the client and the server. Wireshark is a strong tool, which allows us to have more control over what we measure. For example, to measure the handshake time of 2-RTT and 1-RTT at the server side, we can calculate the time between the reception of ClientHello and the first application data arrived. However, for 0-RTT handshake, since the application data is transmitted with the *ClientHello*, we consider the time that the server takes to treat the early data also as a part of the handshake. This means that at the server side, we take the time between the ClientHello arrived and the ServerHello sent as the time takes for a handshake. Following the same rule, at client side, we measure the time between the *ClientHello* sent and the *ServerHello* arrived as the handshake time. However, for 0-RTT, since the client knows whether its early data is accepted after 1-RTT. We consider that the time for the 0-RTT handshake at client side is a ping delay plus the server side 0-RTT handshake time. For a fair comparison, we use the cipher TLS-AES-256-GCM-SHA384 for 1-RTT and 0-RTT and ECDHE-ECDSA-AES256-GCM-SHA384 for 2-RTT.

We launched the tests at 3 different time of the same day: 1000 times at 9 AM, 10000 times at 12 AM (noon) and 1000 times at 9 PM. After the capture using Wireshark, We then use a python script to obtain the results for 2-RTT, 1-RTT and 0-RTT without any system bias and then compute the average connection delay of each handshake protocol.

The CDF of the handshake time at client side and at the server side are illustrated in Fig. 4 and Fig. 5 respectively. Firstly, from the slope of the curve, it can be observed that 0-rtt and 1-rtt have stable performance in three different time periods (9AM, 12AM and 9PM). And without considering security, the two handshake protocols proposed in TLS 1.3, 1-rtt and 0-rtt have better performance on both the client side and the server side, especially the 0-rtt handshake protocol on the user side. The shortest handshake time can be achieved with both ends of the server.

The mean hand shake time at both client and sever side for all three different handshake protocols are give in Table. II. It can be observed that with 0-RTT, at the server side, it received the first application data up to 37 time faster than the 1-RTT and up to 83 time faster than 2-RTT. At client side, we measured the time between the *ClientHello* and the first application data sent for 2-RTT and 1-RTT. However, we cannot do the same ting for 0-RTT since the first application data is sent with the *ClientHello* as early data. Thus, we measured the time between the *ClientHello* sent and the *ServerHello* arrived instead. Therefore, it can be observed that at the client side, the performance of 0-RTT is virtually the same as 1-RTT, as predicted.

## V. Conclusion

In this paper, we focus on testing whether the 0-RTT handshake protocol is supported by service providers, and testing its performance in a real network environment to verify whether it can withstand continuous optimization in terms of security. To do so, we presented a client-server pair at a constant distance and launched the tests at 3 different time of the same day. For evaluation, we used Wireshark to optimally calculate the time of 0-RTT and 1-RTT exchanges. Test results show that with 0-RTT, the server received the first application data up to 37 time faster than the 1-RTT and up to 83 time faster than 2-RTT. However, at the client side, the performance of 0-RTT protocol is virtually the same as 1-RTT, as predicted.

For future work, our tests could be further improved. firstly, we can an option containing a psk file to both the *s_time* and the *s_server* for the 0-RTT handshake. Another possible improvement is to modify the function *s_server* to measure connection time throughout its activity, though it would be more complex to be achieved.

TABLE II
Handshake Time Comparison in second

| | Sever | | | Client | | |
|---|---|---|---|---|---|---|
| | 0-RTT | 1-RTT | 2-RTT | 0-RTT | 1-RTT | 2-RTT |
| 9AM | 0.00065 | 0.024 | 0.054 | 0.021 | 0.023 | 0.055 |
| 12AM | 0.0017 | 0.033 | 0.058 | 0.023 | 0.033 | 0.058 |
| 9PM | 0.00068 | 0.025 | 0.049 | 0.023 | 0.025 | 0.049 |

## References

[1] T. Dierks and C. Allen, “The TLS Protocol Version 1.0,” Internet Engineering Task Force, RFC 2246, Jan. 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2246.txt

[2] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.1,” Internet Engineering Task Force, RFC 4346, Apr. 2006. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4346.txt

[3] ——, “The Transport Layer Security (TLS) Protocol Version 1.2,” Internet Engineering Task Force, RFC 5246, Aug. 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5246.txt

[4] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[5] M. Fischlin and F. Günther, “Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates,” in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017, pp. 60–75.

[6] M. Cebe and K. Akkaya, “A replay attack-resistant 0-rtt key management scheme for low-bandwidth smart grid communications,” in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[7] P. Li, J. Su, and X. Wang, “Itls/idtls: Lightweight end-to-end security protocol for iot through minimal latency,” in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 166–168. [Online]. Available: https://doi.org/10.1145/3342280.3342347

[8] N. Aviram, K. Gellert, and T. Jager, “Session resumption protocols and efficient forward security for tls 1.3 0-rtt,” in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 117–150.

[9] X. Cao, S. Zhao, and Y. Zhang, “0-rtt attack and defense of quic protocol,” in *2019 IEEE Globecom Workshops (GC Wkshps)*, 2019, pp. 1–6.

[10] S. Hohenberger, V. Koppula, and B. Waters, “Adaptively secure puncturable pseudorandom functions in the standard model,” in *Advances in Cryptology – ASIACRYPT 2015*, T. Iwata and J. H. Cheon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 79–102.