# Generating Just-in-Time Shared Keys (JIT-SK) for TLS 1.3 Zero RoundTrip Time (0-RTT)

Eslam G. AbdAllah[1], Yu Rang Kuang[2], and Changcheng Huang[1]
[1]*Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada*
[2]*Quantropi Inc., Ottawa, ON, Canada*
[1]{eslamabdallah, huang}@sce.carleton.ca [2]{randy.kuang}@quantropi.com

***Abstract*—The main goal of Transport Layer Security (TLS) protocol is to provide a secure communication channel between communicating pairs. A new version of the protocol, TLS 1.3, is introduced to improve security and performance for customers. One of the major advantages of TLS 1.3 over earlier versions is that it introduces Zero RoundTrip Time (0-RTT) feature, that saves a round trip at connection setup stage. 0-RTT data security properties are weaker than other kinds of TLS data because the data is not forward secret and it is vulnerable to replay attacks. Existing solutions such as single-use tickets, client hello recording, and freshness checks provide inefficient solutions for 0-RTT problems.**

**In this paper, we propose an efficient technique to utilize 0-RTT feature with forward secrecy and prevent replay attacks. Our technique uses a synchronized pseudorandom number generator (PRNG) that depends on initial shared secret between communicating pairs. The initial secret can be shared using TLS 1.3 three basic key exchange modes. In our technique, the PRNG also uses session shared information such as session ID to dynamically provide Just-in-Time Shared Keys (JIT-SK) for 0-RTT. Client and server sides change the keys in each session and hence securely and efficiently use the 0-RTT. We implement a proof of concept for our technique using our private PRNG, named Quantum Entropy Expansion and Propagation (QEEP), and WolfSSL implementation for TLS 1.3 and show the differences using our solution.**

***Index Terms*—Transport Layer Security (TLS v1.3), Zero RoundTrip Time (0-RTT), pseudorandom number generator (PRNG), Quantum Entropy Expansion and Propagation (QEEP)**

## I. Introduction

The Transport Layer Security (TLS) protocol consists of two primary protocols: a handshake protocol and a record protocol. The handshake protocol main goals are to authenticate the communicating pairs, negotiate cryptographic parameters, and establish shared keys. A record protocol protects the traffic between the communicating pairs using the handshake parameters. The Internet Engineering Task Force (IETF) introduced a new version of Transport Layer Security (TLS) protocol, TLS 1.3, in August 2018 [1]. TLS 1.3 includes a lot of security and performance improvements that can be summarized in the following points: (1) TLS 1.3 introduces Zero RoundTrip Time (0-RTT) feature, (2) current symmetric encryption algorithms in TLS 1.3 are Authenticated Encryption with Associated Data (AEAD) algorithms, (3) all public-key based key exchange mechanisms support forward secrecy, and (4) other updates including adding or removing cryptographic algorithms and modifying some functions in the protocol [1] - [6].

One of the biggest advantages of TLS 1.3 is the 0-RTT. Based on Cloudflare [2], the connections can be classified to two different groups. Around 60% of the connections are from people who are visiting a website for the first time or revisiting after an extended period of time. For this group, TLS 1.3 improves the speed of these connections significantly. Around 40% of the connections are from people who are resuming a previous connection. For this group, TLS 1.3 uses 0-RTT. 0-RTT speeds up resumed connections, and hence leads to a faster response for web sites that people visit regularly.

0-RTT feature was added in TLS 1.3 to save a round trip at connection setup for some application data, however it comes with some security vulnerabilities. 0-RTT data is not forward secret, because the data is encrypted under keys derived using the offered Preshared keys (PSK). By using 0-RTT, there are no guarantees of non-replay between connections. In ordinary TLS 1.3 1-RTT data, the date is protected using servers's random value, whereas 0-RTT data does not depend on the ServerHello random value and hence has weaker guarantee. Additionally, more attacks can be performed based on replay attacks such as reordering of 0-RTT messages and cache timing attacks. As 0-RTT is a new feature in TLS 1.3, it is noncompatible with earlier TLS versions, which also can cause security risks. To overcome 0-RTT security vulnerabilities, TLS 1.3 uses different mitigation techniques such as single-use tickets, client hello recording, and freshness checks, however these techniques provide inefficient solutions for 0-RTT problems.

In this paper, we propose to use a synchronized pseudo-random number generator (PRNG) that can provide Just-in-Time Shared Keys (JIT-SK) for 0-RTT. Client and server sides dynamically change the keys each session. The PRNG depends on initial shared secret key that can be provided using TLS 1.3 key exchange modes in addition to session shared information such as session ID. Then the PRNG generates random keys for each session to secure 0-RTT messages. Using fresh random key with each session provides forward secrecy for 0-RTT data and also prevents replay attacks as the key cannot be reused for multiple sessions. We implement our technique using a private PRNG, named Quantum Entropy Expansion and Propagation (QEEP) [4] and WolfSSL libraries for TLS 1.3. We show the differences between WolfSSL basic implementation that depends on static keys and our approach that provides JIT-SK for 0-RTT.

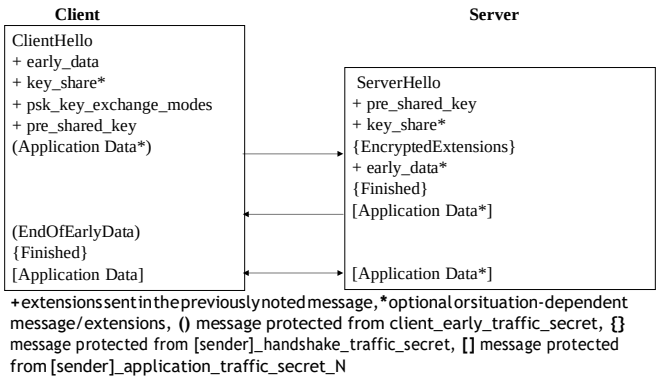


Fig. 1: 0-RTT handshake [1]

## II. Problem Statement and Existing Countermeasures

In TLS 1.3, clients and servers share a PSK, which can be obtained externally or using a previous handshake. TLS 1.3 allows clients to send 0-RTT data on the first flight (early data), as shown in Figure 1. The client uses the PSK to authenticate the server and to encrypt the early data. 0-RTT messages have the same types as other messages in handshake and application date, but 0-RTT messages are encrypted using 0-RTT keys.

The security attributes for 0-RTT data are weaker than other kinds of TLS data [1] - [3]. The main reasons for this weakness can be summarized in the following points. 0-RTT data is not forward secret. Forward security can be defined as the security of past communications even when a certain message is compromised at a later stage. 0-RTT is encrypted using keys derived using the offered PSK, and hence it is not forward secret.

Replay attacks can be easily performed by network attackers by duplicating a flight of 0-RTT data. Additionally, network attackers can use client retry behavior to send multiple copies of an application message. 0-RTT increases this attack especially for any system that does not maintain consistent server state. For example, if a server has multiple zones, in which tickets from zone 1 will not be accepted in zone 2, then the attacker can duplicate 0-RTT for multiple zones. This enforces the server to reject lots of 0-RTT messages and clients need to complete full handshake, which makes 0-RTT feature useless. Performing replay attacks can also cause more potential attacks. The duplication of actions causes side effects (e.g., buying an item or sending money) to be duplicated, thus harming the communicating pairs. Attackers can also reorder 0-RTT messages. Attackers can perform cache timing attacks in order to discover the content of 0-RTT messages. This can be performed by redirecting a 0-RTT message to malicious cache nodes and figure out the requested resource.

Another issue in 0-RTT data is coming from the non compatibility with older servers. This can cause a downgrade attack especially in multi-server environments, where some servers implement TLS 1.3 and others implement TLS 1.2 or earlier versions. A countermeasure for this issue in the mutli-server environments is to ensure a deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

Existing solutions to overcome 0-RTT security challenges can be summarized in the following three techniques. These techniques provide partial solutions and limit the capabilities of 0-RTT feature [1].

**Single-use tickets.** This technique proposes anti-replay defense mechanism that allows each session ticket to be used once. The server can have a database of all valid tickets and deletes each ticket once it is used. When unregistered ticket is provided, the server forces the connection to use the full handshake. This technique requires the session database to be shared between server nodes in environments with multiple distributed servers, and hence it is hard to achieve high rates of successful PSK 0-RTT connections.

**Client hello recording.** In this technique, the anti-replay mechanism is to record a unique value derived from the ClientHello message and reject duplicates. It is infeasible to record all ClientHellos, but a server can record ClientHellos within a given time window by using "obfuscated ticket age" attribute to make sure that tickets cannot be reused outside that window. This technique is suitable to be implemented in distributed systems with high rates of resumption and 0-RTT, because client hello recording mechanism does not require storing all outstanding tickets. In these distributed systems, it is impractical to have large scale reliable and consistent storage of all the received ClientHellos. In this case, the best anti-replay strategy for this mechanism is by using a single storage zone to be responsible for granting or refusing 0-RTT for a ticket in any zone. This simply prevents replay attacks because there is only one zone that will accept or reject 0-RTT data. Another design approach is to implement a storage for each zone, and hence limits the number of replays to once per zone.

**Freshness checks.** ClientHello message includes the time at which the client sent the message. The freshness check technique can use this time to efficiently determine whether a ClientHello was sent reasonably recently or not. This technique only accept 0-RTT for such a recent ClientHello message, otherwise return back to a 1-RTT handshake. For the implementation of this mechanism, a server has to store the time of the generated session ticket and an estimation of the roundtrip time between client and server. This freshness checking is not sufficient to prevent replay attacks because of the following reasons: (1) it does not detect replay attacks during the error window, and (2) it is only performed at the time the ClientHello is received not when continuous early data records are received.

Generally, TLS 1.3 servers are responsible to protect themselves against 0-RTT data replication attacks. The existing countermeasures try to prevent replay at the TLS layer, however they do not provide complete protection against receiving multiple copies of client data. Application protocols need a profile that defines 0-RTT usage. In this profile, the application protocols identify the allowed messages to be used with 0-RTT and what should happen when the server rejects 0-RTT.

## III. PROPOSED SOLUTION (JIT-SK FOR 0-RTT)

In order to access a web page using HTTPS for the first time, there are four basic steps.

**DNS lookup.** DNS is operated by Internet Service Provider (ISP), who caches the IP address for popular domains. The latency in this step is negligible.

**TCP handshake.** This step takes one roundtrip. In this step, the data is sent from a client to server and back using SYN and ACK packets, respectively.

**TLS handshake.** This step requires two roundtrips. The client and server agree on the cryptographic key and establish an encrypted connection.

**HTTP.** This step requires one roundtrip. After establishing the secure connection in TLS handshake step, the browser sends GET request as encrypted HTTP request and the server replies with HTTP response.

In general, there are four roundtrips before the client can access the website. In case the client revisits the site again (resumed connections), the total required roundtrips become three because one of TLS handshakes can be removed using TLS session resumption. A big advantage for TLS 1.3 is that it requires only one roundtrip in TLS handshake for new or revisited website so the total number in each case is just three roundtrips. With TLS 0-RTT another roundtrip can be eliminated for resumed connections. Table I summarizes the required roundtrips in TLS 1.2, TLS 1.3, and TLS1.3 0-RTT.

TABLE I: Required roundtrips for different TLS versions

| TLS version | New Connection | Resumed Connection |
|---|---|---|
| TLS 1.2 (and earlier) | 4 | 3 |
| TLS 1.3 | 3 | 3 |
| TLS 1.3 with 0-RTT | 3 | 2 |

TLS 1.3 supports three basic key exchange modes: Diffie-Hellman over either finite fields or elliptic curves abbreviated as (EC)DHE, PSK-only, and PSK with (EC)DHE. Our solution does not require any modification to the standard TLS 1.3. Instead of using the preshared key for 0-RTT, we use the preshared key as initial secret key (seed) for a synchronized PRNG, that can generate a large number of random keys. A random key can be generated for each session to provide JIT-SK. The client can use the random key to encrypt the first flight (early data). Instead of using PSK just once for 0-RTT as defined in TLS 1.3, we use random keys for each session to extend 0-RTT usage multiple times. Using PRNG, a client and server need to have preshared secret for authentication to avoid man-in-the-middle attack. This secret is needed only once in our approach. The preshared key from TLS 1.3 can be used as our seed. The client encrypts the early data using the generated random number (session key) and securely transmits the encrypted data. The receiver uses a synchronized algorithm to generate the same random number (session key) to decrypt the received message.

In this paper as a PRNG, we use QEEP, which is a software-based technique that depends on unitary permutation matrices. QEEP uses a large key space, which enables QEEP to generate random outputs that is indistinguishable from true randomness. In QEEP, each row in the matrix consists of a permutation of all numbers between 0 to 255. There are total 256! unique states for an 8-bit system and Shannon's entropy is 1684 bits. Figure 2 shows the steps of our technique. We use PSK and session ID to randomly select the permutation matrix and then use PSK to generate the session key (SK) that can be used to encrypt/decrypt the early data. Generally, any good high entropy PRNG such as QEEP, or SHA-256 [7], or a certified private PRNG can be used. The PRNG should have entropy of 256 bits or more to prevent advanced cryptographic attacks coming from classical or quantum computers [8]-[9]. The quality of the generated randomness is tested using various tools such as National Institute of Standards and Technology Statistical Test Suite (NIST STS), specifically SP 800-22 [10], Dieharder [11], and entropy and randomness online tester [12]. QEEP passes all NIST STS SP 800-22 and Dieharder tests and generates random output that follows a Gaussian normal distribution.

For the server to accept TLS 1.3 0-RTT early data, the server first has to accept a PSK cipher suite and select the first available key in the "pre_shared_key" extension. Second, the server has to verify that TLS version number, selected cipher suite, and selected Application-Layer Protocol Negotiation (ALPN) protocol are the same as the selected PSK. In case of PSK exchanged using NewSessionTicket message, the server verifies the values that are negotiated in the connection when the ticket was established. In case of externally established PSKs, the server verifies the values that are provisioned with the key. When a client uses a PSK and early data is allowed for that PSK, this enables the client to send application data in the first flight of messages. In this case, the client provides "pre_shared_key" and "early_data" extensions. The client uses "EarlyDataIndication" value for "extension_data" field of this extension. Our proposed solution does not change TLS 1.3 standard specifications because the PSK and a session shared information (session ID) will be used as inputs to the PRNG and then the server decrypts using the generated random. If the generated randoms between the server and the client, the server forces the communication to return to 1-RTT handshake.

Using synchronized PRNG can achieve the following goals:

**Forward security.** When a communication is compromised at any stage, forward security property guarantees the security of past communications. Imagine that a communication is exposed, using our technique that depends on dynamic random keys, the attacker still cannot infer any information from previous sessions.

**Prevention of replay attacks.** In our proposed solution, the early and application data for 0-RTT are encrypted with JIT-SK for each session. An attacker cannot replay any message as the key is used only once. This simplifies the detection of replay attacks for communicating pairs. The prevention of replay attacks removes the side effects of these attacks such as duplication of actions, reordering of 0-RTT messages, and cache timing attacks.

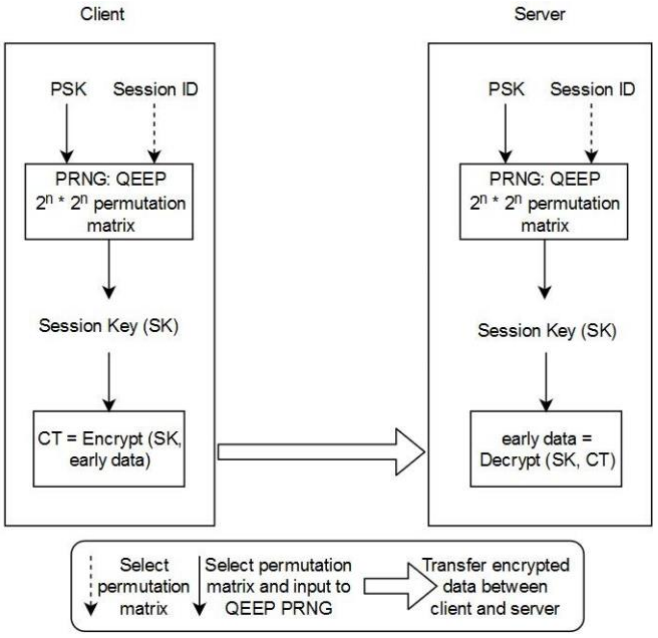


Fig. 2: JIT-SK steps for 0-RTT

**Reusing 0-RTT feature.** Using our technique allows TLS 1.3 to continuously use 0-RTT for multiple sessions. Each session has a fresh shared random number.

## IV. Implementation

TLS 1.3 has been supported and implemented across multiple browsers such as Google Chrome v67+, Mozilla Firefox v61+, and Apple Mac OS v10.3 and iOS v11 [2]. A number of service providers such as Cloudflare, Akamai, and Facebook support TLS 1.3 connections. Also, TLS 1.3 has been implemented using software libraries such as OpenSSL and WolfSSL.

As a proof of concept, we use WolfSSL open source libraries for TLS 1.3. WolfSSL provides static shared key between a server and a client as shown in Figure 3 and Figure 4. The two tests in Figure 5 show that the client and server share and use the same key every time. In order to use PSK in WolfSSL, the following defines should be added to the header file in *IDE WIN user settings.h*

```
#define WOLFSSL_TLS13
#define HAVE_TLS_EXTENSIONS
#define HAVE_SUPPORTED_CURVES
#define HAVE_SESSION_TICKET
#define HAVE_HKDF
#define HAVE_FFDHE_2048
#define WC_RSA_PSS
```

We build with a –enable-psk configure option to remove *#define NO_PSK* from *options.h* as in the following command.

```
$./configure --enable-tls13
--enable-aesccm --enable-keygen
--enable-psk && make
```

Examples to run WolfSSL server and client are shown in the following commands (option -s is used for preshared keys)

```
static WC_INLINE unsigned int my_psk_server_tls13_cb(WOLFSSL* ssl,
    const char* identity, unsigned char* key, unsigned int key_max_len,
    const char** ciphersuite)
{
  int i;
  int b = 0x01;

  (void)ssl;
  (void)key_max_len;

  /* see internal.h MAX_PSK_ID_LEN for PSK identity limit */
  if (strncmp(identity, kIdentityStr, strlen(kIdentityStr)) != 0)
    return 0;

  for (i = 0; i < 32; i++, b += 0x22) {
    if (b >= 0x100)
      b = 0x01;
    key[i] = b;
  }

  *ciphersuite = "TLS13-AES128-GCM-SHA256";

  return 32;  /* length of key in octets or 0 for error */
}
```

Update

```
FILE *fp;
fp = fopen("ServerKey.bin", "rb");
for (i = 0; i < 32; i++){
key[i] = fgetc(fp);}
```

Fig. 3: Server side JIT-SK using WolfSSL implementation

```
static WC_INLINE unsigned int MY_PSK_CLIENT_TLS13_cb(WOLFSSL* ssl,
    const char* hint, char* identity, unsigned int id_max_len,
    unsigned char* key, unsigned int key_max_len, const char** ciphersuite)
{
  int i;
  int b = 0x01;

  (void)ssl;
  (void)hint;
  (void)key_max_len;

  /* see internal.h MAX_PSK_ID_LEN for PSK identity limit */
  strncpy(identity, kIdentityStr, id_max_len);

  for (i = 0; i < 32; i++, b += 0x22) {
    if (b >= 0x100)
      b = 0x01;
    key[i] = b;
  }

  *ciphersuite = "TLS13-AES128-GCM-SHA256";

  return 32;  /* length of key in octets or 0 for error */
}
```

Update

```
FILE *fp;
fp = fopen("ClientKey.bin", "rb");
for (i = 0; i < 32; i++){
key[i] = fgetc(fp);}
```

Fig. 4: Client side JIT-SK using WolfSSL implementation

```
$./examples/server/server -s -v 4 -l
TLS13-AES128-GCM-SHA256
$./examples/client/client -s -v 4 -l
TLS13-AES128-GCM-SHA256
```

In TLS 1.3 the cipher suite parameter can be one of the following five group of algorithms

```
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_GCM_SHA256
TLS_AES_128_CCM_8_SHA256
TLS_AES_128_CCM_SHA256
```

Replay attacks and side effects are applicable in current TLS 1.3 implementation and there is no forward secrecy. As shown in our update in Figure 3 and Figure 4, we dynamically replace these static keys with JIT-SK for 0-RTT by using QEEP. We read the dynamic key from a file each session. This file can be generated offline and be ready for use next session. The client and server can do that using preshared key and session shared information such as session ID. Then QEEP will generate random session keys for securing early data in 0-RTT manner as shown in Figure 6 and Figure 7. These figures show that for each message the client and server use the same JIT-SK.

Algorithm 1 shows the steps for generating JIT-SK. Line 1 selects the permutation matrix using PSK and session ID. The session ID can be replaced with any session shared information. Line 2 generates the session key using PSK. Lines 3 and 4 show the encryption and decryption processes for the

**Test 1 (server and client sides)**

```
Tls 1.3 Server Shared Key at time Wed May  1 09:52:43 2019
123456789abcdef123456789abcdef123456789abcdef123456789abcdef
Tls 1.3 Client Shared Key at time: Wed May  1 09:52:43 2019
123456789abcdef123456789abcdef123456789abcdef123456789abcdef
```

**Test 2 (server and client sides)**

```
Tls 1.3 Server Shared Key at time Wed May  1 09:58:37 2019
123456789abcdef123456789abcdef123456789abcdef123456789abcdef
Tls 1.3 Client Shared Key at time: Wed May  1 09:58:37 2019
123456789abcdef123456789abcdef123456789abcdef123456789abcdef
```

Fig. 5: Static keys at server and client sides

**Test 1 (server side)**

```
Tls 1.3 Server Shared Key at time Tue Jun 18 16:01:06 2019
8a359dec43381b4faa159ca8596721b0e4bf585c15ed409ef7682e2ca241198c 1
peer has no cert!
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_128_GCM_SHA256
SSL curve name is SECP256R1
Server Random : 51867485B69DC40A12ACFDA4042E0288F9693D9586FFEB21A61AAE36B29CC3D5
Client message: hello Quantropi!
```

**Test 2 (server side)**

```
Tls 1.3 Server Shared Key at time Tue Jun 18 16:21:10 2019
d31c23adc5c1603b67f13dfdd71d6674d1216ef1199444fa8a22cdfd29483 2
peer has no cert!
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_128_GCM_SHA256
SSL curve name is SECP256R1
Server Random : A695B63CC7CB956B259557D3CC5F0193298E3A8804BE0AD6173E7DE6B70826BE
Client message: hello Quantropi!
```

**Test 3 (server side)**

```
Tls 1.3 Server Shared Key at time Tue Jun 18 16:21:21 2019
36c6ed19c856bd16eece5686d4d6d7e6fa8d6f9174feebdd2e7b5a34081fcf1 3
peer has no cert!
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_128_GCM_SHA256
SSL curve name is SECP256R1
Server Random : 414302356B0AFC14E654302EF560E9B72818B2CA3AAF07D10E0F74F4BED2BE67
Client message: hello Quantropi!
```

Fig. 6: JIT-SK for 0-RTT data (server side)

early data at the transmitter and receiver sides. Lines 1-4 are repeated for each session.

**Algorithm 1** JIT-SK generation for 0-RTT for each session

**Input**: PSK, Session ID

**Output**: Encrypted/decrypted early data

1: Select permutation matrix (PSK, Session ID)
2: Session key = QEEP (PSK)
3: CT = Enc (Session_key, early_data) *D* for transmitter
4: *D* for receiver: early_data = Dec (Session_key, CT)

## V. Conclusion

Transport Layer Security (TLS) provides a secure channel between two communicating peers that provides the following properties: authentication, confidentiality, and integrity. The Internet Engineering Task Force (IETF) introduced TLS 1.3, which is a big step forward for Internet performance and security. In this paper, we focus on one of the new concepts in TLS 1.3, which is Zero RoundTrip Time (0-RTT) feature. By using TLS 1.3 with 0-RTT, the performance gains are more dramatic. 0-RTT data has security challenges and the existing solutions cannot efficiently mitigate these challenges. In this paper, we use a novel approach that depends on a synchronized

**Test 1 (client side)**

```
Tls 1.3 Client Shared Key at time: Tue Jun 18 16:01:06 2019
8a359dec43381b4faa159ca8596721b0e4bf585c15ed409ef7682e2ca241198c 1
peer has no cert!
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_128_GCM_SHA256
Client Random : E782A5433A3BA5B1E22968879FB883057054BBBBDE460CD633783E20BF8E548B
I hear you Quantropi!
```

**Test 2 (client side)**

```
Tls 1.3 Client Shared Key at time: Tue Jun 18 16:21:10 2019
d31c23adc5c1603b67f13dfdd71d6674d1216ef1199444fa8a22cdfd29483 2
peer has no cert!
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_128_GCM_SHA256
Client Random : EF9780301944BAFD142A284FD95CCDF7960DC27EC3FEDA074F8BEB9780D45F4C
I hear you Quantropi!
```

**Test 3 (client side)**

```
Tls 1.3 Client Shared Key at time: Tue Jun 18 16:21:21 2019
36c6ed19c856bd16eece5686d4d6d7e6fa8d6f9174feebdd2e7b5a34081fcf1 3
peer has no cert!
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_128_GCM_SHA256
Client Random : A3EC0A9FEB702E511B415E3BE5919365C285791BB5DCD076107D1DAA11E3E9E7
I hear you Quantropi!
```

Fig. 7: JIT-SK for 0-RTT data (client side)

pseudorandom number generator (PRNG) to efficiently and securely use 0-RTT feature. We implement our approach using a private PRNG, named Quantum Entropy Expansion and Propagation (QEEP) and WolfSSL libraries for TLS 1.3. The implementation shows that using our approach enables 0-RTT with forward secrecy and without replay attacks.

## Acknowledgments

This research is partially funded by Mitacs Canada.

## References

[1] E. Rescorla, “The transport layer security (TLS) Protocol Version 1.3”, Internet Engineering Task Force, August 2018, Online at https://www.rfc-editor.org/rfc/pdfrfc/rfc8446.txt.pdf, last visited on Jan. 20, 2020.

[2] P. Nohe, “TLS 1.3: Everything you need to know”, 2019, Online at https://www.thesslstore.com/blog/tls-1-3-everything-possibly-needed-know/, last visited on Jan. 20, 2020.

[3] N. Naziridis, “TLS 1.3 is here to stay”, SSL, October 2018, Online at https://www.ssl.com/article/tls-1-3-is-here-to-stay/, last visited on Jan. 20, 2020.

[4] “Lightweight low-latency quantum secure”, Quantropi Inc., Online at https://quantropi.com/#/home, last visited on Jan. 20, 2020.

[5] N. Sullivan, “Introducing zero round trip time resumption (0-RTT)”, Cloudflare, 2017, Online at https://blog.cloudflare.com/ introducing-0-rtt/, 2017, last visited on Jan. 20, 2020.

[6] B. Dowling, M. Fischlin, F. G ünther, and D. Stebila, “A cryptographic analysis of the TLS 1.3 handshake protocol candidates”, In ACM Conference on Computer and Communications Security (ACM CCS), ACM, Denver, CO, USA, pp. 1197–1210, October 2015.

[7] S. Łoza, Ł. Matuszewski, and M. Jessa, “A Random number generator using ring oscillators and SHA-256 as post-processing”, Intl Journal of Electronics and Telecommunications, vol. 61, no. 2, pp. 199–204, June 2015.

[8] F. Arute et al., “Quantum supremacy using a programmable superconducting processor, Nature, vol. 574, pp. 505-510, 2019.

[9] P.W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”, SIAM Journal on Computing, vol. 26, no. 5, pp. 1484-1509, 1997.

[10] M. Sýs, Z. Říha, and V. Matyáš, “Algorithm 970: optimizing the NIST statistical test suite and the berlekamp-massey algorithm”, *ACM Transactions on Mathematical Software*, vol. 43, no. 3, pp. 27-37, 2017.

[11] R. G. Brown, “Dieharder: a random number test suite”, Online at https://webhome.phy.duke.edu/~rgb/General/dieharder.php, last visited on Jan. 20, 2020.

[12] “Entropy and randomness online tester”, Online at https://servertest.online/entropy, 2019, last visited on Jan. 20, 2020.