A preliminary version of this paper appears in the *24th European Symposium on Research in Computer Security, ESORICS 2019*. This is the full version.

Secure Communication Channel Establishment: TLS 1.3 (over TCP Fast Open) vs. QUIC

Shan Chen[1], Samuel Jero[2], Matthew Jagielski[3], Alexandra Boldyreva[1], and Cristina Nita-Rotaru[3]

[1]Georgia Institute of Technology
shanchen@gatech.edu sasha@gatech.edu
[2]Purdue University
sjero@sjero.net
[3]Northeastern University
jagielski.m@husky.neu.edu c.nitarotaru@neu.edu

Abstract

Secure channel establishment protocols such as Transport Layer Security (TLS) are some of the most important cryptographic protocols, enabling the encryption of Internet traffic. Reducing the latency (the number of interactions between parties) in such protocols has become an important design goal to improve user experience. The most important protocols addressing this goal are the just-released TLS 1.3, which is likely to see deployment in the near future, and Quick UDP Internet Connections (QUIC), a secure transport protocol from Google that is available in the Chrome browser. There have been a number of formal security analyses for TLS 1.3 and QUIC, but their security, when layered with their underlying transport protocols, cannot be easily compared. Our work is the first to thoroughly compare the security and availability properties of these protocols. Towards this goal, we develop novel security models that permit "layered" security analysis. In addition to the standard goals of server authentication and data privacy and integrity, we consider the goals of IP spoofing prevention, key exchange packet integrity, secure channel header integrity, and reset authentication, which capture a range of practical threats not usually taken into account by existing security models that focus mainly on the cryptographic cores of the protocols. Equipped with our new models we provide a detailed comparison of TLS 1.3 over TCP Fast Open (TFO), QUIC over UDP, and QUIC[TLS] (a new design for QUIC that uses TLS 1.3 key exchange) over UDP. In particular, we show that TFO's cookie mechanism does provably achieve the security goal of IP spoofing prevention. Additionally, we find several new availability attacks that manipulate the early key exchange packets without being detected by the communicating parties. By including packet-level attacks in our analysis, our results shed light on how the reliability, flow control, and congestion control of the above layered protocols compare, in adversarial settings. We hope that our results will help protocol designers in their future protocol analyses and that our results will help practitioners better understand the advantages and limitations of novel secure channel establishment protocols.

Keywords: applied cryptography, provable security, TLS, QUIC, secure channel, availability, network protocols

1 Introduction

Motivation. Nowadays, more than half of all Internet traffic is encrypted according to a 2017 EFF report [25], with Google reporting that 94% of its traffic is encrypted as of October 2019 [1]. This trend has also been facilitated by efforts like the free digital certificate issuer Let's Encrypt servicing 87 million active (unexpired) certificates and 150 million unique domains at the end of 2018 [2].

This widespread Internet traffic encryption is enabled by protocols that allow two parties (where one or both parties have a public key certificate) to establish a secure communication channel over the insecure Internet. Typically, the parties first authenticate all parties holding a public key certificate and agree on a session key — the key exchange phase. Then, this session key is used to encrypt the communication during the session — the secure channel phase. We will refer to such protocols as secure channel establishment protocols.

The main secure channel establishment protocol in use today is TLS. The session key establishment of the widely deployed standard TLS 1.2 involves 3 round-trip times (RTTs) of end-to-end communication for a full connection and 2 RTTs for resumption (that saves 1 RTT), including the cost of establishing a TCP connection before the TLS connection. Further, this TCP cost is paid every time the two parties communicate with each other, even if the connection is interrupted and then immediately resumed. Given that most encrypted traffic is web traffic, this cost represents a significant performance bottleneck, a nuisance to users, and financial loss to companies. For instance, back in 2006 Amazon found that every 100ms of latency cost them 1% in sales [41], while a typical RTT on a connection from New York to London is 70ms [60].

Not surprisingly, many efforts in recent years have focused on reducing latency in secure channel establishment protocols. The focus has been on reducing the number of interactions (or RTTs) during session establishment and resumption without sacrificing much security. The most important protocols addressing this goal are TLS 1.3 [53] (the just-released successor to the current TLS 1.2 standard) and Google's QUIC [55].

With TLS 1.3, it is possible to reduce the number of RTTs (prior to sending encrypted data) during session resumption to 1. This reduction is achieved by utilizing a session ticket that was saved during a previous communication and multiple keys (which we call stage keys) that can be set within one session, of which some keys are set faster (with slightly less security) so that data can be encrypted earlier. The remaining 1-RTT during session resumption is due to the aforementioned TCP connection. However, one recent optimization for TCP, called TCP Fast Open (TFO) [52, 15] extends TCP to allow for 0-RTT resumption connections, so that the client may begin data transmission immediately. The mechanism underlying this optimization is a cookie saved from previous communication, similar to the ticket used by TLS 1.3.

Like TLS 1.3, Google's QUIC uses weaker initial keys, under which data can be encrypted earlier, and a token saved from previous communication between the parties. But unlike TLS, QUIC operates over UDP rather than TCP. Instead of relying on TCP for reliability, flow control, and congestion control, QUIC implements its own data transmission functionality, integrating connection establishment with key exchange. These features allow QUIC to have 1-RTT full connections and 0-RTT resumption connections.

Table 1: Latency comparison of layered protocols

Layered Protocol	Full Connection	Resumption Connection
TCP+TLS 1.2	3-RTT	2-RTT
TCP+TLS 1.3	2-RTT	1-RTT
TFO+TLS 1.3	2-RTT	**0-RTT**
UDP+QUIC	1-RTT	**0-RTT**
UDP+QUIC[TLS]	1-RTT	**0-RTT**

In Table 1 we show the cost of establishing full and resumption connections for several layered protocol options achieving end-to-end security. These include TLS 1.2 over TCP, TLS 1.3 over TCP, TLS 1.3 over TFO, QUIC over UDP, and the new design for QUIC [28] (which we refer to as QUIC[TLS] [59] to indicate that it borrows the key exchange from TLS 1.3) over UDP. It is clear that the last three win in terms of the number of interactions. *But how does their security compare?*

Related Work. At first glance, the question is easy to answer. Recent works have done formal security analyses of TLS 1.3 [35, 8, 19, 16, 36, 20, 40, 22, 18, 7, 17, 11] and Google's QUIC [24, 42]. Most works confirm that (the cryptographic cores of) both protocols are provably secure under reasonable computational assumptions. Moreover, as shown in [42, 22], their 0-RTT data transmission designs cannot achieve the same strong security guaranteed by classical key exchange protocols with at least one RTT. In particular, the 0-RTT keys do not provide forward secrecy and the 0-RTT data suffers from replay attacks. Overall, it might seem that all three layered protocols mentioned above are equally secure.

However, a closer look reveals that the answer is not that simple. First, all aforementioned formal security analyses, except for [42] analyzing the IP spoofing (source validation) of QUIC, did not consider packet-level availability attacks, i.e., those targeting availability properties like reliable delivery (via packets) of messages. Therefore, it is not clear at the packet level what security can be achieved and what attacks can be prevented by these protocols. In other words, we have no formal understanding of what security can be obtained when layering protocols. Also, TFO uses some cryptographic primitives, such as a cookie, to prevent IP spoofing, but, to the best of our knowledge, no formal analysis has been done. Furthermore, the security of QUIC[TLS] has not been formally analyzed (although some security aspects can be reduced to those of Google's QUIC and TLS 1.3).

Our Contributions. To compare security, we first need to define a general protocol syntax for secure channel establishment and fix a security model for it. Since the only provable security analysis that studies security related to data transmission functionality is [42], we take their *Quick Connections (QC)* protocol definition and *Quick Authenticated and Confidential Channel Establishment (QACCE)* security model as our starting point.

Protocol definition. To accommodate protocol syntaxes of TLS 1.3 and QUIC[TLS], we extend the QC protocol to a more general *Multi-Stage Authenticated and Confidential Channel Establishment (msACCE)* protocol, which allows more keys to be set during each session.

Security definitions. We extend the *Quick Authenticated and Confidential Channel Establishment (QACCE)* security model [42] to two msACCE security models — msACCE-std and msACCE-pauth — that are general enough for all layered secure channel establishment protocols listed in Table 1. The former is fairly standard and is for core cryptographic security notions such as Server Authentication and Channel Security. The latter security model is novel and is for packet-level security.

For the second model that deals with packet-level availability attacks, we first follow QACCE [42] to consider IP-spoofing prevention (also known as address validation) and further extend it to additionally capture IP-spoofing attacks in the full connections. Then, we design several novel notions for packet-level authentication as follows.

First, we define Header Integrity to capture the integrity of the whole unencrypted packet header. (Note that previous models like QACCE only cover the header integrity implied by the authenticity security of the underlying authenticated encryption scheme.) To enable fine-grained security analyses and comparisons, we split the above notion into two related ones, Key Exchange (KE) Header Integrity and Secure Channel (SC) Header Integrity, which capture header integrity during the key exchange phase and secure channel phase respectively. Furthermore, we define the notion of KE Payload Integrity to cover availability attacks that modify the payloads of packets sent during key exchange. We note that unlike the availability attacks shown in [42], successful attacks under our new notions do not affect the client's session key establishment and therefore are harder or impossible to detect by the client. This makes such attacks more harmful and their treatment more important. Finally, we formalize the new goal of Reset Authentication to deal with attacks forging a reset packet to abruptly

terminate an honest party's session.[1]

Security analyses. Equipped with our new models, we study the security and availability functionalities provided by TFO+TLS 1.3, UDP+QUIC, and UDP+QUIC[TLS]. We first confirm that all protocols provably satisfy the standard security notions of Server Authentication and Channel Security given that their building blocks are secure. The results mostly follow from prior works and we just have to argue that they still hold for the extended model. Similarly, prior results showed that QUIC achieves IP-spoofing prevention and we show that this extends to our stronger notion. As for TFO+TLS 1.3, its IP-spoofing prevention relies on TCP sequence number randomization and TFO's cookie mechanism (but no prior former analysis confirmed its security). We prove that TFO+TLS 1.3 does satisfy this security assuming that the underlying block cipher is a pseudorandom function.

Regarding SC Header Integrity, we show that while UDP+QUIC is secure, TFO+TLS 1.3, on the other hand, is insecure because it allows header-only packets to be sent in the secure channel phases and does not authenticate the TCP headers of encrypted packets. This theoretical result captures practical availability attacks that the networking community has been slowly uncovering via manual investigation over the last 30 years [56, 32, 4, 14, 38, 37, 30, 51, 13, 26, 50, 45, 61, 31], such as TCP flow control manipulation, TCP acknowledgment injection, etc.

We next show that neither protocol satisfies KE Header Integrity. For TFO+TLS 1.3 this result leads to a TFO cookie removal attack that we discover, which allows the attacker to undermine the whole benefit of TFO. Then, we show that UDP+QUIC is not secure in the sense of KE Payload Integrity. This leads to a new availability attack that we call ServerReject Triggering. Note that unlike the QUIC attacks (e.g., server config replay attack, connection ID manipulation attack, etc.) discovered in [42], ServerReject Triggering is harder to detect and more harmful in this sense. We show that TFO+TLS 1.3, on the other hand, achieves KE Payload Integrity.

We further show that neither TFO+TLS 1.3 nor UDP+QUIC provide Reset Authentication, justifying the TCP Reset attack [61] relevant for TFO+TLS 1.3 and the PublicReset attack for UDP+QUIC. For completeness, we recall the results from [42, 22] showing that neither protocol provides forward secrecy for the keys encrypting 0-RTT data and that this data can be replayed.

We finally show that the new UDP+QUIC[TLS] protocol achieves the strongest security of the three designs. While formally it does not provide KE Payload Integrity, the related attacks can also happen in TFO+TLS 1.3 in a similar way, while the latter satisfies KE Payload Integrity mainly because its availability functionalities are all carried in its protocol headers rather than payloads. More importantly, UDP+QUIC[TLS] is the only protocol that guarantees Reset Authentication (based on the unpredictability of its reset tokens).

Table 2: Security comparison

	TLS 1.3 +TFO	QUIC +UDP	QUIC[TLS] +UDP
0-RTT Key Forward Secrecy [22]	✗	✗	✗
0-RTT Data Anti-Replay [22]	✗	✗	✗
Server Authentication	✓	✓	✓
Channel Security	✓	✓	✓
IP-Spoofing Prevention	✓	✓	✓
KE Header Integrity	✗	✗	✗
KE Payload Integrity	✓	✗	✗
SC Header Integrity	✗	✓	✓
Reset Authentication	✗	✗	✓

Our results are summarized in Table 2. Even though QUIC may not be able to sustain the com-

[1] Note that reset authentication and *secure termination* [9], though look similar, capture very different security goals: the former guarantees a session can be reset only by the intended party while the latter ensures that upon receipt of a signifying message the receiver has received all messages sent and will ever be sent by the sender.

petition in the long run despite stronger security, we hope our models will help protocol designers and practitioners better understand the important security aspects of novel secure channel establishment protocols.

Paper Organization. The rest is organized as follows. We provide an overview of relevant design information for TFO, TLS 1.3, and QUIC in Section 2. Sections 3 and B specify our notations and preliminaries. Section 4 formally defines our msACCE protocol and its security. Section 5 provides the details of our security analyses and summarizes our findings about how security of the three layered protocols compare. Section 6 concludes our paper.

2 Background

Network protocols are designed and implemented following a layered network stack model where each layer has its own functionality, defines an interface for use by higher layers, and relies only on the properties of lower layers. In this work, we are concerned with three layers: network, represented by the IP protocol; transport, represented by UDP and TCP with the Fast Open optimization (TFO); and application, represented by TLS or QUIC.

2.1 TLS 1.3 over TFO

TCP Fast Open. TCP Fast Open (TFO) is an optimization to the TCP protocol. TCP itself provides the following services to an application (or higher protocol): (1) reliability, (2) ordered delivery, (3) flow control, and (4) congestion control. It is connection-oriented and consists of three phases: connection establishment, data transfer, and connection tear-down. TCP relies on control information from its header to implement this functionality. For example, as shown in Fig. 4 in Appendix A, control bits specify what type of packets are sent over the network, which determines whether the packets are establishing a new connection, sending data, acknowledging data, or tearing down the connection.

The disadvantage of layering protocols is that higher level protocols have no control over the internal mechanics of lower level protocols and can interact with them only through defined interfaces. A protocol using standard TCP for transferring data needs to wait for connection establishment at the TCP layer to complete before it receives notification of a new connection and can begin its own processing and data transfer.

The TFO optimization introduces a simple modification to the TCP connection establishment handshake to reduce the 1-RTT connection establishment latency of TCP and allow for 0-RTT handshakes, so that data transmission may begin immediately. TFO fulfills the same design goals mentioned for TCP above, assuming the connection is established correctly.

The mechanism through which 0-RTT is achieved is a cookie that is obtained by the client the first time it communicates with a server and cached for later uses. This cookie is intended to prevent replay attacks while avoiding the need for servers to keep expensive state. It is generated by the server, authenticates client IP address, and has a limited lifetime. Generation and verification have low overhead.

Cookies are sent in the TFO option field in SYN packets. The first two message exchanges in Fig. 1 (a), show how a cookie is obtained. The client requests a cookie by using the TFO option in the SYN with the cookie field set to 0, indicating that it would like to use TFO. The server generates an appropriate cookie and places it in the TFO option field of the SYN-ACK. The client caches this cookie for subsequent connections to this server. If a cookie was not provided, the client instead caches the negative response, indicating that TFO connections should not be tried to this server, for some time.

In subsequent connections to this server (first message in Fig. 1 (b)), the client places its cached TFO cookie in the TFO option in the SYN packet. The client is also allowed to send 0-RTT data in the remainder of the SYN packet. This might be an HTTP GET request or a TLS `ClientHello`

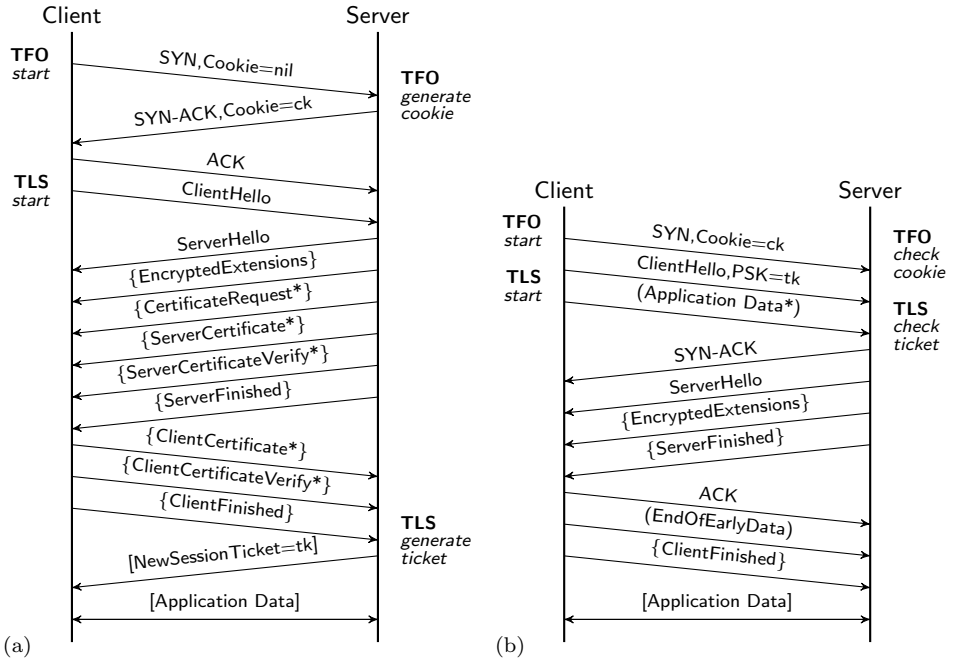

Figure 1: TFO+TLS 1.3 (EC) DHE 2-RTT full handshake (a) and TFO+TLS 1.3 PSK-(EC) DHE 0-RTT resumption handshake (b). * indicates optional messages. () indicates messages protected using the 0-RTT keys derived from a pre-shared key. {} and [] indicate messages protected with initial and final keys.

message. When the server receives the SYN, it will validate the cookie. If the cookie is valid, it responds with a SYN-ACK acknowledging the 0-RTT data and a response to the 0-RTT data. If the cookie is invalid (expired or otherwise), a full handshake is required and any initial data ignored.

TLS 1.3. TLS provides confidentiality, authentication, and integrity of communication over a secure channel between a client and a server. This is accomplished in two phases – the handshake and the record protocol. The handshake sets up appropriate parameters for the record protocol to achieve these three goals. These include parameters like the cipher suite to use and the shared secret key. Unfortunately, the handshake in TLS 1.2 takes 2-RTTs to complete. Additionally, the naive layering of TLS 1.2 over TCP, as traditionally used for HTTPS, would require a full 3-RTTs before the HTTP request could be sent. Fortunately, the recently standardized TLS 1.3 [53] provides many improvements over TLS 1.2. Most relevant for our purposes, it enables 0-RTT handshakes at the TLS level.

In a TLS 1.3 full connection (see Fig. 1 (a), 4th message), the client begins by sending a `ClientHello` message containing a list of ciphersuites the client is willing to use with key shares for each and optional extensions. The server responds with a `ServerHello` message containing the ciphersuite to use and its key share. At this point, an initial encryption key is derived and all future messages are encrypted. The server also sends an `EncryptedExtensions` message containing any extension data, a `CertificateRequest` message if doing client authentication, a `ServerCertificate` message containing the server's certificate, a `ServerCertificateVerify` message containing a signature over the handshake with the private key corresponding to the server's certificate, and a `ServerFinished` message containing an HMAC of all messages in the handshake. The client receives these messages, verifies their contents, and responds with `ClientCertificate` and `ClientCertificateVerify` messages if doing client authentication before finishing with a `ClientFinished` message containing an HMAC of all messages in the handshake. At this point, a final encryption key is derived and used

for encrypting all future messages. If the server supports 0-RTT connections, one final handshake message, the `NewSessionTicket` message, will be sent by the server to provide the client with an opaque session ticket to be used in a resumption session.

In later TLS 1.3 resumption connections to this server, the client uses the session ticket established in the prior full connection to do a 0-RTT connection. In this case, the client sends a `ClientHello` message indicating a pre-shared-key ciphersuite, a ciphersuite to use for the final key, and the cached session ticket. The client can then derive an encryption key and begin sending 0-RTT data. The server will verify the session ticket, use it to establish the same encryption key, and send a `ServerHello` message containing the ciphersuite to use and its final key share. At this point, an initial encryption key is derived and all future messages are encrypted. The server also sends an `EncryptedExtensions` message containing any extension data and a `ServerFinished` message containing an HMAC of all messages in the handshake. The client receives these messages, verifies their contents, and responds with an `EndOfEarlyData` message and a `ClientFinished` message containing an HMAC of all messages in the handshake. At this point, a final encryption key is derived and used for encrypting all future messages.

TLS 1.3 over TFO. TLS assumes that lower layers provide reliable, in-order delivery of TLS messages. As a result, TLS is usually layered on top of TCP, which provides these properties. This usually results in a delay for the TCP handshake followed by a delay for the TLS handshake. This is obviously undesirable. However, the combination of TLS 1.3 and TCP Fast Open enables true 0-RTT connections.

In a full connection to a TFO+TLS 1.3 server, the client requests a TFO cookie in the TCP SYN and then does a full TLS 1.3 handshake once the TCP connection completes. This takes 3-RTTs (see Fig. 1 (a)), but provides a cached TFO cookie and cached TLS session ticket.

In subsequent resumption connections to this server, the client can use the TFO cookie to establish a 0-RTT TCP connection and include the TLS 1.3 `ClientHello` message in the SYN packet. The TLS `ClientHello` message can use the cached TLS session ticket to perform a 0-RTT resumption handshake. Thus, the TCP and TLS 1.3 connections are established at the same time, as shown in Fig. 1 (b).

2.2 QUIC over UDP

UDP. UDP [48] is an extremely simple transport protocol providing unreliable datagram delivery, the ability to multiplex data between multiple applications, and an optional checksum. A UDP sender simply wraps the message to be sent with a UDP header (see Fig. 5 in Appendix A) and the receiver unwraps the message and delivers it to the application, after possibly verifying the checksum. No other processing is performed.

UDP has been typically used for applications where low latency is crucial, like video gaming and real-time streaming video. As a result, it can traverse NAT devices and firewalls that often block unknown or rare protocols.

QUIC. Quick UDP Internet Connections (QUIC) is a transport protocol developed by Google and implemented by Chrome and Google servers since 2013 [55]. It now provides service for the majority of requests by Chrome to Google properties [58]. QUIC's goal was to provide secure communication comparable with TLS while achieving reduced connection setup latency compared to traditional TCP+TLS 1.2. To do so, it provides the following services to applications: (1) reliability, (2) in order delivery, (3) flow control, (4) congestion control, (5) data confidentiality, and (6) data authenticity. For repeated connections to the same server it also provides (7) 0-RTT connections, enabling useful data to be sent in the first round trip. In short, QUIC provides a very similar set of services to TFO+TLS 1.3.

Instead of modifying TCP to enable 0-RTT connection establishment, QUIC replaces TCP entirely, using UDP to provide application multiplexing and enabling it to traverse the widest possible swath of the Internet. QUIC then provides all other guarantees itself.

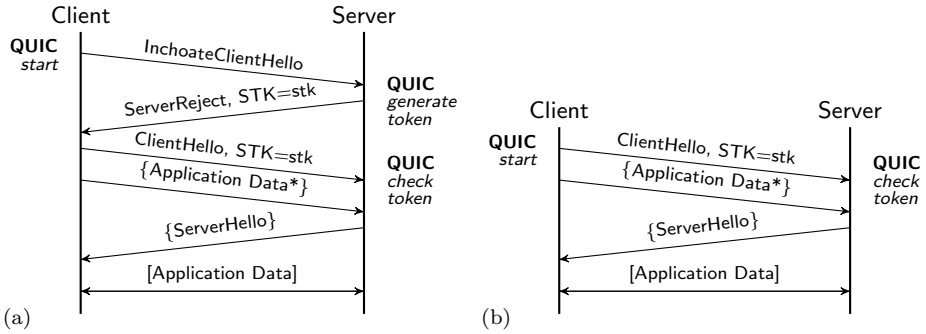

Figure 2: QUIC 1-RTT full handshake (a) and UDP+QUIC 0-RTT resumption handshake (b). * indicates optional messages. {} and [] indicate messages protected with initial and final keys.

QUIC packets contain a public header and a set of frames that are encrypted and authenticated after initial connection setup. The header contains a set of public flags, a unique 64-bit connection identifier referred to as `cid`, and a variable length packet number. All other protocol information is carried in control and stream (data) frames that are encrypted and authenticated.

To provide 0-RTT, QUIC caches important information about the server that will enable the client to determine the encryption key to be used for each new connection. As shown in Fig. 2 (a), the first time a client contacts a given server it has no cached information, so it sends an empty (Inchoate) `ClientHello` message. The server responds with a `ServerReject` message containing the server's certificate and three pieces of information for the client to cache. The first of these is an object called an `scfg`, or server config. The `scfg` contains a variety of information about the server, including a Diffie-Hellman share from the server, supported encryption and signing algorithms, and flow control parameters. This `scfg` has a defined lifetime and is signed by the server's private key to enable authentication using the server's certificate. Along with the `scfg`, the server sends the client a source-address token or `stk`. The `stk` is used to prevent IP spoofing. It contains an encrypted version of the client's IP address and a timestamp.

With this cached information, a client can establish an encrypted connection with the server. It first ensures that the `scfg` is correctly signed by the server's certificate and is valid, then sends a `ClientHello` indicating the `scfg` it is using, the `stk` value it has cached, a Diffie-Hellman share for the client, and a client nonce. After sending the `ClientHello`, the client can create an initial encryption key and send additional encrypted `Application Data` packets. In fact, to take advantage of the 0-RTT connection establishment it must do so. When the server receives the `ClientHello` message, it validates the `stk` and client nonce parameters and creates the same encryption key using the server share from the `scfg` and the client's share from the `ClientHello` message.

At this point, both client and server have established the connection and setup encryption keys and all further communication between the parties is encrypted. However, the connection is not forward secure yet, meaning that compromising the server would compromise all previous communication because the server's Diffie-Hellman share is the same for all connections using the same `scfg`. To provide forward secrecy for all data after the first RTT, the server sends a `ServerHello` message after receiving the client's `ClientHello` which contains a newly generated Diffie-Hellman share. Once the client receives this message, client and server derive and begin using the new forward secure encryption key.

If a client has previously connected to a server, it can instead initiate a resumption connection to the same server. This consists of only the last two steps of a full connection, sending the `ClientHello` and `ServerHello` messages as shown in Fig. 2 (b).

2.3 QUIC with TLS 1.3 Key Exchange

A new version of QUIC [28], which also supports 0-RTT, describes several improvements of the previous design. The most important change is replacing QUIC's key exchange with the one from TLS 1.3, as specified in the latest Internet draft [59]. We provide more details (e.g., the new stateless reset feature) of this new QUIC (denoted by QUIC[TLS]) in Section 5.

3 Preliminaries

Notations. Let $\{0,1\}^*$ denote the set of all finite-length binary strings (including the empty string ε) and $\{0,1\}^n$ denote the set of n-bit binary strings. $[n]$ denotes the set of integers $\{1,2,\ldots,n\}$. For a finite set $\mathcal{R}$, let $|\mathcal{R}|$ denote its size and $r \xleftarrow{\$} \mathcal{R}$ denote sampling r uniformly at random from $\mathcal{R}$. For a binary string s, let $|s|$ denote its length in bits. $y \leftarrow F(x)$ (resp. $y \xleftarrow{\$} F(x)$) denotes y being the output of the deterministic (resp. probabilistic) function F with input x. Let $x \leftarrow a$ denote assigning value a to variable x. We use the wildcard $\cdot$ to indicate any valid input of a function.

Public Key Infrastructure. For simplicity, we do not consider certificates or certificate checks but assume any public key associated with a party is supported by a *public key infrastructure (PKI)* and hence certified and bound to the party's identity. That is, we omit PKI details but simply assume that the binding of public keys to party identities is publicly known by default.

PRF and AEAD. In Appendix B we recall the security definitions of a *pseudorandom function (PRF)* F and a stateful *authenticated encryption with associated data (AEAD)* scheme sAEAD with authentication level $al \in [4]$ (i.e., protecting against the first al types of the following attacks: forgeries, replays, reordering, or dropping) [33]. Accordingly, there we provide the definitions for the corresponding advantages: $\mathbf{Adv}^{\mathsf{prf}}_{F}(A), \mathbf{Adv}^{\mathsf{aead}\text{-}al}_{\mathsf{sAEAD}}(A)$. We also refer to [54] for the syntax and security definitions of a nonce-based AEAD scheme.

4 Multi-Stage Authenticated and Confidential Channel Establishment

In this section, we define the syntax and two security models for *Multi-Stage Authenticated and Confidential Channel Establishment (msACCE)* protocols.

4.1 Protocol Syntax

Our msACCE protocol is an extension to the *Quick Connection (QC)* protocol proposed by Lychev *et al.* [42] and the *Multi-Stage Key Exchange (MSKE)* protocol proposed by Fischlin and Günther [24] (and further developed by [19, 20, 40, 22]). Even though the authors of [42] claimed their QC protocol syntax to be general, TLS 1.3 does not fit it well because TLS 1.3 has two initial keys and one final key in 0-RTT resumption while QC captures only one initial key. On the other hand, the MSKE protocol and its extensions focus only on the key exchange phases.

Our msACCE protocol syntax inherits many parts of the QC protocol syntax but extends it to a multi-stage structure and additionally covers session resumptions (explicitly, unlike QC), session resets, and header-only packets exchanged in secure channel phases. The detailed protocol syntax is defined below.

A msACCE protocol is an interactive protocol between a client and a server. The protocol consists of one or more stages and each stage consists of two phases (that are formally defined later): key exchange and secure channel. The client and server establish keys in key exchange phases and exchange messages encrypted and decrypted with these keys in secure channel phases. Messages are exchanged

via *packets*. A packet consists of source and destination IP addresses[2] $\mathrm{IP_s}, \mathrm{IP_d} \in \{0,1\}^{32} \cup \{0,1\}^{64}$, a header, and a payload. Each party P has a unique IP address IP_P.

The protocol is associated with the security parameter $\lambda \in \mathbb{N}_+$, a key generation algorithm Kg that takes as input 1^λ and outputs a public and secret key pair, a header space[3] (for transport and application layers) $\mathcal{H} \subseteq \{0,1\}^*$, a payload space $\mathcal{PD} \subseteq \{0,1\}^*$, header and payload spaces $\mathcal{H}_{\mathsf{rst}} \subseteq \mathcal{H}, \mathcal{PD}_{\mathsf{rst}} \subseteq \mathcal{PD}$ for reset packets (described later), a resumption state space $\mathcal{RS} \subseteq \{0,1\}^*$, a stateful AEAD scheme[4] $\mathsf{sAEAD} = (\mathsf{sG}, \mathsf{sI}, \mathsf{sE}, \mathsf{sD})$ (with a message space $\mathcal{M} \subseteq \{0,1\}^*$, an associated data space $\mathcal{AD} \subseteq \{0,1\}^*$, and a state space $\mathcal{ST} \subseteq \{0,1\}^*$), *disjoint*[5] message spaces $\mathcal{M}_{\mathsf{KE}}, \mathcal{M}_{\mathsf{SC}}, \mathcal{M}_{\mathsf{pRST}} \subseteq \mathcal{M}$ with $\mathcal{M}_{\mathsf{KE}}, \mathcal{M}_{\mathsf{SC}}$ for messages encrypted during key exchange and secure channel phases respectively and $\mathcal{M}_{\mathsf{pRST}}$ for pre-reset messages (described later) encrypted in a secure channel phase, and a server configuration generation function $\mathsf{scfg\_gen}$ described below.

The protocol's execution is associated with the universal notion of time divided into discrete periods $\tau_1, \tau_2, \ldots$.[6] During its execution, both parties can keep states that are initialized to the empty string ε. In the beginning of each time period, the protocol may update each server's configuration state $\mathtt{scfg}$ with $\mathsf{scfg\_gen}$ (which takes as input 1^λ, a server secret key, and a time period, then outputs a server configuration state). Otherwise, $\mathsf{scfg\_gen}$ is undefined and without loss of generality the protocol is executed within a single time period.

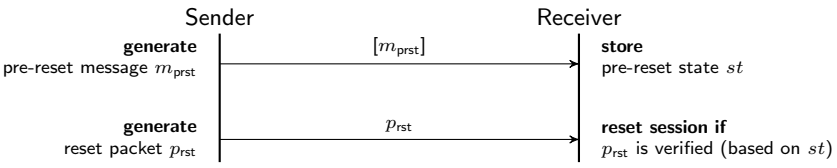

Figure 3: Stateless reset. [] indicates messages protected in a secure channel.

A *reset* packet enables a sender, who lost its session state due to some error condition (e.g., server reboots, denial-of-service attacks, etc.), to abruptly terminate a session with the receiver. A *pre-reset* message (e.g., a reset token in QUIC[TLS]) is sent to the receiver in a secure channel phase[7] before the sender loses its state in order to authenticate the sender's reset packet, as shown in Fig. 3. Each session has *at most one* pre-reset message for each party. A *non-reset* packet is not a reset packet. A *header-only* packet has no payload.

We say a party *rejects* a packet if its processing the packet leads to an error (defined according to the protocol), and *accepts* it otherwise.

The protocol has two modes, *full* and *resumption*. Its corresponding executions are referred to as the full and resumption sessions. Each resumption session is associated with a *single* previous full session and we say the resumption session *resumes* its associated full session. In the beginning of a full or resumption session, each party takes as input a list of messages[8] $\mathcal{M}^{\mathsf{snd}} = (M_1, \ldots, M_l), M_i \in$

[2]For the network-layer protocols, we only consider the Internet Protocol and its IP address header fields because our model mainly focuses on the application and transport layers and additionally only captures the IP-spoofing attack.

[3]Some protocol header fields (e.g., port numbers, checksums, etc.) can be excluded if they are not the focus of the security analysis.

[4]To fit TLS 1.3's encryption scheme, unlike QACCE we model QUIC's encryption scheme as a more general stateful AEAD scheme rather than a nonce-based one.

[5]Disjointness is a reasonable assumption as practical protocols (such as those in Table 1) enforce different leading bits for different types of messages.

[6]In practice, the duration of each time period is arbitrarily specified by a server to determine the lifetime of its configuration state.

[7]A pre-reset message can also be carried within an *encrypted* key exchange packet. We consider it encrypted as a separate secure channel packet to get a clean packet-authentication security model described later.

[8] Note that in reality the exchanged messages may depend on each other, but modeling that complicates our already complex protocol syntax and it has no real implications on our security analysis. Furthermore, for simplicity we consider transportation of *atomic* messages rather than a data *stream* that can be modeled as a stream-based channel [23] and later extended to capture multiplexing [47].

$\mathcal{M}_{\mathsf{SC}}, l \in \mathbb{N}$ (where the total message length $|\mathcal{M}^{\mathsf{snd}}|$ is polynomial in λ and $\mathcal{M}^{\mathsf{snd}}$ can be empty) as well as the other party's IP address. In a full session, the server takes as input its associated public and secret key pair (generated by running $\mathsf{Kg}(1^\lambda)$) and the client takes the server's public key as input[9]. In a resumption session, each party additionally takes as input its own resumption state $rs \in \mathcal{RS}$ (set in the associated full session). In either case, the client sends the first packet to start the session.

A D-stage (for an arbitrary $D \in \mathbb{N}_+$) msACCE protocol consists of D successive stages and each stage, e.g., the d-th ($d \in [D]$) stage, consists of one or two phases described as follows:

1) Key Exchange. At the end of this phase each party sets its d-th stage key $k^d = (k_c^d, k_s^d)$. At most one of k_c^d and k_s^d could be $\perp$, i.e., unused.[10] If this is the final stage in a full session, each party can send additional messages[11] in $\mathcal{M}_{\mathsf{KE}}$ encrypted with k^d and by the end of this phase each party sets its own resumption state.

2) Secure Channel. This phase is mandatory for the final stage but optional for other stages. In this phase, the parties can exchange messages from their input lists as well as pre-reset messages, encrypted and decrypted using the associated stateful AEAD scheme with k^d.[12] The client uses k_c^d to encrypt and the server uses it to decrypt, whereas the server uses k_s^d to encrypt and the client uses it to decrypt. They may also send reset or header-only packets. At the end of this phase, each party outputs a list of received messages (which may be empty).

Each message exchanged between the parties must belong to some unique phase at some unique stage. One stage's second phase and the next stage's first phase may overlap, and the two phases in the final stage may also overlap. We call the final stage key the *session* key and the other stage keys the *interim* keys.

Correctness. Consider a client and a server running a D-stage msACCE protocol in either mode without sending any reset packet. Each party's input message list $\mathcal{M}^{\mathsf{snd}}$, in which the messages are sent among D stages according to any possible partitioning $\mathcal{M}^{\mathsf{snd}} = \mathcal{M}_1^{\mathsf{snd}}, \ldots, \mathcal{M}_D^{\mathsf{snd}}$, is equal to the other party's total output message list $\mathcal{M}^{\mathsf{rcv}} = \mathcal{M}_1^{\mathsf{rcv}}, \ldots, \mathcal{M}_D^{\mathsf{rcv}}$, in which the message order is preserved. Each party terminates its session upon receiving the other party's reset packet.

Remark. With our more general protocol syntax, the ACCE [29] and QC [42] protocols can be classified into 1-stage and 2-stage msACCE protocols respectively.

4.2 Security Models

We propose two security models respectively for basic authenticated and confidential channel security and novel packet authentication. Our models do not consider the key exchange and secure channel phases independently, as was the case for some previous QUIC and TLS 1.3 security analyses [24, 19, 20, 40, 22], because QUIC's key exchange and secure channel phases are inherently inseparable and the TLS 1.3 full handshake does not fit into a composability framework, as discussed in [42, 20].

1) msACCE Standard Security Model:

In this msACCE standard (msACCE-std) security model, we consider the standard security goals such as server authentication[13] and channel security (which captures data privacy and integrity) for msACCE protocols. Our msACCE-std model is very similar to the standard security portion of the QACCE model [42], but extends it to capture more stages and use a more general stateful encryption scheme to fit both TLS 1.3 and QUIC.

[9] Since the client knows the identifier of its intended communicating peer when the protocol starts, our protocol is analyzed in the *pre-specified peer model* [44].

[10] This captures the case where a 0-RTT key only consists of a client encryption key while the server encryption key does not exist.

[11] This captures the post-handshake key exchange messages that are used for session resumption, post-handshake authentication, key update, etc.

[12] Note that our work does not capture key updates inside the secure channel, which is modeled by a recent work [27].

[13] Our msACCE-std model focuses on the most common server authentication, but can be extended to mutual authentication, e.g., as described in [35].

Like QACCE and other previous models, we consider a very powerful adversary who can control communications between honest parties, can adaptively learn their stage keys, and can adaptively corrupt servers to learn their long-term keys and secret states.

Our detailed security model is defined below.

Protocol Entities. The set of parties $\mathcal{P}$ consists of two disjoint type of parties: clients $\mathcal{C}$ and servers $\mathcal{S}$, i.e., $|\mathcal{P}| = |\mathcal{C}| + |\mathcal{S}|$.

Session Oracles. To capture multiple sequential and parallel protocol executions, each party $P \in \mathcal{P}$ is associated with a set of session oracles $\pi_P^1, \pi_P^2, \ldots$, where π_P^i models P executing a protocol instance in session $i \in \mathbb{N}_+$.

Session Identifiers. As part of the security model, *session identifiers* are used to model entity authentication, session key confirmation, and handshake integrity. Compared to the general definition of matching conversations [6, 29] used for modeling entity authentication, a session identifier `sid` is defined according to the protocol specifications and security goals, often as a *subset* of the entire communication transcript. For instance, QUIC's `sid` in QACCE [42] is defined as the second-round key exchange messages, i.e., `ClientHello` and `ServerHello`, while the first-round messages are excluded to allow for valid but different source-address tokens or signatures. Similarly, TLS 1.2's `sid` in ACCE [35] is defined as the first three key exchange messages, while the rest are excluded to allow for valid but different encrypted `Finished` messages. A msACCE protocol may have different session identifiers in full and resumption modes, but for simplicity we use the same notation `sid`.

Peers and Partners. We say a client oracle and a server oracle are each other's *peer* if they share the same session identifier `sid`; we say they are each other's *partner* if they share the same first-stage session identifier $\texttt{sid}_1$ (i.e., `sid` restricted to the first stage), which intuitively means that they set the first stage key with each other. Note that a client oracle may have more than one partner if $\texttt{sid}_1$ consists of only message(s) sent from the client oracle, which can be replayed to the same[14] server to establish multiple (identical) first-stage keys. Therefore, a session oracle's partner may not be its final unique communication peer.

Security Experiments. In the beginning of the experiments, run $\mathsf{Kg}(1^\lambda)$ for all servers to generate the public and secret key pairs and initialize the global states of all parties and the local states of all session oracles. In the beginning of each time period, run $\mathsf{scfg\_gen}$ (if defined) for each server to update its configuration state `scfg`. We assume that both the server oracles and the adversary $\mathcal{A}$ are aware of the current time period. Let $N \in \mathbb{N}_+$ denote the maximum number of msACCE protocol instances for each party and $D \in \mathbb{N}_+$ denote the maximum number of stages in each session. The channel security experiment is associated with an authentication level $al \in [4]$. Each oracle π_P^i is associated with a random bit $b_P^i \xleftarrow{\$} \{0,1\}$ and has a global state $\widetilde{m}$ (initialized to $\perp$) that stores its pre-reset message. The adversary $\mathcal{A}$ is given all public keys and the IP addresses associated with all parties and then interacts with the session oracles via the following queries:

- $\mathsf{Connect}(\pi_C^i, S)$, for $C \in \mathcal{C}, S \in \mathcal{S}, i \in [N]$.

This query allows the adversary to ask a specified client oracle to start a full session with a specified server. π_C^i outputs the first packet it would send to S in a full session according to the protocol. This output packet is returned to $\mathcal{A}$ instead of S. After this query, we say S is the *intended server* of π_C^i.

- $\mathsf{Resume}(\pi_C^i, S, i')$, for $C \in \mathcal{C}, S \in \mathcal{S}$, $i, i' \in [N]$, $i' < i$.

This query allows the adversary to ask a specified client oracle to start a resumption session with a specified server to resume a specified full session between the two parties, if the associated previous client oracle has set its resumption state. π_C^i takes the resumption state set by $\pi_C^{i'}$ as input and outputs the first packet it would send to S in a resumption session according to the protocol, where the output packet is returned. If $\pi_C^{i'}$ has not set its resumption state, returns $\perp$.

- $\mathsf{Send}(\pi_P^i, pkt)$, for $P \in \mathcal{P}, i \in [N], pkt \in \{0,1\}^*$.

This query allows the adversary to send any packet to a specified session oracle and get its response

[14] In practice, 0-RTT replay attacks can be mounted to *different* servers with the same public-secret key pair. However, 0-RTT key exchange message(s) replayed to other servers with different public-secret key pairs should be rejected.

in a key exchange phase. If π_P^i is in a key exchange phase, pkt is sent to π_P^i and the response is returned, otherwise, returns $\perp$.

• $\mathsf{Reveal}(\pi_P^i, d)$, for $P \in \mathcal{P}, i \in [N], d \in [D]$.

This query allows the adversary to learn any stage key of a specified session oracle. It returns the contents of π_P^i's d-th stage key state k^d. After this query, we say k^d was *revealed*.

• $\mathsf{Corrupt}(S)$, for $S \in \mathcal{S}$.

This query allows the adversary to learn the long-term secret of a specified server along with its current states. It returns S's secret key, configuration and resumption states in the current time period, and all other current states. After this query, we say S was *corrupted*.

• $\mathsf{Encrypt}(\pi_P^i, d, ad, m_0, m_1)$, for $P \in \mathcal{P}, i \in [N], d \in [D], ad \in \mathcal{AD}, m_0, m_1 \in \mathcal{M}_{\mathsf{SC}} \cup \mathcal{M}_{\mathsf{pRST}} \cup \{\mathtt{rst}\}$.

This query allows the adversary to specify any associated data and any two secure channel or pre-reset messages of the same length, then get the ciphertext of one message determined by b_P^i, the random bit associated with a specified session oracle at a specified stage. This query also stores one of the specified pre-reset messages determined by b_P^i and allows the adversary to retrieve it on request. (Recall that each oracle has at most one pre-reset message, so this query stores only the first pre-reset message and rejects others.)

The detailed procedures are listed as follows:

1: if m_0, m_1 are of different types (i.e., $\mathcal{M}_{\mathsf{SC}}$ or $\mathcal{M}_{\mathsf{pRST}}$ or $\{\mathtt{rst}\}$) or $|m_0| \neq |m_1|$ or π_P^i is not in its d-th secure channel phase or $k_p^d = \perp$ (where $p = c$ if $P \in \mathcal{C}$ and $p = s$ if $P \in \mathcal{S}$), return $\perp$
2: if $m_0, m_1 \in \mathcal{M}_{\mathsf{pRST}}$, set $\widetilde{m} \leftarrow m_{b_P^i}$ if $\widetilde{m} = \perp$ or return $\perp$ otherwise
3: if $m_0 = m_1 = \mathtt{rst}$, return $\widetilde{m}$
4: (upon setting each encryption stage key, initialize st_e with sl and set $u \leftarrow 0$, $sent \leftarrow \varepsilon$)
5: $u \leftarrow u + 1$, $(sent.ct_u, st'_e) \xleftarrow{\$} \mathsf{sE}(k_p^d, ad, m_{b_P^i}, st_e)$
6: $(sent.ad_u, st_e) \leftarrow (ad, st'_e)$
7: return $sent.ct_u$

• $\mathsf{Decrypt}(\pi_P^i, d, ad, ct)$, for $P \in \mathcal{P}, i \in [N], d \in [D], ad \in \mathcal{AD}, ct \in \{0,1\}^*$.

This query allows the adversary to specify any associated data and any ciphertext to be decrypted by the *partner(s)* of a specified session oracle at a specified stage, then get the secret bit b_P^i if the decrypted message is a valid secure channel or pre-reset message and this query is "out-of-sync" (defined below), otherwise, it gets $\perp$.

The detailed procedures are listed as follows:

1: if π_P^i is not in its d-th secure channel phase or $k_p^d = \perp$ (where $p = c$ if $P \in \mathcal{C}$ and $p = s$ if $P \in \mathcal{S}$), return $\perp$
2: (upon setting each decryption stage key, initialize st_d with sl and set $v \leftarrow 0$, $rcvd \leftarrow \varepsilon$, $\mathtt{outofsync} \leftarrow 0$)
3: $v \leftarrow v + 1$, $rcvd.ct_v \leftarrow ct$, $(m, st'_d) \leftarrow \mathsf{sD}(k_p^d, ad, ct, st_d)$
4: $(rcvd.ad_v, st_d) \leftarrow (ad, st'_d)$
5: if $m \notin \mathcal{M}_{\mathsf{SC}} \cup \mathcal{M}_{\mathsf{pRST}}$, set $m \leftarrow \perp$
6: if $(al = 4) \wedge \mathsf{cond}_4$ or $(al \leq 3) \wedge (m \neq \perp) \wedge \mathsf{cond}_{al}$,
set $\mathtt{outofsync} \leftarrow 1$
7: if $(\mathtt{outofsync} = 1) \wedge (m \neq \perp)$, return b_P^i, otherwise, return $\perp$

The above "out-of-sync" condition (line 6) varies with the associated authentication level $al \in [4]$. We recall the 4 authentication conditions cond_{al} defined in [10] below:

cond_1: $(\nexists w : (ct = sent.ct_w) \wedge (ad = sent.ad_w))$

cond_2: $(\nexists w : (ct = sent.ct_w) \wedge (ad = sent.ad_w)) \vee (\exists w < v : ct = rcvd.ct_w)$

cond_3: $(\nexists w : (ct = sent.ct_w) \wedge (ad = sent.ad_w)) \vee (\exists w, x, y : (w < v) \wedge (sent.ct_x = rcvd.ct_w) \wedge (sent.ct_y = rcvd.ct_v) \wedge (x \geq y))$

cond_4: $(u < v) \vee (ct \neq sent.ct_v) \vee (ad \neq sent.ad_v)$

Note that cond_1 corresponds to the lowest authentication level (e.g., the stateful AEAD scheme in QUIC) that only guarantees no forgeries, while cond_4 corresponds to the highest authentication level

(e.g., the stateful AEAD scheme in TLS 1.3) that prevents forgeries, replays, reordering, and dropping.

Advantage Measures. An adversary $\mathcal{A}$ against a msACCE protocol Π in msACCE-std has the following advantage measures (that each takes the security parameter as an implicit input).

• *Server Authentication.* We define $\mathbf{Adv}_{\Pi}^{\mathsf{s\text{-}auth}}(\mathcal{A})$ as the probability that there exist a client oracle π_C^i and its intended server S such that the following conditions hold:

1. π_C^i has set its session key;
2. S was not corrupted before π_C^i set its session key;
3. No interim keys of π_C^i or its partner(s) were revealed;[15]
4. There is no unique server oracle π_S^j that is π_C^i's peer.

The above captures the attacks in which the adversary impersonates a server to make the client mistakenly believe that it shares the session key with the server.

• *(level-al)*[16] *Channel Security.* We define $\mathbf{Adv}_{\Pi}^{\mathsf{cs}\text{-}al}(\mathcal{A})$ as $|2\Pr[b_P^i = b'] - 1|$, where $al \in [4]$ is the specified authentication level and (P, i, b') is output by $\mathcal{A}$, such that the following conditions hold:

1. If $P = S \in \mathcal{S}$, π_S^i has a peer π_C^j; If $P = C \in \mathcal{C}$, denote S as π_C^i's intended server;
2. If S was corrupted, then 1) no $\mathsf{Encrypt}(\pi_P^i, \cdot, \cdot, \cdot, \cdot)$ queries were made after S was corrupted, and 2) furthermore, no $\mathsf{Encrypt}(\pi_P^i, d, \cdot, \cdot, \cdot)$ queries were made if *forward secrecy* is not required for the d-th stage keys;
3. No stage keys of π_P^i or its partner(s) were revealed;
4. If two different pre-reset messages were queried in the d-th stage, later no $\mathsf{Encrypt}(\pi_P^i, \cdot, \cdot, \mathtt{rst}, \mathtt{rst})$ queries were made.

The above captures the attacks in which the adversary compromises the privacy or integrity of secure channel messages without revealing stage keys or revealing the hidden pre-reset message or corrupting the server before the client sets its last stage key (which may not be the session key). If some stage key is not supposed to provide forward secrecy, the adversary is further restricted to not corrupt the server during the same time period associated with the target session.

2) msACCE Packet-Authentication Security Model:

In this msACCE packet-authentication (msACCE-pauth) security model, we consider security goals related to packet authentication beyond those captured by the msACCE-std model. Note that msACCE-std essentially focuses only on the packet fields in the application layer, while msACCE-pauth further covers transport-layer headers and IP addresses.

First, we consider IP spoofing prevention (a.k.a. source authentication) as with the QACCE model, but, as illustrated later, generalize one of the QACCE queries to additionally capture IP spoofing attacks in the full sessions. Then, more importantly, we define four novel packet-level security notions (elaborated later): *KE Header Integrity*, *KE Payload Integrity*, *SC Header Integrity*, and *Reset Authentication*, which enable a comprehensive and fine-grained security analysis of layered protocols.

In particular, KE Header and Payload Integrity respectively capture the header and payload integrity of key exchange packets. Such security issues have not been investigated before and, as we show later, lead to new availability attacks for both TFO+TLS 1.3 and UDP+QUIC. Furthermore, we employ SC Header Integrity to capture the header integrity of non-reset packets in secure channel phases. Note that, unlike the availability attacks shown in [42], successful attacks breaking our security

[15] More precisely, we allow revealing other stage keys (if any), except the first stage key, of a π_C^i's partner which observes the same session identifier at only the first stage but not the next one, because such a partner's key exchange message is never received by π_C^i. Similar condition relaxation also holds for our other security notions.

[16] Note that Server Authentication does not depend on *al*. To see this, consider the weaker Server Authentication notion defined in the same way except its security experiment has no Decrypt query and the Encrypt query takes only one message as input. One can easily reduce our Server Authentication security with an arbitrary authentication level *al* to the above weaker Server Authentication security. Therefore, those notions are all equivalent.

notions are *harder or impossible to detect* by the client as they do not affect the client's session key establishment, so they are more harmful in this sense. Finally, our model captures malicious *undetectable* session resets in a secure channel phase with Reset Authentication.

As with the msACCE-std model, msACCE-pauth captures multiple stages and considers a very powerful adversary. It also inherits the same definitions of protocol entities, session oracles, peers, and partners.

Security Experiments. Consider the same experiment setups as in msACCE-std, except that no random bit b_P^i is needed. The adversary $\mathcal{A}$ is given all the public parameters and interacts with the session oracles via the same $\mathsf{Connect}, \mathsf{Resume}, \mathsf{Send}, \mathsf{Reveal}, \mathsf{Corrupt}$ queries as in the msACCE-std model[17], as well as the following:

• $\mathsf{Connprivate}(\pi_C^i, \pi_S^j, \mathtt{cmp})$, for $C \in \mathcal{C}$, $S \in \mathcal{S}$, $i, j \in [N]$, $\mathtt{cmp} \in \{0, 1\}$.

This query allows the adversary to run a complete or partial full session between any specified client and server oracles without observing their communication. This query always returns $\perp$. If $\mathtt{cmp} = 1$, π_C^i and π_S^j run a *complete* full session privately without showing their communication to $\mathcal{A}$. If $\mathtt{cmp} = 0$, π_C^i and π_S^j run a *partial* full session privately such that the last packet sent from π_C^i right before π_S^j sets its first stage key is blocked. Note that this query extends the QACCE $\mathsf{Connprivate}$ query [42] to model IP-spoofing attacks targeting both *full* and resumption sessions, with help of an input flag $\mathtt{cmp}$.

• $\mathsf{Pack}(\pi_P^i, ad, m)$, for $P \in \mathcal{P}, i \in [N], ad \in \mathcal{AD}, m \in \mathcal{M}_{\mathsf{SC}} \cup \mathcal{M}_{\mathsf{pRST}} \cup \{\mathtt{prst}, \mathtt{rst}\}$.

This query allows the adversary to specify any associated data and any message in a secure channel phase, then get the packet output by a specified session oracle. The adversary can also ask the specified session oracle to set a specified pre-reset message or output its reset packet.

The detailed procedures are listed as follows:

1: If π_P^i is not in a secure channel phase, return $\perp$
2: If $m \in \mathcal{M}_{\mathsf{pRST}}$, π_P^i sets its pre-reset message equal to m
3: If $m = \mathtt{prst}$, π_P^i generates its own pre-reset message (hidden from $\mathcal{A}$)
4: (Recall that each oracle has at most one pre-reset message, so the above two cases are allowed at most once for π_P^i.)
5: If $m \neq \mathtt{rst}$, π_P^i outputs the packet it would send to its peer according to the protocol, for the specified associated data ad and message m (which are useless if $m = \mathtt{prst}$), then this packet is returned
6: If $m = \mathtt{rst}$, π_P^i outputs its reset packet (if any), which is returned

• $\mathsf{Deliver}(\pi_P^i, pkt)$, for $P \in \mathcal{P}, i \in [N], pkt \in \{0, 1\}^*$.

This query allows the adversary to deliver any packet to a specified session oracle and get its response in a secure channel phase. If π_P^i is in a secure channel phase, pkt is delivered to π_P^i and its response (if any) is returned, otherwise, returns $\perp$.

Advantage Measures. An adversary $\mathcal{A}$ against a msACCE protocol Π in msACCE-pauth has the following associated advantage measures (that each takes the security parameter as an implicit input).

• *IP-Spoofing Prevention.* We define $\mathbf{Adv}_{\Pi}^{\mathsf{ipsp}}(\mathcal{A})$ as the probability that there exist a client oracle π_C^i and a server oracle π_S^j such that the following conditions hold:

1. π_S^j has set its first stage key right after a $\mathsf{Send}(\pi_S^j, (\mathrm{IP}_C, \mathrm{IP}_S, \cdot, \cdot))$ query;
2. S was not corrupted before π_S^j set its first stage key;
3. The only allowed queries concerning both C and S in the time period associated with π_S^j are:
   - $\mathsf{Connprivate}(\pi_C^x, \pi_S^y, \cdot)$ for any $x, y \in [N]$, and
   - $\mathsf{Send}(\pi_S^y, (\mathrm{IP}_C, \mathrm{IP}_S, \cdot, \cdot))$ for any $y \in [N]$, where $(\mathrm{IP}_C, \mathrm{IP}_S, \cdot, \cdot)$ is the last packet received by π_S^y right before it sets its first stage key.

[17] Note that $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$ queries are not needed because msACCE-pauth does not consider data privacy explicitly.

The above captures the attacks in which the adversary fools a server into accepting a spurious connection request seemingly from an impersonated client, without observing any previous communication between the client and server in the same time period. Such attacks can lead to exhaustion of server resources (i.e., denial-of-service) caused by excessive stage key derivation.

• *KE Header Integrity.* We define $\mathbf{Adv}^{\mathsf{int\text{-}keh}}_{\Pi}(\mathcal{A})$ as the probability that there exist a client oracle π^i_C and a server oracle π^j_S such that the following conditions hold:

1. π^i_C has set its session key and has a peer π^j_S;
2. S was not corrupted before π^i_C set its session key;
3. No interim keys of π^i_C or its partner(s) were revealed;
4. In a key exchange phase before π^i_C set its session key, π^i_C (resp. π^j_S) accepted a packet with a header that was not output by π^j_S (resp. π^i_C).

The above captures the attacks in which the adversary modifies the protocol header of a key exchange packet of the communicating parties without affecting the client setting its session key. In the above definition, we assume that a client sets its session key *immediately* after sending its last key exchange packet(s) (if any). Then, a forged packet that leads to a successful attack cannot be any of these last packet(s), which have not yet been sent to the server. The same assumption is made for KE Payload Integrity defined below.

• *KE Payload Integrity.* We define $\mathbf{Adv}^{\mathsf{int\text{-}kep}}_{\Pi}(\mathcal{A})$ as the probability that there exist a client oracle π^i_C and a server oracle π^j_S such that the same (1)∼(3) conditions as in the above KE Header Integrity notion hold and the following holds:

4. In a key exchange phase before π^i_C set its session key, π^i_C (resp. π^j_S) accepted a packet with a payload that was not output by π^j_S (resp. π^i_C).

The above captures the attacks in which the adversary modifies the payload of a key exchange packet of the communicating parties without affecting the client setting its session key. Note that the above notion can be trivially achieved if the session identifier includes the entire key exchange transcript (excluding some headers), which is the case for TLS 1.3. However, some other protocols like QUIC only have partial transcripts as their session identifiers.

• *SC Header Integrity.* We define $\mathbf{Adv}^{\mathsf{int\text{-}h}}_{\Pi}(\mathcal{A})$ as the probability that $\mathcal{A}$ outputs (P, i, d) such that the same (1)∼(3) conditions as in the Channel Security notion hold and the following holds:

4. In the secure channel phase of the d-th stage, π^i_P accepted a non-reset packet with a header that was not output by its partner(s) (via Pack queries), or π^i_P accepted a non-reset header-only packet.

The above captures the attacks in which the adversary creates a valid non-reset secure channel packet by forging the protocol header without breaking any Channel Security conditions. Note that in the above security notion an invalid header forgery is detected *immediately* after the malicious packet is received and processed, while the detection of invalid packet forgeries in a key exchange phase (e.g., for plaintext packets) can be *delayed* to the point when the client sets its session key, according to the definitions of KE Header and Payload Integrity.

• *Reset Authentication.* We define $\mathbf{Adv}^{\mathsf{rst\text{-}auth}}_{\Pi}(\mathcal{A})$ as the probability that $\mathcal{A}$ outputs (P, i, d) such that the same (1)∼(3) conditions as in the Channel Security notion hold and the following holds:

4. In the secure channel of the d-th stage, π^i_P accepted a packet output by a $\mathsf{Pack}(\cdot, \cdot, \mathtt{prst})$ query to its partner $\pi^j_{P'}$. Later (in the d-th or a later stage), π^i_P accepted a reset packet but $\mathcal{A}$ made no $\mathsf{Pack}(\pi^j_{P'}, \cdot, \mathtt{rst})$ queries.

The above captures the attacks in which the adversary forges a valid reset packet without breaking any Channel Security conditions. Note that such attacks are *undetectable* by the accepting party, as opposed to a network attacker that simply drops packets.

Remark about msACCE Security Model Completeness and Low-Layer Integrity. Note that the payload integrity in secure channels is captured by Channel Security. Our msACCE-std and msACCE-pauth models *completely* capture the authentication (or integrity) of all packet fields in the transport and application layers. Furthermore, msACCE-pauth captures (network-layer) IP-Spoofing Prevention against weaker off-path attackers (i.e., those can only inject packets without observing the communication), but leaves other integrity attacks on low layers (e.g., network, link, and physical layers) uncovered. Such attacks may affect packet forwarding, node-to-node data transfer, or raw data transmission, which are outside the scope of our work.

5 Provable Security Analysis

Equipped with msACCE security models, we now analyze and compare the security of TFO+TLS 1.3, UDP+QUIC, and UDP+QUIC[TLS]. The security results are summarized in Table 2 (recalled below). As mentioned in the Introduction, by [22] results, none of the above protocols achieves forward secrecy for 0-RTT keys or protects against 0-RTT data replays (which contribute to the first two rows in the table). We now move to the detailed analyses and start with TFO+TLS 1.3.

Table 2: Security comparison (reminder)

	TLS 1.3 +TFO	QUIC +UDP	QUIC[TLS] +UDP
0-RTT Key Forward Secrecy [22]	✗	✗	✗
0-RTT Data Anti-Replay [22]	✗	✗	✗
Server Authentication	✓	✓	✓
Channel Security	✓	✓	✓
IP-Spoofing Prevention	✓	✓	✓
KE Header Integrity	✗	✗	✗
KE Payload Integrity	✓	✗	✗
SC Header Integrity	✗	✓	✓
Reset Authentication	✗	✗	✓

5.1 TLS 1.3 over TFO

1) Protocol Description: Referring to the msACCE protocol syntax, a TFO+TLS 1.3 2-RTT full handshake (see Fig. 1 (a)) is a 2-stage msACCE protocol in the full mode and a 0-RTT resumption handshake (see Fig. 1 (b)) is a 3-stage msACCE protocol in the resumption mode. Note that we focus only on the main components of the handshakes and omit more advanced features such as 0.5-RTT data, client authentication, and post-handshake messages (except `NewSessionTicket`). In a full handshake, the initial keys are set after sending or receiving `ServerHello` and the final keys (i.e., session keys) are set after sending or receiving `ClientFinished` (but only handshake messages up to `ServerFinished` are used for final key generation). In a 0-RTT resumption handshake, the parties set 0-RTT keys to encrypt or decrypt 0-RTT data, after sending or receiving `ClientHello`.

According to the TFO and TLS 1.3 specifications [15, 53], the TFO+TLS 1.3 header contains the TCP header (see Fig. 4 in Appendix A). We ignore some uninteresting header fields such as port numbers and the checksum because modifying them only leads to redirected or dropped packets. Such adversarial capabilities are already considered in the msACCE security models. We thus define the header space $\mathcal{H}$ as containing the following TCP header fields: a 32-bit sequence number `sqn`, a 32-bit acknowledgment number `ack`, a 4-bit data offset `off`, a 6-bit reserved field `resvd`, a 6-bit control

bits field `ctrl`, a 16-bit window `window`, a 16-bit urgent pointer `urgp`, a variable-length ($\leq$ 320-bit) padded options `opt`. For encrypted packets, $\mathcal{H}$ additionally contains the TLS 1.3 record header fields: an 8-bit type `type`, a 16-bit version `ver`, and a 16-bit length `len`. We further define reset packets as those with the RST bit (i.e., the 4-th bit of `ctrl`) set to 1. Note that $\mathsf{scfg\_gen}$ is undefined.

TLS 1.3 enforces different content types for encrypted key exchange and secure channel messages. For simplicity, we define $\mathcal{M}_{\mathsf{KE}}$ and $\mathcal{M}_{\mathsf{SC}}$ as consisting of bit strings differing in their first bits. $\mathcal{M}_{\mathsf{pRST}} = \varnothing$. In Appendix C, we define TLS 1.3's stateful AEAD scheme $\mathsf{sAEAD}_{\mathrm{TLS}} = (\mathsf{sG}, \mathsf{sE}, \mathsf{sD})$ based on the underlying nonce-based AEAD scheme $\mathsf{AEAD} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$ (instantiated with AES-GCM [43] or others as documented in [53]).

We refer to Appendix E for the remaining details of TFO and refer to [22, 11] for the detailed descriptions of TLS 1.3 handshake messages and key generations in earlier TLS 1.3 drafts as well as [53] for the latest updates.

2) Security Results: TFO+TLS 1.3's session identifier $\mathtt{sid}_{\mathrm{TLS}}$ is defined as all key exchange messages (that could be encrypted) from `ClientHello` to `ServerFinished`, excluding TCP headers and IP addresses. The msACCE-std security of TFO+TLS 1.3 is by definition independent of TCP headers and is hence provided by the TLS 1.3 component. Previous works [21, 22, 40] only proved TLS 1.3's authenticated key exchange security, i.e., the stage keys are authenticated and indistinguishable from random ones under reasonable computational assumptions. In Appendix D, we outline how their security results can be adapted to prove TLS 1.3's Server Authentication and level-4 Channel Security in our msACCE-std model, by additionally relying on the level-4 AEAD security of $\mathsf{sAEAD}_{\mathrm{TLS}}$ (which can be reduced to the nonce-based AEAD security of the underlying AEAD as shown in [18]).

The msACCE-pauth security analyses are shown as follows.

IP-Spoofing Prevention. This security of TFO+TLS 1.3 is provided by the TFO component through TCP sequence number randomization and TFO cookies. By modeling the cookie generation function F, an AES-128 block cipher, as a PRF, we have the following theorem with the proof in Appendix E:

Theorem 1 *For any efficient adversary $\mathcal{A}$ making at most q_{S} Send queries, there exists an efficient adversary $\mathcal{B}$ such that:*

$$\mathbf{Adv}^{\mathsf{ipsp}}_{\text{TFO+TLS 1.3}}(\mathcal{A}) \leq |\mathcal{S}|\mathbf{Adv}^{\mathsf{prf}}_{F}(\mathcal{B}) + \frac{q_{\mathsf{S}}|\mathcal{S}|}{2^{|\mathtt{sqn}|}} ,$$

where $|\mathcal{S}|$ is the number of servers and $|\mathtt{sqn}|$ is the bit length of the TCP sequence number.

Remark. Note that the above second probability term is not very small since $|\mathtt{sqn}| = 32$. Actually, an attacker can indeed successfully establish a TCP connection using a spoofed client IP address with roughly 2^{32} random guesses. However, our security bound is still acceptable because each guess the attacker made corresponds to an "online" Send query. Here "online" means that the attacker has to interact with the server every time it makes a guess, no matter how many resources it may consume offline. Such online attacks are easy to detect and suppress.

KE Header Integrity. TFO+TLS 1.3 does not achieve this security notion because TCP headers are never authenticated. We find a new practical attack below, where an efficient adversary $\mathcal{A}$ can always get $\mathbf{Adv}^{\mathsf{int\text{-}keh}}_{\text{TFO+TLS 1.3}}(\mathcal{A}) = 1$:

TFO Cookie Removal. $\mathcal{A}$ can first make $\pi_C^{i'}$ complete a full handshake with $\pi_S^{j'}$ (via $\mathsf{Connect}, \mathsf{Send}$ queries), then query $\mathsf{Resume}(\pi_C^i, S, i')$ ($i' < i$) to get the output packet $(\mathrm{IP}_C, \mathrm{IP}_S, H, pd)$, which is a SYN packet with a TFO cookie. $\mathcal{A}$ then modifies the `opt` field of H to get a new $H' \neq H$ that contains no cookie. The resulting SYN packet will be accepted by a new server oracle π_S^j, which will then respond with a SYN-ACK packet that does not contain a TFO cookie, indicating a fallback to the standard 3-way TCP. As a result, a 1-RTT handshake is needed to complete the connection and any 0-RTT data sent with SYN would be retransmitted. This eliminates the entire benefit of TFO without being detected, resulting in reduced performance and increased handshake latency. A similar attack is possible by removing the TFO cookie in a server's SYN-ACK packet.

Interestingly, clients are supposed to cache negative TFO responses and avoid sending TFO connections again for a lengthy period of time. This is because the most likely explanation for this behavior is that the server does not support TFO, but only standard TCP [15]. As a result, performing this attack for a single connection prevents TFO from being used with this server for a lengthy time period (i.e., days or weeks).

KE Payload Integrity. TFO+TLS 1.3 is secure in this regard simply because $\texttt{sid}_{\text{TLS}}$ consists of the payloads of all key exchange packets exchanged between the communicating parties before the client sets its session key. That is, for any client oracle that has a peer server oracle, by definition they observe the same $\texttt{sid}_{\text{TLS}}$ and hence no key exchange packet payload can be modified, i.e., $\mathbf{Adv}^{\mathsf{int\text{-}kep}}_{\text{TFO+TLS 1.3}}(\mathcal{A}) = 0$ for any efficient adversary $\mathcal{A}$.

SC Header Integrity. TFO+TLS 1.3 does not achieve this security notion again because of the unauthenticated TCP headers. A efficient adversary $\mathcal{A}$ can get $\mathbf{Adv}^{\mathsf{int\text{-}h}}_{\text{TFO+TLS 1.3}}(\mathcal{A}) = 1$ by either modifying the TCP header of an encrypted packet (e.g., reducing the `window` value) or by forging a header-only packet (e.g., removing the payload of an encrypted packet and changing its `ack` value). Such packets are valid and will be accepted by the receiving session oracle.

The above fact exposes the adversary's ability to arbitrarily modify or even entirely forge the information in the TCP header, which is being relied on to provide reliable delivery, in-order delivery, flow control, and congestion control for the targeted flow. This leads to a whole host of availability attacks that the networking community has been slowly uncovering via manual investigation over the last 30 years [56, 32, 4, 14, 38, 37, 30, 51, 13, 26, 50, 45, 61, 31]. Some of the practical attacks are described in Appendix F.

Reset Authentication. TFO+TLS 1.3 is insecure in this sense because its reset packet, TCP Reset, is an unauthenticated header-only packet. This leads to a practical attack below, where an efficient adversary $\mathcal{A}$ always gets $\mathbf{Adv}^{\mathsf{rst\text{-}auth}}_{\text{TFO+TLS 1.3}}(\mathcal{A}) = 1$:

TCP Reset Attack. $\mathcal{A}$ can first make two session oracles complete a handshake using $\mathsf{Connect}, \mathsf{Send}$ queries, then use $\mathsf{Pack}, \mathsf{Deliver}$ queries to let them exchange secure channel packets. By observing these packet headers, $\mathcal{A}$ can easily forge a valid reset packet by setting its RST bit to 1 and the remaining header fields to reasonable values. This attack will cause TCP to tear down the connection immediately without waiting for all data to be delivered.

Note that even an off-path adversary who can only inject packets into the communication channel may be able to accomplish this attack. The injected TCP reset packet needs to be within the receive window for the client or server, but [61] demonstrated that a surprisingly small number of packets is needed to achieve this, thanks to the large receive windows typically used by implementations.

5.2 QUIC over UDP

1) Protocol Description: Referring to the msACCE protocol syntax, an UDP+QUIC 1-RTT full handshake (see Fig. 2 (a)) is a 2-stage msACCE protocol in the full mode and a 0-RTT resumption handshake (see Fig. 2 (b)) is a 2-stage msACCE protocol in the resumption mode. The initial keys are set after sending or receiving `ClientHello` and the final keys (i.e., session keys) are set after sending or receiving `ServerHello`.

According to the UDP and QUIC specifications [55, 48, 39], the UDP+QUIC header contains the UDP header (see Fig. 5 in Appendix A) and the QUIC header (described below). As with the TCP header, we ignore the port numbers and checksum in the UDP header. Similarly, we also ignore the UDP length field because it affects only the length of the QUIC header and payload, while the adversary is already allowed to directly modify the packet length. We thus can completely omit the UDP header and define the header space $\mathcal{H}$ as containing the following QUIC header fields: an 8-bit public flag `flag`, a 64-bit connection ID `cid`, a variable-length (≤ 48 bits) sequence number `sqn`, and other optional fields. We further define reset packets as those with the PUBLIC_FLAG_RESET bit (i.e., the 7-th bit of `flag`) set to 1. A reset packet header only contains `flag` and `cid`.

As with TLS 1.3, for UDP+QUIC we define $\mathcal{M}_{\mathsf{KE}}$ and $\mathcal{M}_{\mathsf{SC}}$ as consisting of bit strings differing in their first bits. $\mathcal{M}_{\mathsf{pRST}} = \varnothing$. In Appendix C, we define QUIC's stateful AEAD scheme $\mathsf{sAEAD}_{\mathrm{QUIC}} = (\mathsf{sG}, \mathsf{sE}, \mathsf{sD})$ based on the underlying nonce-based AEAD scheme $\mathsf{AEAD} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$ (instantiated with AES-GCM [43]).

We refer to [42] for the detailed descriptions of scfg_gen and QUIC handshake messages and key generations.

2) Security Results: UDP+QUIC's session identifier $\mathtt{sid}_{\mathrm{QUIC}}$ is defined as the `ClientHello` payload and `ServerHello`, excluding IP addresses. The msACCE-std security of UDP+QUIC follows from prior works as we discuss in Appendix G. Note that UDP+QUIC only achieves level-1 Channel Security, but, as discussed in [42], QUIC implicitly prevents packet reordering by authenticating `sqn` in the packet header. It also prevents replays and dropping with frame sequence numbers encrypted in the payload. Therefore, UDP+QUIC essentially achieves level-4 authentication as TLS 1.3 does.

The msACCE-pauth security analyses are shown as follows.

IP-Spoofing Prevention. In [42], QUIC has been proven secure against IP spoofing based on the AEAD security. Their IP-spoofing security notion is the same as our IP-Spoofing Prevention notion for UDP+QUIC except that ours additionally captures attacks in full sessions. However, since source-address tokens are validated in both full and resumption sessions, their results can be trivially adapted to show that UDP+QUIC achieves IP-Spoofing Prevention.

KE Header and Payload Integrity. UDP+QUIC does not achieve these security notions because its first-round key exchange messages, i.e., `InchoateClientHello` and `ServerReject`, and any invalid `ClientHello` are not fully authenticated. Interestingly, a variety of existing attacks on QUIC's availability discovered in [42] are all examples of key exchange packet manipulations (e.g., the server config replay attack, connection ID manipulation attack, etc.), but these attacks cause connection failure and hence are easy to detect. However, successful attacks breaking KE Header or Payload Integrity will be harder (if not impossible) to detect.

For KE Header Integrity, we do not find any harmful attacks but theoretical attacks exist. For instance, an efficient adversary $\mathcal{A}$ can get $\mathbf{Adv}^{\mathsf{int\text{-}keh}}_{\mathrm{UDP+QUIC}}(\mathcal{A}) = 1$ as follows. $\mathcal{A}$ can first query $\mathsf{Connect}(\pi^i_C, S)$ to get the output packet $(\mathrm{IP}_C, \mathrm{IP}_S, H, pd)$, then modify the `flag` and `sqn` fields of H to get a new header $H' \neq H$ that only changes `sqn`'s length but not its value. The resulting packet will be accepted by a new server oracle π^j_S. This attack has no practical impact on UDP+QUIC but it successfully modifies the protocol header without being detected.

For KE Payload Integrity, we find a new practical attack described below where an efficient adversary $\mathcal{A}$ can get $\mathbf{Adv}^{\mathsf{int\text{-}kep}}_{\mathrm{UDP+QUIC}}(\mathcal{A}) \approx 1$:

ServerReject Triggering. $\mathcal{A}$ can first let $\pi^{i'}_C$ complete a full handshake with $\pi^{j'}_S$ with $\mathsf{Connect}, \mathsf{Send}$ queries, then query $\mathsf{Resume}(\pi^i_C, S, i')$ $(i' < i)$ to get the output `ClientHello` packet. $\mathcal{A}$ then modifies its payload by replacing the source-address token `stk` with a random value, which with high probability is invalid. Sending this modified packet to a new server oracle π^j_S will trigger a `ServerReject` packet containing a new valid `stk`. This as a result downgrades the original 0-RTT resumption connection to a full 1-RTT connection, which causes increased latency and results in the retransmission of any 0-RTT data. Note that this attack is hard to detect because π^i_C may think its original $\mathtt{stk}'$ has expired (although this does not happen frequently).

SC Header Integrity. UDP+QUIC is secure in this regard because it does not allow header-only packets to be sent in the secure channel phases and the *entire* protocol header is taken as the associated data authenticated by the underlying encryption scheme. Therefore, UDP+QUIC's SC Header Integrity can be reduced to its level-1 Channel Security. Formally, for any efficient adversary $\mathcal{A}$ there exists an efficient adversary B such that $\mathbf{Adv}^{\mathsf{int\text{-}h}}_{\mathrm{UDP+QUIC}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathsf{cs\text{-}1}}_{\mathrm{UDP+QUIC}}(\mathcal{B})$. This is because $\mathcal{B}$ can simulate $\mathcal{A}$'s security experiment perfectly by answering the Pack queries using $\mathsf{Encrypt}$ with same inputs and forwarding the $\mathsf{Deliver}$ queries to $\mathsf{Decrypt}$, and if $\mathcal{A}$ outputs (P, i, b') and wins then $\mathcal{B}$ knows the secret bit $b^i_P = b'$.

Reset Authentication. UDP+QUIC does not achieve this security notion because, similar to TCP

Reset, its reset packet PublicReset is not authenticated either. In the following availability attack, an efficient adversary $\mathcal{A}$ can always get $\mathbf{Adv}^{\mathsf{rst\text{-}auth}}_{\mathrm{UDP+QUIC}}(\mathcal{A}) = 1$:

PublicReset Attack. $\mathcal{A}$ can first make two session oracles complete a handshake using $\mathsf{Connect}, \mathsf{Send}$ queries, then use $\mathsf{Pack}, \mathsf{Deliver}$ queries to let them exchange secure channel packets. By observing these packet headers, $\mathcal{A}$ can easily forge a valid (plaintext) reset packet by setting its PUBLIC_FLAG_RESET bit to 1 and the remaining packet fields to reasonable values (which is easy because it simply contains the connection ID cid, the sequence number of the rejected packet, and a nonce to prevent replay). This attack will cause similar effects as described in the TCP Reset attack. Note that this vulnerability is fixed in QUIC[TLS] shown below.

5.3 QUIC[TLS] over UDP

1) Protocol Description: As mentioned in the Background, QUIC[TLS] replaces QUIC's key exchange with the TLS 1.3 key exchange. So, as with TLS 1.3, a UDP+QUIC[TLS] 2-RTT full handshake is a 2-stage msACCE protocol in the full mode and a 0-RTT resumption handshake is a 3-stage msACCE protocol in the resumption mode. The stage keys are set in the same way as in TLS 1.3.

The header fields (as specified in [28]) are similar to those in UDP+QUIC. Reset packets are defined as those whose first two header bits are 01. $\mathsf{scfg\_gen}$ is undefined. UDP+QUIC[TLS] also enforces different frame types for encrypted key exchange, secure channel, and pre-reset messages. For simplicity, we define $\mathcal{M}_{\mathsf{KE}}, \mathcal{M}_{\mathsf{SC}}, \mathcal{M}_{\mathsf{pRST}}$ as consisting of bit strings differing in their first two bits. UDP+QUIC[TLS]'s stateful encryption scheme is the same as $\mathsf{sAEAD}_{\mathrm{QUIC}}$ based on the underlying nonce-based AEAD scheme $\mathsf{AEAD} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$ (instantiated with AES-GCM [43] or others as documented in [53]).

QUIC[TLS] still provides source validation with a secure token generated by the server, similar to the case in Google's QUIC. We discuss QUIC[TLS]'s stateless reset mechanism later in the security analysis of Reset Authentication and refer to [28, 59] for the detailed UDP+QUIC[TLS] handshake messages and key generations.

2) Security Results: UDP+QUIC[TLS]'s session identifier $\mathtt{sid}_{\mathrm{QUIC[TLS]}}$ is defined as $\mathtt{sid}_{\mathrm{TLS}}$. By construction, UDP+QUIC[TLS] inherits the msACCE-std security from TLS 1.3 (but using QUIC's underlying encryption scheme). That is, it achieves level-1 Channel Security and implicitly achieves level-4 authentication as discussed before. UDP+QUIC[TLS] has a similar source-validation token scheme as QUIC. If the token is generated with an authenticated encryption scheme, the IP-Spoofing Prevention security of UDP+QUIC[TLS] can be reduced to the encryption scheme's authenticity security. However, such a source-validation scheme suffers from an availability attack against KE Payload Integrity similar to ServerReject Triggering for UDP+QUIC, where the adversary replaces the source-validation token with a random value to downgrade a 0-RTT resumption connection. As noted in [59], an adversary can also modify the unauthenticated ACK frames in the Initial packets without being detected. Furthermore, UDP+QUIC[TLS] achieves SC Header Integrity in the same way as UDP+QUIC. We are only left to show its security of KE Header Integrity and Reset Authentication.

KE Header Integrity. UDP+QUIC[TLS] does not achieve these security notions because its first-round Initial packets (see [28]) are not fully authenticated. For instance, an efficient adversary $\mathcal{A}$ can get $\mathbf{Adv}^{\mathsf{int\text{-}keh}}_{\mathrm{UDP+QUIC[TLS]}}(\mathcal{A}) = 1$ as follows. $\mathcal{A}$ first queries $\mathsf{Connect}(\pi^i_C, S)$ to get π^i_C's Initial packet $(\mathrm{IP}_C, \mathrm{IP}_S, H, pd)$. Then, as described in [59], $\mathcal{A}$ can decrypt this packet with its Destination Connection ID dcid in H, change it to another value $\mathtt{dcid}'$, and re-encrypt the whole packet with this new $\mathtt{dcid}'$. The resulting packet $(\mathrm{IP}_C, \mathrm{IP}_S, H', pd')$, where $H \neq H'$, is valid and will be accepted by a new server oracle π^j_S without being detected by the client. However, this is only a theoretical attack with no practical impact.

Reset Authentication. In UDP+QUIC[TLS], the stateless reset works as follows. One party generates a 128-bit reset token using its static key and a random 64-bit QUIC connection ID cid as input. Then this token (carried within the pre-reset message) is sent to the other party in a secure

channel phase. Later, the same party that generated this token can perform a stateless reset by regenerating the token and sending it to the other party in clear (via a reset packet).

The Reset Authentication security of UDP+QUIC[TLS] can be reduced to its level-1 Channel Security and the PRF security of the reset token generation function F (modeled as a PRF), as shown in the following theorem with the proof in Appendix H:

Theorem 2 *For any efficient adversary $\mathcal{A}$ delivering at most q_{rst} forged reset packets (via* Deliver *queries), there exist efficient adversaries $\mathcal{B}$ and $\mathcal{C}$ such that:*

$$\mathbf{Adv}^{\mathsf{rst\text{-}auth}}_{\text{UDP+QUIC[TLS]}}(\mathcal{A}) \;\leq\; |\mathcal{P}|\mathbf{Adv}^{\mathsf{prf}}_{F}(\mathcal{B}) + |\mathcal{P}|\mathbf{Adv}^{\mathsf{cs\text{-}1}}_{\text{UDP+QUIC[TLS]}}(\mathcal{C}) + \frac{q_{\mathsf{rst}}|\mathcal{P}|}{2^{128}} + \frac{|\mathcal{P}|N^2}{2^{|\mathtt{cid}|}},$$

where $|\mathcal{P}|$ is the number of parties and N is the maximum number of sessions for each party.

Remark. Recall that $|\mathtt{cid}| = 64$. Similar to the remark following Theorem 1, regarding the last probability term, any attacker has to observe roughly 2^{32} online sessions in order to find a `cid` collision. To forge (basically replay) a valid reset packet, the attacker has to further make roughly 2^{32} online $\mathsf{Pack}(\cdot,\cdot,\mathtt{rst})$ queries to the target server to retrieve its reset packets in those observed sessions,[18] which is an online attack that is easy to detect and suppress.

6 Conclusion

Our work is the first to provide a thorough, formal, and fine-grained security comparison of the most efficient secure channel establishment protocols on the market today. By including packet-level attacks in our analysis, our results shed light on how the reliability, flow control, and congestion control of TFO+TLS 1.3, UDP+QUIC, and UDP+QUIC[TLS] compare besides their basic security, in adversarial settings.

We found that availability functionalities provided by transport-layer protocols like TCP can be easily compromised without packet-level authentication, which may undermine the performance of their supporting application-layer protocols. To protect against availability attacks, new protocols should better implement and authenticate their own transport functionalities like QUIC does. Besides, the key exchange packet integrity should also be scrutinized to avoid serious undetectable availability attacks.

We hope that our model will help protocol designers in their future protocol analyses and that our results will help practitioners better understand the advantages and limitations of novel secure channel establishment protocols.

Acknowledgments

We thank the anonymous reviewers for their comments. This paper is based upon work supported by the National Science Foundation under Grant No. 1422794.

References

[1] HTTPS encryption on the web - Google transparency report, 2018.

[2] Josh Aas. Let's Encrypt: Looking forward to 2019, December 2018.

[3] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Cryptographers Track at the RSA Conference*, pages 143–158. Springer, 2001.

[18] If the target server can open 2^{32} concurrent sessions, then the attacker only needs to make a single online query. However, we assume in practice the server cannot open many concurrent connections since there are at most 65535 TCP sockets per IP address.

[4] Raz Abramov and Amir Herzberg. TCP Ack storm DoS attacks. In *IFIP International Information Security Conference*, pages 29–40, 2011.

[5] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):206–241, 2004.

[6] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO 1993*, pages 232–249. Springer, 1993.

[7] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy*, pages 483–502. IEEE, 2017.

[8] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO 2014*, pages 235–255. Springer, 2014.

[9] Colin Boyd and Britta Hale. Secure channels and termination: The last word on TLS. In *International Conference on Cryptology and Information Security in Latin America*, pages 44–65. Springer, 2017.

[10] Colin Boyd, Britta Hale, Stig Frode Mjølsnes, and Douglas Stebila. From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS. In *Cryptographers Track at the RSA Conference*, pages 55–71. Springer, 2016.

[11] Jacqueline Brendel, Marc Fischlin, and Felix Günther. Breakdown resilience of key exchange protocols: NewHope, TLS 1.3, and Hybrids. In *European Symposium on Research in Computer Security*, pages 521–541. Springer, 2019.

[12] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In *CRYPTO 2017*, pages 651–681. Springer, 2017.

[13] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *USENIX Security Symposium*, pages 209–225, 2016.

[14] Centre for the Protection of National Infrastructure. Security assessment of the Transmission Control Protocol. Technical Report CPNI Technical Note 3/2009, Centre for the Protection of National Infrastructure, 2009.

[15] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. RFC 7413, December 2014.

[16] C. Cremers, M. Horvat, S. Scott, and T. v. Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy*, pages 470–485, 2016.

[17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788. ACM, 2017.

[18] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482. IEEE Computer Society, 2017.

[19] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *2015 ACM SIGSAC Conference on Computer and Communications Security*, pages 1197–1210. ACM, 2015.

[20] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. `https://eprint.iacr.org/2016/081`.

[21] Benjamin James Dowling. *Provable security of internet protocols*. PhD thesis, Queensland University of Technology, 2017.

[22] 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*, pages 60–75. IEEE, 2017.

[23] M. Fischlin, F. Günther, G. Azzurra Marson, and K. G Paterson. Data is a stream: Security of stream-based channels. In *CRYPTO 2015*, pages 545–564. Springer, 2015.

[24] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In *2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1193–1204. ACM, 2014.

[25] Gennie Gebhart. Tipping the scales on HTTPS: 2017 in review, December 2017.

[26] Yossi Gilad and Amir Herzberg. Off-path attacking the web. In *WOOT 2012*, pages 41–52, 2012.

[27] Felix Günther and Sogol Mazaheri. A formal treatment of multi-key channels. In *CRYPTO 2017*, pages 587–618. Springer, 2017.

[28] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport, November 2019.

[29] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, pages 273–293. Springer, 2012.

[30] S. Jero, H. Lee, and C. Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2015.

[31] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and C. Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[32] Laurent Joncheray. A simple active attack against TCP. In *USENIX Security Symposium*, 1995.

[33] Tadayoshi Kohno, Adriana Palacio, and John Black. Building secure cryptographic transforms, or how to encrypt and mac. 2003. `https://eprint.iacr.org/2003/177`.

[34] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO 2010*, pages 631–648. Springer, 2010.

[35] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO 2013*, pages 429–448. Springer, 2013.

[36] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 81–96. IEEE, 2016.

[37] V. A. Kumar, P. S. Jayalekshmy, G. K. Patra, and R. P. Thangavelu. On remote exploitation of TCP sender for low-rate flooding denial-of-service attack. *IEEE Communications Letters*, 13(1):46–48, 2009.

[38] Aleksandar Kuzmanovic and Edward Knightly. Low-rate TCP-targeted denial of service attacks and counter strategies. *IEEE/ACM Transactions on Networking*, 14(4):683–696, 2006.

[39] A Langley and W Chang. QUIC crypto, 2016.

[40] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. Multiple handshakes security of TLS 1.3 candidates. In *2016 IEEE Symposium on Security and Privacy*, pages 486–505. IEEE, 2016.

[41] Greg Linden. Make data useful, 2006.

[42] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru. How secure and quick is QUIC? provable security and performance analyses. In *2015 IEEE Symposium on Security and Privacy*, pages 214–231, 2015.

[43] David A McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In *International Conference on Cryptology in India*, pages 343–355. Springer, 2004.

[44] Alfred Menezes and Berkant Ustaoglu. Comparing the pre-and post-specified peer models for key agreement. In *Australasian Conference on Information Security and Privacy*, pages 53–68. Springer, 2008.

[45] Robert Morris. A weakness in the 4.2 BSD Unix TCP/IP software. Technical report, AT&T Bell Leboratories, 1985.

[46] Kenneth G Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 372–389. Springer, 2011.

[47] Christopher Patton and Thomas Shrimpton. Partially specified channels: The TLS 1.3 record layer without elision. In *2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1415–1428. ACM, 2018.

[48] Jon Postel. User Datagram Protocol. RFC 768 (Standard), 1980.

[49] Jon Postel. Transmission Control Protocol. RFC 793 (Standard), 1981.

[50] Zhiyun Qian and Z. Morley Mao. Off-path TCP sequence number inference attack - how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361, 2012.

[51] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *2012 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2012.

[52] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 21. ACM, 2011.

[53] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[54] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM, 2002.

[55] J. Roskind. QUIC(Quick UDP Internet Connections): Multiplexed stream transport over UDP. *Technical report, Google*, 2013.

[56] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review*, 29(5), 1999.

[57] Ahren Studer and Adrian Perrig. The Coremelt attack. In *European Symposium on Research in Computer Security*, pages 37–52, 2009.

[58] Ian Swett. QUIC deployment experience @Google. `https://www.ietf.org/proceedings/96/slides/slides-96-{QUIC}-3.pdf`, 2016.

[59] M. Thomson and S. Turner. Using Transport Layer Security (TLS) to secure QUIC, November 2019.

[60] Verizon Enterprise Solutions. IP latency statistics — Verizon Enterprise Solutions, 2018.

[61] Paul Watson. Slipping in the window: TCP reset attacks. Technical report, CanSecWest, 2004.

A Transport Protocol Headers

Fig. 4 presents the TCP header while Fig. 5 presents the UDP header.

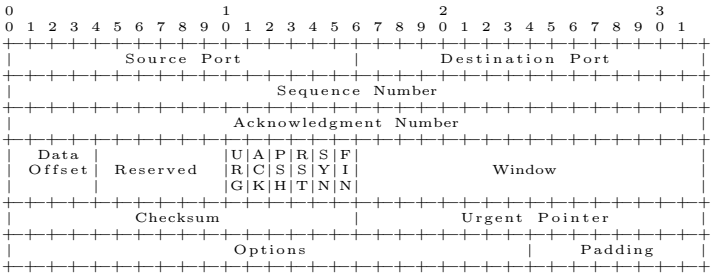

Figure 4: TCP header. [49]

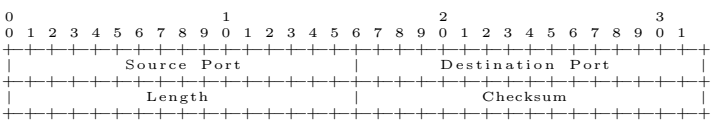

Figure 5: UDP header. [48]

B Preliminary Definitions

B.1 Pseudorandom Functions

For a function family $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^m$, consider the following security experiment associated with an adversary $\mathcal{A}$. In the beginning, sample a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, $\mathcal{A}$ is given oracle access, i.e., can make queries, to $F_k(\cdot) = F(k, \cdot)$ where $k \xleftarrow{\$} \{0,1\}^\lambda$. If $b = 1$, $\mathcal{A}$ is given oracle access to $f(\cdot)$ that maps elements from $\{0,1\}^n$ to $\{0,1\}^m$ uniformly at random. In the end, $\mathcal{A}$ outputs a bit b' as a guess of $\mathcal{B}$. The advantage of $\mathcal{A}$ is defined as $\mathbf{Adv}^{\mathsf{prf}}_F(\mathcal{A}) = |\Pr[b' = 1|b = 0] - \Pr[b' = 1|b = 1]|$, which measures $\mathcal{A}$'s ability to distinguish F_k (with random k) from a random function f.

F is a *pseudorandom function (PRF)* if:

- For any $k \in \{0,1\}^\lambda$ and $x \in \{0,1\}^n$, there exists a polynomial-time (in λ) algorithm to compute $F(k, x)$; and
- For any efficient adversary $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{prf}}_F(\mathcal{A})$ is sufficiently small (e.g., roughly $2^{-\lambda}$).

B.2 Stateful Authenticated Encryption with Associated Data

We follow [33, 10] in extending the stateful authenticated encyption notion of Bellare *et al.* [5] to capture a hierarchy of stateful AEAD security notions based on different authentication levels. The following definitions are the same as [10], except that we exclude the length-hiding property proposed by Paterson *et al.* [46] for conciseness.

Syntax. A stateful AEAD scheme sAEAD is a 4-tuple $(\mathsf{sG}, \mathsf{sI}, \mathsf{sE}, \mathsf{sD})$ associated with a message space $\mathcal{M} \subseteq \{0,1\}^*$, an associated data space $\mathcal{AD} \subseteq \{0,1\}^*$, and a state space $\mathcal{ST} \subseteq \{0,1\}^*$. sG is a probabilistic algorithm that samples a random key k from a finite non-empty key space $\mathcal{K}$. sI is an algorithm that initializes the encryption and decryption states st_e, st_d. sE is a probabilistic encryption algorithm that takes as input $k \in \mathcal{K}, ad \in \mathcal{AD}, m \in \mathcal{M}$ and st_e and outputs a ciphertext $ct \in \{0,1\}^*$ with an updated st_e. sD is a deterministic decryption algorithm that takes as input $k \in \mathcal{K}, ad \in \mathcal{AD}, ct \in \{0,1\}^*$ and st_d and outputs $m \in \mathcal{M} \cup \{\perp\}$ with an updated st_d. The *correctness* requires that, for any $k \in \mathcal{K}$ sampled by sG, any $st_e = st^0_e, st_d = st^0_d$ initialized by sI, and any sequence of encryptions $\{(ct_{i+1}, st^{i+1}_e) \xleftarrow{\$} \mathsf{sE}(k, ad_i, m_i, st^i_e)\}_{i\geq 0}$, the sequence of decryptions $\{(m'_{i+1}, st^{i+1}_d) \leftarrow \mathsf{sD}(k, ad, \mathsf{E}(k, ad_i, ct_i, st^i_d))\}_{i\geq 0}$ satisfies $m_i = m'_i, i \geq 0$.

Security. Consider the following experiment with an authentication level $al \in [4]$. In the beginning, run sG to generate a key k and run sI to initialize st_e, st_d. Sample $b \xleftarrow{\$} \{0,1\}$ and set $(u, v, \mathtt{outofsync}) \leftarrow (0, 0, 0)$. Then, the adversary $\mathcal{A}$ is given access to the following oracles:

$\underline{\mathsf{Enc}(ad, m_0, m_1)}$:

1: $u \leftarrow u + 1$, $(sent.ct_u, st'_e) \xleftarrow{\$} \mathsf{sE}(k, ad, m_b, st_e)$
2: $(sent.ad_u, st_e) \leftarrow (ad, st'_e)$, return $sent.ct_u$

$\underline{\mathsf{Dec}(ad, ct)}$:

1: if $b = 0$, return $\perp$
2: $v \leftarrow v + 1$, $(m, st'_d) \leftarrow \mathsf{sD}(k, ad, ct, st_d)$
3: $(rcvd.ad_v, st_d) \leftarrow (ad, st'_d)$
4: if $(al = 4) \wedge \mathsf{cond}_4$ or $(al \leq 3) \wedge (m \neq \perp) \wedge \mathsf{cond}_{al}$,[19]
set $\mathtt{outofsync} \leftarrow 1$
5: if $\mathtt{outofsync} = 1$, return m, otherwise, return $\perp$

In the end, $\mathcal{A}$ outputs a bit b'. The stateful AEAD scheme sAEAD is *secure* with authentication level al if and only if $\mathbf{Adv}^{\mathsf{aead}\text{-}al}_{\mathsf{sAEAD}}(\mathcal{A}) = |2\Pr[b = b'] - 1|$ is sufficiently small (e.g., roughly $2^{-\log|\mathcal{K}|}$) for any efficient adversary $\mathcal{A}$.

[19] Authentication conditions cond_{al} are defined in the same way as in the msACCE-std Decrypt query.

C QUIC and TLS 1.3's Stateful AEAD Schemes and Their Security

C.1 QUIC's Stateful AEAD Scheme and its Security

First, we show QUIC's stateful encryption scheme $\mathsf{sAEAD}_{\mathrm{QUIC}}$ constructed from a nonce-based AEAD scheme $\mathsf{AEAD} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$ as follows.

$\underline{\mathsf{sG}()}$:
  $k_e \xleftarrow{\$} \mathsf{G}(), k_m \xleftarrow{\$} \{0,1\}^{32}$
  return (k_e, k_m)
$\underline{\mathsf{sI}()}$:
  return $(st_e, st_d) \leftarrow (\varnothing, \perp)$
$\underline{\mathsf{sD}(k, ad, ct, st_d)}$:
  $(k_e, k_m) \leftarrow k$, $(\mathtt{cid}, \mathtt{sqn}) \leftarrow ad$
  $m \leftarrow \mathsf{D}(k_e, k_m \| \mathtt{sqn}, ad, ct)$
  return $(m, \perp)$

$\underline{\mathsf{sE}(k, ad, m, st_e)}$:
  $(k_e, k_m) \leftarrow k$, $(\mathtt{cid}, \mathtt{sqn}) \leftarrow ad$
  if $\mathtt{sqn} \in st_e$,
    return $(\perp, st_e)$
  $c \leftarrow \mathsf{E}(k_e, k_m \| \mathtt{sqn}, ad, m)$
  $st_e \leftarrow st_e \cup \{\mathtt{sqn}\}$
  return (c, st_e)

Note that $\mathsf{sAEAD}_{\mathrm{QUIC}}$ uses the encryption state to keep track of used nonces to avoid repeating and the decryption state is not used.

To reduce $\mathsf{sAEAD}_{\mathrm{QUIC}}$'s level-1 AEAD security to the underlying AEAD's nonce-based AEAD security, we first recall that the nonce-based AEAD security is defined as two separate parts, privacy and authenticity. For privacy, the adversary guesses the secret bit of a left-or-right encryption oracle but cannot make queries with a repeated nonce. The associated advantage is denoted by $\mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AEAD}}(\mathcal{A})$. For authenticity, the adversary tries to forge a valid ciphertext (together with a nonce and an associated data), given an encryption oracle (without the secret bit). The associated advantage is denoted by $\mathbf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathsf{AEAD}}(\mathcal{A})$. Now, we are ready to prove the following theorem.

Theorem 3 *For any efficient adversary $\mathcal{A}$, there exist efficient adversaries $\mathcal{B}$ and $\mathcal{C}$ such that:*

$$\mathbf{Adv}^{\mathsf{aead\text{-}1}}_{\mathsf{sAEAD}_{\mathrm{QUIC}}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathsf{AEAD}}(\mathcal{B}) + \mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AEAD}}(\mathcal{C}) \ .$$

Proof: Consider two games G_0 and G_1. G_0 is the real experiment for $\mathcal{A}$ and G_1 is the same as G_0 except that it will always return $\perp$ for Dec queries. Denote $\Pr_i$ as the advantage of $\mathcal{A}$ in G_i. $|\Pr_0 - \Pr_1|$ is bounded by the probability that $\mathcal{A}$ forges a new valid ciphertext given $b = 1$, which by definition is bounded by $\mathbf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathsf{AEAD}}(\mathcal{B})$ for some efficient adversary $\mathcal{B}$. Then, note that according to the $\mathsf{sAEAD}_{\mathrm{QUIC}}$ construction nonces in AEAD encryption queries never repeat and G_1 can be simulated by an efficient adversary $\mathcal{C}$ against the nonce-based AEAD privacy security, which implies $\Pr_1 \leq \mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AEAD}}(\mathcal{C})$. Therefore, we have $\mathbf{Adv}^{\mathsf{aead\text{-}1}}_{\mathsf{sAEAD}_{\mathrm{QUIC}}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathsf{AEAD}}(\mathcal{B}) + \mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AEAD}}(\mathcal{C})$. □

Remark. It is not hard to see that level-1 AEAD security is the best $\mathsf{sAEAD}_{\mathrm{QUIC}}$ can achieve. Consider an adversary $\mathcal{A}$ against the level-al ($al > 1$) AEAD security of $\mathsf{sAEAD}_{\mathrm{QUIC}}$. $\mathcal{A}$ can easily set $\mathtt{outofsync} \leftarrow 1$ by querying Dec twice with the same ciphertext output by a previous Enc query, since $\mathsf{sAEAD}_{\mathrm{QUIC}}$ does not prevent replays.

C.2 TLS 1.3's Stateful AEAD Scheme and its Security

Next, we show TLS 1.3's stateful encryption scheme $\mathsf{sAEAD}_{\mathrm{TLS}}$ constructed from a nonce-based AEAD scheme $\mathsf{AEAD} = (\mathsf{G}, \mathsf{E}, \mathsf{D})$ as follows:

$\underline{\mathsf{sG}()}$:
$k_e \xleftarrow{\$} \mathsf{G}(), k_m \xleftarrow{\$} \{0,1\}^n$
return (k_e, k_m)

$\underline{\mathsf{sI}()}$:
return $(st_e, st_d) \leftarrow (0, 0)$

$\underline{\mathsf{sE}(k, ad, m, st_e)}$:
$(k_e, k_m) \leftarrow k$
$c \leftarrow \mathsf{E}(k_e, k_m \oplus st_e, ad, m)$, $st_e \leftarrow st_e + 1$
return (c, st_e)

$\underline{\mathsf{sD}(k, ad, ct, st_d)}$:
if $st_d = \perp$,
return $(\perp, \perp)$
$(k_e, k_m) \leftarrow k$
$m \leftarrow \mathsf{D}(k_e, k_m \oplus st_d, ad, ct)$
if $m = \perp$,
$st_d \leftarrow \perp$
otherwise,
$st_d \leftarrow st_d + 1$
return (m, st_d)

Note that in the above TLS's stateful encryption scheme, nonce repeating is prevented by the increasing counter kept by the encryption state st_e. Following a very similar argument as in the above proof of Theorem 3, one can show that the level-4 AEAD security of $\mathsf{sAEAD}_{\mathrm{TLS}}$ is also reduced to the nonce-based AEAD security of AEAD. This result has been proved by previous work (Theorem 3 in [18]), but their stateful AEAD security definition is slightly different from ours. For instance, in their game the adversary needs to distinguish ciphertexts from random, while in our game the adversary distinguishes ciphertexts of two messages.

D TFO+TLS 1.3's msACCE-std Security

Due to the high similarity among the abundant TLS 1.3 proofs in the MSKE model (and its extensions) and a security proof in our msACCE-std model, we only provide a proof sketch below.

Previous works [21] and [22] respectively proved that the TLS 1.3 draft-16 (EC)DHE full handshake and draft-14 PSK-(EC)DHE 0-RTT resumption handshake are secure in the MSKE model based on the collision resistance of the hash function, unforgeability of the signature and MAC schemes, PRF security of the key derivation function, and pseudorandom function oracle Diffie-Hellman (PRF-ODH) assumption [29, 35, 12]. Their MSKE security, which captures only the key exchange phases, ensures the Bellare-Rogaway-style key secrecy [6] (i.e., the stage keys are indistinguishable from random ones) with various authentication properties (for which our msACCE-std model focuses on the unilateral server authentication). These results derived the overall TLS 1.3 security using a *compositional* approach, i.e., composing a secure key exchange protocol (e.g., the TLS 1.3 handshake protocol) in the MSKE model with an arbitrary secure symmetric key protocol (e.g., the TLS 1.3 record protocol). However, as stated in [22], this generic composition result only works for key-independent, forward-secret, external, and non-replayable stage keys. In particular, it does not apply to the final session keys in full handshakes or the interim handshake keys because they are used internally in the key exchange phases. Besides, it does not apply to the 0-RTT keys, which are replayable and non-forward-secret. In order to adjust their security results to prove TLS 1.3's Server Authentication and level-4 Channel Security in our model, we need to address a few TLS 1.3 updates and model differences as follows.

First, we outline why the security results in [21, 22] for old TLS 1.3 drafts can be extended to the final standard TLS 1.3 [53], i.e., the standard TLS 1.3 (EC)DHE full handshake and PSK-(EC)DHE 0-RTT handshake are secure in the MSKE model.

1) The multi-stage key generation procedures are updated in the just-released TLS 1.3. Recall that TLS 1.3 performs key derivation in the extract-then-expand paradigm [34] using the HMAC-based Extract-and-Expand Key Derivation Function (HKDF), which consists of two functions HKDF.Extract and HKDF.Expand. In particular, it first extracts an internal secret (e.g., early secret, handshake secret, and master secret), then expands it (twice) to derive the corresponding stage key. The latest standard TLS 1.3 performs an expand-then-extract procedure instead of a single extract procedure for the extraction of the handshake secret and master secret. However, these two additional expand steps do not affect the MSKE security because they only add a constant (single) query to HKDF.Expand, leading to a larger constant for its PRF advantage. Besides, compared to [21], such extra expand

steps help TLS 1.3's MSKE security no longer rely on the PRF security of the underlying HMAC primitive of both HKDF functions.

2) The message flows of the PSK-(EC)DHE 0-RTT resumption handshake are updated in the just-released TLS 1.3. The 0-RTT `Finished` message is replaced by a pre-shared key (PSK) binder. They are both HMAC values generated with very similar procedures and have the same purpose, i.e., to authenticate the `ClientHello` message and to bind the current resumption session with the assoicated full session. Such a replacement does not affect the TLS 1.3's MSKE security. Besides, a new `EndOfEarlyData` message is added as an indicator to end 0-RTT data transmission. This is an empty handshake message independent of key generation so does not affect the security either.

Then, based on the above extended TLS 1.3 MSKE security, one can apply the security results in [40] to get the Multi-Level&Stage security of the combination of the TLS 1.3 full handshake and 0-RTT resumption handshake. Referring to their notions [40], our msACCE-std model focuses only on two modes, i.e., the (EC)DHE full handshake and PSK-(EC)DHE 0-RTT resumption handshake, and two levels, i.e., one level of full handshakes followed by one level of 0-RTT resumption handshakes.

Finally, we outline why the above TLS 1.3 security result in the Multi-Level&Stage model [40] can be augmented to prove TLS 1.3's Server Authentication and level-4 Channel Security.

1) The above security result guarantees server authentication, i.e., a client oracle that has set its final session key must share the same session identifier with a unique partner server oracle. However, their session identifier is defined as *unencrypted* key exchange messages in order to capture key independence (i.e., revealing independent stage keys in the same session does not break the unrevealed stage key's secrecy). We instead use a "real" encrypted session identifier to simplify our model and make reducing KE Payload Integrity to Server Authentication easy. (Note that an unencrypted session identifier may correspond to many valid encrypted session identifiers but KE Payload Integrity requires no modification in the encrypted payload). To prove Server Authentication, we need to follow their proof of the TLS 1.3 Multi-Level&Stage server authentication to replace handshake keys with independent and random values, then use $\mathsf{sAEAD}_{\mathrm{TLS}}$'s AEAD oracles to simulate encrypted key exchange messages in $\mathtt{sid}_{\mathrm{TLS}}$ and the decryption of them. In this way, Server Authentication can be reduced to the TLS 1.3 Multi-Level&Stage server authentication and the AEAD security.

2) To prove level-4 Channel Security, we follow their proof of the TLS 1.3 Multi-Level&Stage security to replace all stage keys with independent and random values and then use the AEAD oracles to simulate encrypted key exchange messages and $\mathsf{Encrypt}, \mathsf{Decrypt}$ queries. In this way, level-4 Channel Security can be reduced to the TLS 1.3 Multi-Level&Stage security and the level-4 AEAD security of $\mathsf{sAEAD}_{\mathrm{TLS}}$. Note that the AEAD oracles are also used to simulate post-handshake messages like `NewSessionTicket`. This bypasses the composition issue [20] faced by the MSKE model (and its extensions), in which the application keys in full handshakes cannot be composed with secure symmetric key protocols because these keys are used internally in the key exchange phase to encrypt `NewSessionTicket` messages.

E Proof of Theorem 1

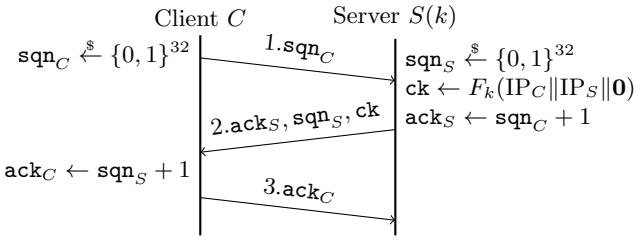

Figure 6: TFO initial connection.

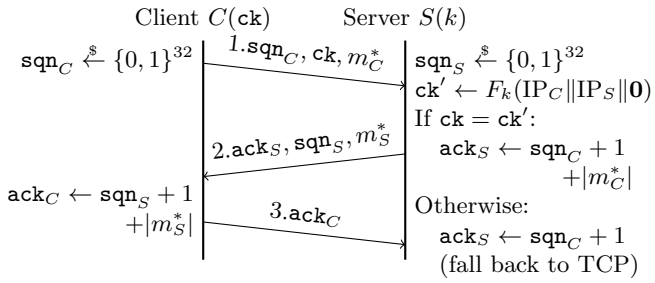

Figure 7: TFO 0-RTT resumption connection. * indicates optional messages.

The TFO protocol specifications are shown in Fig. 6 and Fig. 7, where the server takes a cookie generation key $k \xleftarrow{\$} \{0,1\}^{128}$ as input.

Proof: Consider a sequence of games (i.e., experiments) and let $\Pr_i, i \geq 0$ denote the winning probability of $\mathcal{A}$ in **Game** i.

Game 0: This is the real experiment for $\mathcal{A}$, so $\Pr_0 = \mathbf{Adv}^{\mathsf{ipsp}}_{\text{TFO+TLS 1.3}}(\mathcal{A})$.

Game 1: The challenger proceeds as before except it first guesses the target server S that accepts a spurious connection request and aborts if the guess is wrong. Since the probability of a correct guess is at least $1/|\mathcal{S}|$, we have $\Pr_0 \leq |\mathcal{S}| \Pr_1$.

Game 2: The challenger proceeds as before except it replaces the PRF F used by S for TFO cookie generation with a truly random function f. By the PRF definition, there exists an efficient adversary $\mathcal{B}$ such that $|\Pr_1 - \Pr_2| \leq \mathbf{Adv}^{\mathsf{prf}}_F(\mathcal{B})$.

Now, in Game 2, the TFO cookies generated by S are independent from each other for each client. We can bound $\Pr_2$ by considering two cases. 1) $\mathcal{A}$ wins by sending a valid ACK packet in a full session. In this case, $\mathcal{A}$ must have generated a valid $\mathtt{ack}_C$ by correctly guessing the target server's TCP sequence number $\mathtt{sqn}_S$. The winning probability of each guess is exactly $1/2^{|\mathtt{sqn}|}$. 2) $\mathcal{A}$ wins by sending a valid SYN packet in a resumption session. In this case, $\mathcal{A}$ must have forged a valid TFO cookie $\mathtt{ck} \in \{0,1\}^{128}$. The winning probability of each forgery is exactly $1/2^{128}$ because the TFO cookie generation function is a truly random function. By applying a union bound on the q_{S} queries and noting that $|\mathtt{sqn}| = 32 < 128$, we have $\Pr_2 \leq q_{\mathsf{S}} / \min\{2^{|\mathtt{sqn}|}, 2^{128}\} = q_{\mathsf{S}}/2^{|\mathtt{sqn}|}$. □

F TCP Attacks

TCP Flow Control Manipulation. An adversary with access to the communication channel can impact TCP's flow control mechanism to decrease the sending rate or stall the connection by modifying TCP's `window` header field. This field controls the amount of received data the sender of this packet is prepared to buffer. By reducing this quantity, the throughput of the connection can be reduced and if it is set to zero the connection will completely stall.

One example of this attack would be to modify the window field to zero in a TCP packet containing a TLS-encrypted HTTP request. Since TCP headers are not authenticated, this modification will not be detected. As a result, when the server receives this request and attempts to send the response, it will believe that the client cannot currently accept any data and will delay sending the response. After some timeout, TCP will probe the client with a single packet of data to determine whether the window is still zero. If the adversary also modifies the responses to these probes, the connection will remain stalled indefinitely; otherwise, the connection will eventually recover after a lengthy delay.

TCP Acknowledgment Injection. An adversary who can observe a target connection and forge packets can inject new acknowledgment packets into the TCP connection. Acknowledgment packets have

no data making them undetectable by either TLS or the application. However, they are used by congestion control to determine the allowed sending rate of a connection.

Injecting duplicate or very slowly increasing acknowledgments can be used to slow a target connection down drastically. [31] demonstrated a 12x reduction in throughput using this approach with the attacker required to expend only 40Kbps. This, of course, represents a significant performance degradation for a TFO+TLS 1.3 connection.

Injecting acknowledgments can also be used to dramatically increase the sending rate of a connection, turning it into a firehose that an attacker can point at their desired target. This is done by sending acknowledgments for data that has not been received yet, an attack known as Optimistic Ack [56]. This attack renders TCP insensitive to congestion and can completely starve competing flows. It could be used with great effect to cause denial of service against a server or the Internet infrastructure as a whole [57].

G UDP+QUIC's msACCE-std Security

It has been proven in [42] that QUIC is QACCE-secure in the random oracle model based on the unforgeability of the signature scheme, the computational Diffie-Hellman (DH) assumption [3], and the nonce-based AEAD security. Note that msACCE-std with $\mathsf{sAEAD}_{\mathrm{QUIC}}$ is semantically equivalent to QACCE with nonce-based AEAD and $\mathsf{get\_iv}$ (defined in [42]), so their QACCE security results can be easily adapted to show that UDP+QUIC achieves Server Authentication and level-1 Channel Security in our msACCE-std model. Note that one can also prove UDP+QUIC's msACCE-std security relying on the level-1 AEAD security of $\mathsf{sAEAD}_{\mathrm{QUIC}}$ instead of the nonce-based AEAD security of the underlying AEAD, where the former can be reduced to the latter as shown in Appendix C.

H Proof of Theorem 2

Proof: Consider a sequence of games (i.e., experiments) and let $\Pr_i, i \geq 0$ denote the winning probability of $\mathcal{A}$ in **Game** i.

Game 0: This is the real experiment for $\mathcal{A}$, so $\Pr_0 = \mathbf{Adv}^{\mathsf{rst\text{-}auth}}_{\mathrm{UDP+QUIC[TLS]}}(\mathcal{A})$.

Game 1: The challenger proceeds as before except it aborts if the connection IDs repeat for a party. Since the probability of `cid` collision for each party is at most $N^2/2^{|\mathtt{cid}|}$, by a union bound we have $|\Pr_0 - \Pr_1| \leq |\mathcal{P}|N^2/2^{|\mathtt{cid}|}$.

Game 2: The challenger proceeds as before except it first guesses the target party P that accepts a spurious reset packet and aborts if the guess is wrong. Since the probability of a correct guess is at least $1/|\mathcal{P}|$, we have $\Pr_1 \leq |\mathcal{P}| \Pr_2$.

Game 3: The challenger proceeds as before except it replaces the PRF F used by P for reset token generation with a truly random function f. By the PRF definition, there exists an efficient adversary $\mathcal{B}$ such that $|\Pr_2 - \Pr_3| \leq \mathbf{Adv}^{\mathsf{prf}}_{F}(\mathcal{B})$.

Game 4: The challenger proceeds as before except it replaces the encrypted pre-reset messages (via $\mathsf{Pack}(\cdot,\cdot,\mathtt{prst})$ queries) with encryption of independent random pre-reset messages. There exists an efficient adversary $\mathcal{C}$ against the Channel Security such that $|\Pr_3 - \Pr_4| \leq \mathbf{Adv}^{\mathsf{cs\text{-}1}}_{\mathrm{UDP+QUIC[TLS]}}(\mathcal{C})$.

$\mathcal{C}$ can simulate Pack and $\mathsf{Deliver}$ queries with $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$ queries. For a $\mathsf{Pack}(\cdot,\cdot,\mathtt{prst})$ query, $\mathcal{C}$ generates two random reset tokens $x, y \xleftarrow{\$} \{0,1\}^{128}$ and uses them to construct two pre-reset messages. Then, $\mathcal{C}$ queries $\mathsf{Encrypt}$ with these pre-reset messages as challenge messages, uses the output ciphertext to form a pre-reset packet, and sends it to $\mathcal{A}$. For a $\mathsf{Pack}(\cdot,\cdot,\mathtt{rst})$ query, $\mathcal{C}$ returns the pre-reset message constructed from x. This perfectly simulates Game 3 in the left world or Game 4 in the right world. $\mathcal{C}$ outputs 1 if and only if $\mathcal{A}$ wins.

Now, in Game 4, the random pre-reset messages are independent from $\mathcal{A}$'s view and each guess is correct with probability $1/2^{128}$. By a union bound, we have $\Pr_4 \leq q_{\mathsf{rst}}/2^{128}$. □