2015 IEEE Symposium on Security and Privacy

How Secure and Quick is QUIC? Provable Security and Performance Analyses

Robert Lychev
MIT Lincoln Laboratory
robert.lychev@ll.mit.edu

Samuel Jero
Purdue University
sjero@purdue.edu

Alexandra Boldyreva
Georgia Institute of Technology
sasha@gatech.edu

Cristina Nita-Rotaru
Purdue University
cnitarot@purdue.edu

***Abstract*—QUIC is a secure transport protocol developed by Google and implemented in Chrome in 2013, currently representing one of the most promising solutions to decreasing latency while intending to provide security properties similar with TLS. In this work we shed some light on QUIC's strengths and weaknesses in terms of its provable security and performance guarantees in the presence of attackers. We first introduce a security model for analyzing performance-driven protocols like QUIC and prove that QUIC satisfies our definition under reasonable assumptions on the protocol's building blocks. However, we find that QUIC does not satisfy the traditional notion of forward secrecy that is provided by some modes of TLS, e.g., TLS-DHE. Our analyses also reveal that with simple bit-flipping and replay attacks on some public parameters exchanged during the handshake, an adversary could easily prevent QUIC from achieving minimal latency advantages either by having it fall back to TCP or by causing the client and server to have an inconsistent view of their handshake leading to a failure to complete the connection. We have implemented these attacks and demonstrated that they are practical. Our results suggest that QUIC's security weaknesses are introduced by the very mechanisms used to reduce latency, which highlights the seemingly inherent trade off between minimizing latency and providing 'good' security guarantees.**

I. Introduction

The proliferation of mobile and web applications and their performance requirements have exposed the limitations of current transport protocols. Specifically, protocols like TLS [1] have a relatively high connection establishment latency overhead, causing user unhappiness and often resulting in a decreased number of customers and financial losses. As a result, several efforts [2], [3], [4], [5] have gone into designing new transport protocols that have low latency as one of the major design goals, in addition to basic security goals such as confidentiality, authentication, and integrity.

One of the most promising protocols is QUIC [2], a secure transport protocol developed by Google and implemented in Chrome in 2013 [6]. QUIC integrates ideas from TCP, TLS, and DTLS [7] in order to provide security functionality comparable to TLS, congestion control comparable with TCP, as well as minimal round-trip costs during setup/resumption and in response to packet loss. Some of the major design differences from TLS are not relying on TCP in order to eliminate redundant communication and the use of initial keys to achieve faster connection establishment. However, the exact security and performance advantages and disadvantages of QUIC are not clear when compared to existing protocols such as TLS and DTLS. Shedding light on this problem is the main focus of our work.

The way to assess and compare security is by providing a provable security analysis. However, while the importance of provable security analysis for practical protocols is gaining wider acceptance, it is still common for a protocol to be deployed first and analyzed later. For example, the complete provable security results for TLS remained elusive for many years and have only recently been identified [8], [9], [10]. Not surprisingly, no formal guarantees of the provided services existed for QUIC, with the exception of informal arguments in its design specifications, before the recent (concurrent) work by Fischlin and Günter [11] and our work. Even though [11] assesses QUIC's security, its results do not overlap with ours and its analysis is limited to the key exchanges rather than the entire protocol. We provide a detailed comparison of these works in Section II.

Furthermore, even if certain security properties about a protocol may be proven to hold, its usefulness in real-life deployments can be undermined by attacks that prevent connections from being established in the first place, especially in ways that are more subtle than just dropping traffic, e.g. TCP reset attacks against TLS. Such types of attacks have gained notoriety since it became known that they have been used for censorship by some governments to deter users from viewing certain content on the Internet [12].

As QUIC has been deployed widely among Google servers, and may eventually be deployed outside of Google, it is critical to provide its provable security analysis and to understand its performance guarantees in the presence of attackers before it becomes more widely used. Understanding its performance guarantees is particularly important considering that QUIC is envisioned mainly for web content delivery and mobile applications.

Our contributions. We provide the provable-security analysis of QUIC and assess its performance guarantees in

Robert Lychev did most of his work on this paper while at Georgia Institute of Technology.

DOI 10.1109/SP.2015.21

the presence of attackers. Our study is suitable for a general class of performance-driven communication protocols that employ an initial session key to enable data exchange even before the final session key is set. We call such protocols Quick Communications (QC) protocols. While QUIC is our main focus, the recently announced version 1.3 of TLS [13] also fits the QC framework.

One of our major contributions is the security model for QC protocols. We designed a new model since the existing security definitions were unsuitable. The Authenticated and Confidential Channel Establishment (ACCE) model [8], [9] which was used in proving TLS did not fit QUIC for several reasons. First, TLS and its security model use one session key, while QUIC uses two, and the data may start being encrypted before the final session key is set. Therefore, the model has to deal with key and data exchange under multiple keys. Second, QUIC does not run on top of TCP and implements many of the features provided by TCP itself. This is done primarily for performance reasons, but QUIC also adds some cryptographic protection, such as protection against IP spoofing and packet re-ordering. Hence, it makes sense to model additional attacks such as IP spoofing or packet re-ordering. Also, we cannot analyze the key and data exchange phases separately using the established security definitions and then compose them to get a composition result implying the security of the whole protocol, because in QUIC these phases use common parameters (such as IV) and can overlap (data can be exchanged while the final session key is being set).

Our security definition is an extension of the ACCE definition to fit QC protocols. We call our model QACCE for Quick ACCE. We consider a very powerful attacker who knows all servers' public keys, can initiate and observe the communications between honest parties, and can intercept, drop, misorder, or modify the contents of the exchanged packets. We also consider DoS attacks such as IP spoofing. The adversary can adaptively corrupt servers and learn their (long-term) secret keys and secret states. It can also, again adaptively, learn parties' initial and final session keys. The adversary can also have partial knowledge of the data being exchanged by the parties. Given such strong adversarial capabilities, the attacker should not be able to prevent the parties from establishing session keys (without the parties noticing that something went wrong) and using these keys to achieve data exchange with privacy and integrity. We note that the sender authentication can only be achieved one-way, as only servers hold public keys.

Our security model formally captures the different levels of security guaranteed for data encrypted under the initial and final session keys.[1] While the attacker can cause honest parties to agree on distinct initial keys (something which is not possible in TLS), we still require that data exchanged under either key is protected. For the final session keys, the security requirement is similar to that for session keys in TLS: if one party sets the key, it is guaranteed that the other party sets the same key, and moreover, that the key is "good enough" to securely exchange data. Finally, we also consider forward secrecy. Unlike TLS-RSA, currently the most commonly deployed mode of TLS, QUIC provides certain forward secrecy guarantees such that corrupting a server during one time period does not let the attacker break the security of the data sent in previous time periods. On the other hand, because the initial keys, used for initial data exchange, are derived using parameters that change only once per time period, QUIC does not provide forward secrecy guarantees against attackers that may corrupt the server after, but in the same time period as, the data that was sent. Thus, QUIC's overall forward secrecy guarantees are not as strong as those of TLS-DHE, a TLS mode that has recently gained popularity. However, in practice, TLS SessionTickets [14] are often used to minimize round trips. Their use in some sense cancels the forward secrecy guarantees provided by TLS because the SessionTicket key, which must be retained for sufficiently long periods of time for resumption to be effective, can be used to decrypt previous communication. In addition to the formal model, we provide extensive explanations and discussions to help practitioners understand the security level we target.

We then analyze the security of the cryptographic core of QUIC, which we extracted from [2], [6], [15]. We prove that QUIC satisfies our security model assuming strong unforgeability of the underlying signature scheme, security of the underlying symmetric authenticated encryption scheme with associated data, and the strong Diffie-Hellman assumption, in the random oracle model. Due to lack of space we provide the proof in the full version of this paper [16].

We also analyze QUIC's latency goals in the presence of attackers. We show that the very mechanisms used in QUIC to minimize latency, such as unprotected fields on handshake packets and the use of publicly available information on both client and server sides, can be exploited by an adversary during the handshake to introduce extra latencies and possibly lead to DoS attacks. We implemented five attacks against QUIC. Four of these attacks prevent a client from establishing a connection with a server while the fifth is a DoS attack against QUIC servers. In all cases, we found the attacks easy to implement and completely effective. In many cases, the client is forced to wait for QUIC's ten-second connection establishment timeout before giving up.

Our results suggest that the techniques that QUIC uses to minimize latency may not be useful in the presence of malicious parties. Although these weaknesses are not completely unexpected, they are of significant concern to the QUIC team at Google who have been developing a dedicated monitoring infrastructure to try to address them [17]. However, we have found that there may be fundamental

[1]The security goals were not formally stated in QUIC's documentation.

limitations to effectively mitigating these weaknesses.

We note that similar types of attacks have been used against TLS and TCP (recall that TLS runs on top of TCP). However, TLS and TCP made no general promises about their performance in the presence of adversaries. We find that even if QUIC's performance may not be perfect, it is not worse than that of TLS in the worst case, and is much better in the absence of adversaries.

To summarize, our contributions are:

- A security model for QC protocols that formally captures the different levels of security guaranteed for data encrypted under the initial and final session keys in the presence of a very strong adversary, Section VI;
- A provable-security analysis for QUIC under the considered security model, Section VII;
- A quantitative analysis of the performance properties of QUIC under adversarial settings, Section VIII;
- A practicality demonstration of attacks, Section IX.

Our study has shed some light on QUIC's security guarantees and weaknesses that would be useful for practitioners and protocol developers. On a high level, our provable security analysis study confirms the soundness of QUIC's security protection design. And by doing so, our study details the exact level of security the protocol provides, e.g., for data encrypted under the initial and final session keys; something which the protocol description did not specify in sufficient detail. Our performance analysis results confirm yet again that there is no free lunch: either practitioners have to put up with the extra latencies inherent in setting up TLS connections with TCP, or they have to figure out how to deal with the additional security risks introduced by the very mechanisms used to reduce those latencies. Similar tradeoffs were observed with respect to a performance-driven key exchange protocol proposed in [18]. Although in principle QUIC outperforms TLS in terms of latency when there are no attackers, there seems to be a fundamental tradeoff between minimizing latency and providing 'good' security guarantees that practitioners should keep in mind when considering whether to deploy and/or work to improve QUIC or other performance-driven protocols such as TLS 1.3 and TLS with SessionTicket resumption.

FUTURE DIRECTIONS. It would be interesting to see if analyses permitting machine-checked or even automatically-generated proofs using systems like Coq, CryptoVerif [19], EasyCrypt [20] or a type system by Fournet et al. [21] used in TLS analyses [22], [23] could be applied to performance-oriented protocols such as QUIC or TLS 1.3.

II. CONCURRENT AND INDEPENDENT WORK

In (concurrent) work, Fischlin and Günter [11] analyze the key exchange of QUIC. They show that QUIC's (multi-stage) key exchange satisfies a reasonable notion of security. However, this notion does not "compose" with the notions for data exchange, meaning that even if one uses a secure authenticated encryption scheme for data exchange, the security of the QUIC protocol as a whole is not guaranteed. For such a desirable composition to hold, QUIC has to be slightly modified. Traditionally, it has proved very hard to convince practitioners to change implementations unless a serious attack has been deployed. While we believe Google may be more agreeable to tweak the protocol to make it suitable for modular analysis, until that happens, the security of QUIC as a whole is not known.

Furthermore, the change proposed in [11] will not suffice because the way authenticated encryption is used in QUIC prevents its security from generically composing with secure key exchange. Specifically, part of the nonce IV used for encryption is not picked at random independently from everything else but is derived in the same way as the party's secret keys, fixed, and not given to the adversary. To enable a result about the composed security, Google would need to modify its use of encryption. Although we fully support complex protocol design that allows for modular security treatment, we also realize that it is often done differently in practice. So far, our analysis is the only one suitable for the unmodified QUIC.

Another limitation of the analysis in [11] is that it does not address packet-level attacks and IP spoofing. The security model of [11] also does not consider time periods and refreshments of the server configuration. The latter is treated as the long term secret of the server, and the communication of the public portion of it to the client is not considered; i.e., they do not consider 1-RTT connections. Hence, while [11] gives a good insight in the design of multi-stage protocols in a way supporting modular security analyses, our work captures more accurately the current implementation of QUIC and the corresponding practical threats.

III. PRELIMINARIES

NOTATION AND CONVENTIONS. We denote by $\{0,1\}^*$ the set of all binary strings of finite length. If x, y are strings then (x, y) denotes the concatenation of x and y from which x and y are uniquely decodable. If $\kappa \in \mathbb{N}$ then 1^κ denotes the string consisting of κ consecutive "1" bits. If S is a finite set, then $s \xleftarrow{\$} \mathsf{S}$ denotes that s is selected uniformly at random from S. All algorithms are assumed to be randomized and efficient (i.e. polynomial in the size of the input). For any $n \in \mathbb{N}$, $[n]$ denotes the set of integers $\{1, \ldots, n\}$.

PKI. Whenever we use public keys, we also (implicitly) assume that a *public key infrastructure (PKI)* is supported, i.e. the public keys are valid, bound to users' identities, and publicly known. Thus, we omit certificates and certificate checking of public keys in our analysis.

BASE PRIMITIVES AND ASSUMPTIONS.

Digital Signature Scheme. A digital signature scheme is used in QUIC by servers to authenticate certain data, so we define the primitive and its security here.

A *digital signature scheme* $\mathcal{SS} = (\mathsf{Kg}, \mathsf{Sign}, \mathsf{Ver})$ with associated *message space* MsgSp is defined by three algorithms. The randomized *key generation* algorithm Kg takes the security parameter λ and outputs a public–secret key pair: $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(\lambda)$. The *signing* algorithm Sign takes the secret key and message $m \in \mathsf{MsgSp}$ and outputs a signature: $\sigma \xleftarrow{\$} \mathsf{Sign}(sk, m)$. The *verification* algorithm Ver takes the public key, a message and a signature and outputs a bit $b \in \{0,1\}$ indicating whether the signature is deemed valid or not: $b \leftarrow \mathsf{Ver}(pk, m, \sigma)$.

For correctness, it is required that for every (pk, sk) output by $\mathsf{Kg}(\lambda)$ and every $m \in \mathsf{MsgSp}$, $\mathsf{Ver}(pk, m, \mathsf{Sign}(sk, m)) = 1$.

To define security consider the experiment $\mathbf{Exp}^{\text{suf}}_{\mathcal{SS}}(A)$ associated with an adversary A. First, a pair of keys is generated: $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(\lambda)$. Then A is given pk, the oracle $\mathsf{Sign}(sk, \cdot)$, and it has to output a message and a forgery: $(M, \sigma) \xleftarrow{\$} A^{\mathsf{Sign}(sk,\cdot)}(pk)$. The adversary wins and the experiment returns 1 iff $\mathsf{Ver}(pk, m, \sigma) = 1$, $m \in \mathsf{MsgSp}$ and σ was never output by the $\mathsf{Sign}(sk, \cdot)$ oracle. We say that $\mathcal{SS}$ is *strongly unforgeable against chosen message attack (suf-cma)* if $\mathbf{Adv}^{\text{suf}}_{\mathcal{SS}}(A) = \Pr\left[\, \mathbf{Exp}^{\text{suf}}_{\mathcal{SS}}(A) = 1 \,\right]$ is negligible in λ, for all efficient algorithms A.

Authenticated Encryption with Associated Data. After the parties using QUIC establish the shared key, they should be able to use the secure channel to exchange data in a secure manner. The secure channel is implemented by using an authenticated encryption with associated data scheme, which we now define. We adopt the definition of an authenticated encryption with associated data scheme and its corresponding security definition from [24].

An authenticated-encryption with associated-data scheme AEAD consists of two algorithms $\text{AEAD} = (\mathcal{E}, \mathcal{D})$ and is associated with key space $\{0,1\}^\lambda$, nonce space $\{0,1\}^n$, additional authenticated data space $\{0,1\}^*$ and message space $\{0,1\}^*$. The key is generated via $\kappa \xleftarrow{\$} \{0,1\}^\lambda$. $\mathcal{E}$ is a deterministic encryption algorithm that takes inputs key κ, nonce $\text{IV} \in \{0,1\}^n$, additional authenticated data $H \in \{0,1\}^*$ and plaintext $m \in \{0,1\}^*$, and outputs a ciphertext c. $\mathcal{D}$ is a deterministic decryption algorithm that takes inputs key κ, nonce $\text{IV} \in \{0,1\}^n$, additional authenticated data $H \in \{0,1\}^*$, and ciphertext c, and outputs either the plaintext m or $\perp$.

For correctness, it is required that $\mathcal{D}(\kappa, \text{IV}, H, \mathcal{E}(\kappa, \text{IV}, H, m)) = m$ for all $\kappa \in \{0,1\}^\lambda$, $\text{IV} \in \{0,1\}^n$, $H, m \in \{0,1\}^*$.

MESSAGE PRIVACY. To define message privacy let A be an adversary and consider the experiment $\mathbf{Exp}^{\text{ind-cpa}}_{\text{AEAD}}(A)$. It first generates the key $\kappa \xleftarrow{\$} \{0,1\}^\lambda$ and flips a bit $b \xleftarrow{\$} \{0,1\}$. A has access to the encryption oracle $\mathcal{E}(\kappa, \cdot, \cdot, LR(\cdot, \cdot, b))$, where $LR(\cdot, \cdot, b)$ on inputs $m_0, m_1 \in \{0,1\}^*$ with $|m_0| = |m_1|$ returns m_b. At the end A outputs a bit b', and we define A's advantage to be $\mathbf{Adv}^{\text{ind-cpa}}_{\text{AEAD}}(A) = 2\Pr[\, b' = b \,] - 1$.

AUTHENTICITY. To define message integrity and authenticity let A be an adversary and consider the experiment $\mathbf{Exp}^{\text{auth}}_{\text{AEAD}}(A)$. It first generates the key $\kappa \xleftarrow{\$} \{0,1\}^\lambda$. A has access to oracle $\mathcal{E}(\kappa, \cdot, \cdot, \cdot)$. $\mathbf{Exp}^{\text{auth}}_{\text{AEAD}}(A)$ outputs 1 iff A outputs (IV, H, c) such that $\mathcal{D}(\kappa, \text{IV}, H, c) \neq \perp$ and A did not query $\mathcal{E}(\kappa, \text{IV}, H, m)$ for some m that resulted in a response c. We define $\mathbf{Adv}^{\text{auth}}_{\text{AEAD}}(A) = \Pr\left[\, \mathbf{Exp}^{\text{auth}}_{\text{AEAD}}(A) = 1 \,\right]$.

We say that A is nonce-respecting, if it never repeats IV in its oracle queries. We say that an AEAD scheme is *indistinguishable under chosen plaintext attack* (ind-cpa-secure) if $\mathbf{Adv}^{\text{ind-cpa}}_{\text{AEAD}}(A)$ is negligible in λ for any efficient, nonce-respecting adversary A. We say that an AEAD scheme is auth-secure if $\mathbf{Adv}^{\text{auth}}_{\text{AEAD}}(A)$ is negligible in λ for any efficient, nonce-respecting adversary A. We say that any AEAD is secure if it is ind-cpa- and auth-secure.

Strong Computational Diffie-Hellman (SCDH) Assumption. We define the SCDH assumption [25], on which security of QUIC will rely. This assumption was commonly used for analyses of other protocols, including TLS [9].

Consider the experiment $\mathbf{Exp}_{\texttt{SCDH}}(A)$ associated with an adversary A and security parameter λ. A is given (g, q, g^a, g^b), where q is prime of size λ, g is a generator of a cyclic group of order q, and a, b are picked uniformly at random from Z_q. A is also given access to verification oracle $\mathsf{V}(g, g^a, \cdot, \cdot)$, which returns 1 iff queried on g^x, g^{ax} for some $x \in Z_q$. $\mathbf{Exp}_{\texttt{SCDH}}(A)$ returns 1 iff A outputs g^{ab}. We define $\mathbf{Adv}_{\texttt{SCDH}}(A) = \Pr[\, \mathbf{Exp}_{\texttt{SCDH}}(A) = 1 \,]$. We say that the SCDH problem is hard if $\mathbf{Adv}_{\texttt{SCDH}}(A)$ is negligible in λ, for all efficient adversaries A.

IV. QUICK CONNECTIONS PROTOCOL DEFINITION

In this section we formally define a *Quick Connections (QC)* protocol, which is a communication protocol between a client and a server (the latter holds a public key and the corresponding secret key). The parties first agree on an initial session key, which can be used to exchange data until the final key is set. After the final key is set, it is used for further data exchange. The QC definition fits QUIC and is also applicable to other protocols, such as TLS 1.3. This formal definition is necessary for the provable-security analysis.

The protocol is associated with the security parameter λ, a server key generation protocol Kg that on input λ returns public and secret keys, an authenticated encryption with associated data scheme $\text{AEAD} = (\mathcal{E}, \mathcal{D})$ with key space $\{0,1\}^\lambda$, header space $\{0,1\}^*$, message space $\{0,1\}^*$, an IV-extraction function get_iv that takes a key and a header and outputs an $\text{IV} \in \{0,1\}^n$ for each message to be encrypted or decrypted via the associated AEAD, and a scfg_gen function that the server can use to update part of its global state $\texttt{scfg}$. The server can maintain global state other than its $\texttt{scfg}$. All global state is initially set to ε. We associate

a protocol's execution with the universal notion of time, which is divided into discrete periods $\tau_1, \tau_2, \ldots$. The keys are generated via $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(\lambda)$. The input of each party (representing what parties know at the beginning) consists of the public key of the server pk and the list of messages $M^{send} = M_1, \ldots, M_m$ for some $m \in \mathbb{N}$ and where each $M_i \in \{0,1\}^*$, that a party needs to send securely (M^{send} can also be ε). The server has an additional input: the secret key. All parties can keep global state.

In our model, the client and server are given vectors of messages as input. While in practice the messages that the parties exchange may depend on each other, for simplicity we chose not to complicate the protocol syntax. This decision has no implications on our overall security analysis.

Data is exchanged between the parties via packets that must consist of source and destination IP addresses and port numbers followed by the payload associated with the protocol.[2] Each party gets a 32-bit IP address associated with $2^{16} - 1$ port numbers as part of its input. We say that all received and sent packets by a client party belong to that client party's *connection with* a particular server party if the source IP address and port number (as well as any other protocol-specific source information included in packets) of all packets received by that client party correspond to that server and are the same as the destination IP address and port number (as well as any other protocol-specific destination information included in packets) of all packets sent by that client party. We define a server party's connection with a particular client analogously.

Note that different protocols may establish connections based on parameters other than just IP and port numbers (e.g., `cid` in QUIC as will be described in Section V), which is why our definition allows for other protocol-specific parameters contained in packets to be included. The notion of a connection is relevant to the notion of one party *setting a key with* another party which we will establish below and use in our security analysis.

The first packet of data is sent from the client to the server, and we refer to this packet as the *connection request*.

The interactive protocol consists of four phases. Each message exchanged by the parties must belong to some unique stage, but the second and third stages may overlap:

Stage 1: Initial Key Agreement. At the end of this stage each party sets the initial key variable $ik = (ik_c, ik_s, iaux)$, where $iaux \in \{0,1\}^*$ (initially set to ε) is any additional information used for encryption and decryption.

Stage 2: Initial Data Exchange. In this stage, messages from the input data list can be transmitted using the associated AEAD scheme and the key ik. The server uses ik_c to encrypt and ik_s to decrypt, whereas the client uses ik_s to encrypt and ik_c to decrypt. At the end of this stage, each party outputs the list of messages $M^{iget} = M_1, \ldots, M_{m'}$ for some $m' \in \mathbb{N}$ and where each $M_i \in \{0,1\}^*$, (M^{iget} can also be ε), representing the messages the party received in the initial data exchange phase.

Stage 3: Key Agreement. At the end of this stage, each party sets the session key variable $k = (k_c, k_s, aux)$, where $aux \in \{0,1\}^*$ (initially set to ε) is any additional information used for encryption and decryption.

Stage 4: Data Exchange. In this stage, messages from the input data list can be sent using the associated AEAD scheme and the key k. The server uses k_c to encrypt and k_s to decrypt, whereas the client uses k_s to encrypt and k_c to decrypt. At the end of this stage, each party outputs the list of messages $M^{get} = M_1, \ldots, M_{m''}$ for some $m'' \in \mathbb{N}$ and where each $M_i \in \{0,1\}^*$, (M^{get} can also be ε), representing the messages the party received in the final stage.

We say that a party rejects a packet if it outputs $\perp$, and accepts it otherwise.

When a client (or server) party sets ik in Stage 1 corresponding to a particular QC protocol execution instance, we say that client (or server) party *sets* that ik *with* a particular server (or client) party if every sent and received packet by that client (or server) party in Stage 1 of that QC protocol execution instance belongs to that client (or server) party's connection with that server (or client) party. We can define an analogous notion for setting k with respect to Stage 3. We will refer to parties that set ik's in Stage 1 with each other as each other's *peers*.

The *correctness* of the protocol requires that the input data of one party's M^{send} be equal to outputs of the other party's M^{iget}, M^{get}. In other words, the protocol is correct if it allows the parties to exchange the data that they intended to exchange with their corresponding communication partners in the protocol, while preserving the order of the messages.

V. The QUIC Protocol

In this section we present the QUIC protocol. Our description follows the definition for a QC protocol primitive.

In QUIC, the parties associate a connection ID `cid` with the source and destination IP addresses and port numbers of every packet corresponding to that connection. Every incoming packet is checked to see if the source and destination IPs and port numbers correspond to those previously observed for that connection, and that connection is closed if they do not match. For simplicity of presentation, we omit this check in our description below.

Let $\text{AEAD} = (\mathcal{E}, \mathcal{D})$ be an authenticated encryption with associated data scheme, let $\mathcal{SS} = (\mathsf{Kg}_s, \mathsf{Sign}, \mathsf{Ver})$ be a digital signature scheme, and let λ be a security parameter. The signature algorithms supported by QUIC are ECDSA-SHA256 and RSA-PSS-SHA256. AES Galois-Counter mode (GCM) scheme [26] is used as AEAD. QUIC's key generation protocol runs $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(\lambda)$,

[2]We ignore time to live (TTL), header checksums, and other header information not directly relevant to our analysis.

$k_{\text{stk}} \xleftarrow{\$} \{0,1\}^{128}$, and returns pk as the server's public key and (sk, k_{stk}) as the server's secret key.[3]

We assume that the server's scfg is refreshed every time period using the scfg_gen function described below.[4]

scfg_gen(sk, τ_t, λ):

$q \xleftarrow{\$} \{\text{primes of size } \lambda\}$, $g \xleftarrow{\$} \{\text{generators of } \mathbb{Z}_q\}$
$x_s \xleftarrow{\$} \mathbb{Z}_{q-1}$, $y_s \leftarrow g^{x_s}$, $\text{pub}_s \leftarrow (g, q, y_s)$, $\text{sec}_s \leftarrow x_s$
$\text{expy} \leftarrow \tau_{t+1}$, $\text{scid} \leftarrow \mathcal{H}(\text{pub}_s, \text{expy})$
str $\leftarrow$ "QUIC server config signature"
$\text{prof} \leftarrow \mathsf{Sign}(sk, (\text{str}, 0\text{x}00, \text{scid}, \text{pub}_s, \text{expy}))$
$\text{scfg}^{\text{t}}_{\text{pub}} \leftarrow (\text{scid}, \text{pub}_s, \text{expy}, \text{prof})$
$\text{scfg} \leftarrow (\text{scfg}^{\text{t}}_{\text{pub}}, \text{sec}_s)$

$\mathcal{H}$ is the SHA-256 hash function. Note that the generation of scfg and the signing of its public parameters are done independently of clients' connection requests. Although in QUIC there may be several distinct configuration parameters scfg that are valid at any given time, we omit this detail in our analysis, and we do not consider the problem of them expiring during the initial or session key agreement stages.

QUIC supports two connection establishment schemes: 1-RTT handles the case when the client tries to achieve a connection with a server for the first time in a particular time period. 0-RTT considers the case when the client is trying to connect to a server that it has already established at least one connection with in that time period.

A. 1-RTT Connection Establishment

We first describe the case when a client C is trying to achieve a connection with a server S for the very first time at the beginning of time period τ_t. The protocol follows the four stages of the QC model and is presented in Figure 1.

Both C and S know that the current time period is τ_t. C's input message is $M_c = (M_c^1, M_c^2, \cdots, M_c^u)$, while S's input message is $M_s = (M_s^1, M_s^2, \cdots, M_s^w)$. S generates keys $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(\lambda)$ and $k_{\text{stk}} \xleftarrow{\$} \{0,1\}^{128}$.

Initial Key Agreement consists of three packets m_1, m_2, m_3. C initiates a connection by sending the initial connection-request packet m_1 which contains a randomly generated connection id cid, used later by both parties to identify this session. Specifically, C runs c_i_hello(pk) which outputs a packet with sequence number 1.

c_i_hello(pk):

$\text{cid} \xleftarrow{\$} \{0,1\}^{64}$
return $(\text{IP}_c, \text{IP}_s, \text{port}_c, \text{port}_s, \text{cid}, 1)$

S responds with m_2 by running s_reject(m_1). m_2 contains a source-address token stk that will be used later by C to identify itself to S for this and any other additional sessions in 0-RTT connection requests (which we discuss below). An stk is similar to a TLS SessionTicket [14]. It consists of an authenticated-encryption block of the client's IP address and a timestamp. To generate an stk, the server uses the same $\mathcal{E}$ algorithm associated with the AEAD with k_{stk}. The initialization vector iv_{stk} is selected randomly. stk can be used by the client in later connection requests as long as it does not expire and the client does not change its IP address. For simplicity, we take the range of validity of stk to be bound by the time period during which it was generated or set up. m_2 also contains S's current state $\text{scfg}^{\text{t}}_{\text{pub}}$, which contains S's Diffie-Hellman (DH) public values with an expiration date and a signature prof over all the public values under the server's secret key sk.

s_reject(m):

$\text{iv}_{\text{stk}} \xleftarrow{\$} \{0,1\}^{96}$
$\text{stk} \leftarrow (\text{iv}_{\text{stk}}, \mathcal{E}(k_{\text{stk}}, \text{iv}_{\text{stk}}, \varepsilon, (\text{IP}_c, \text{current\_time}_s)))$
return $(\text{IP}_s, \text{IP}_c, \text{port}_s, \text{port}_c, \text{cid}, 1, \text{scfg}^{\text{t}}_{\text{pub}}, \text{stk})$

After receiving m_2, C checks that $\text{scfg}^{\text{t}}_{\text{pub}}$ is authentic and not expired. Note that we assume here that a proper PKI

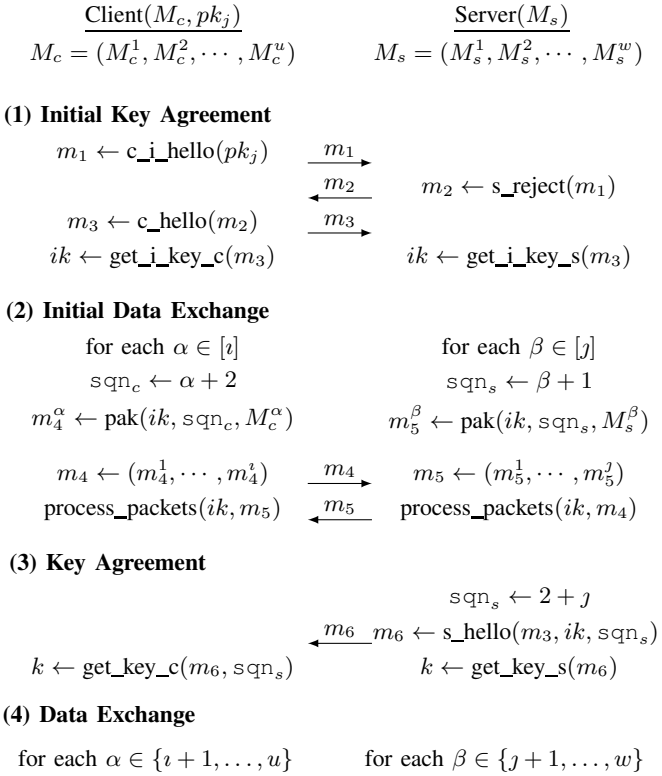

Figure 1. Summary of QUIC's 1-RTT Connection Establishment

[3] In QUIC, k_{stk} is derived using similar methods as the initial and session keys and may depend on user-supplied inputs. Poorly chosen user inputs could lead to IP-spoofing opportunities, but we do not address this weakness because quantifying the predictability of user inputs is out of scope. For simplicity, we assume that users setting up QUIC servers provide unpredictable inputs, and treat k_{stk} as a random string in our analysis.

[4] We ignore the optional server nonce used in the case of persistent time synchronization problems, and such parameters as the server's supported algorithms for key generation, authenticated encryption with associated data and congestion control as they are not pertinent to our security analysis.

is in place, so C possesses the public key of S and is able to perform this check. C then generates a nonce and its own DH values by running c_hello(m_2). C then sends its nonce and public DH values to the server in m_3.

c_hello(m):
abort if $\texttt{expy} \leq \tau_t$ or
$\mathsf{Ver}(pk, (\text{str}, 0\text{x}00, \texttt{scid}, \texttt{pub}_\text{s}, \texttt{expy}), \texttt{prof}) \neq 1$,
where str $\leftarrow$ "QUIC server config signature"
$r \xleftarrow{\$} \{0,1\}^{160}$, $\texttt{nonc} \leftarrow (\texttt{current\_time}_\text{c}, r)$
$x_c \xleftarrow{\$} \mathbb{Z}_{q-1}$, $y_c \leftarrow g^{x_c}$, $\texttt{pub}_\text{c} \leftarrow (g, q, y_c)$
pkt_info $\leftarrow (\texttt{IP}_\text{c}, \texttt{IP}_\text{s}, \texttt{port}_\text{c}, \texttt{port}_\text{s})$
return $(\text{pkt\_info}, \texttt{cid}, 2, \texttt{stk}, \texttt{scid}, \texttt{nonc}, \texttt{pub}_\text{c})$

After this point, C and S derive the initial key material ik by running get_i_key_c(m_3) and get_i_key_s(m_3) respectively. The server has to make sure that it does not process the same connection twice, so it keeps track of used nonces with a mechanism called the strike-register or strike. The client includes a timestamp in its nonc, such that servers only need to maintain state for a limited amount of time, this requires a clock sync between client and server. A server rejects a connection request from a client if its nonc is already included in its strike or contains a timestamp that is outside the allowed time range called $\texttt{strike}_\texttt{rng}$. We consider $\texttt{strike}_\texttt{rng}$ to be bound by the time period during which it was generated or set up.

$ik = (ik_c, ik_s, \texttt{iv})$ consists of two parts: the two 128-bit application keys (ik_c, ik_s) and the two 4-byte initialization vector prefixes $\texttt{iv} = (\texttt{iv}_\text{c}, \texttt{iv}_\text{s})$. C uses ik_s and $\texttt{iv}_\text{s}$ to encrypt data that it sends to S, while using ik_c and $\texttt{iv}_\text{c}$ to decrypt data it receives from S, and vice versa. This stage needs to take place only once per each time period τ_t for which $\texttt{scfg}^\text{t}_\text{pub}$ and stk are not expired. We model the HMAC with a random oracle in our analysis.

get_i_key_c(m):
$\texttt{ipms} \leftarrow y_s^{x_c}$
return xtrct_xpnd($\texttt{ipms}, \texttt{nonc}, \texttt{cid}, m, 40, 1$)

get_i_key_s(m):
$(\texttt{iv}_\text{stk}, \text{tk}) \leftarrow \texttt{stk}$
$\text{dec} \leftarrow \mathcal{D}(k_\text{stk}, \texttt{iv}_\text{stk}, \varepsilon, \text{tk})$
abort if either $\text{dec} = \perp$, or first 4 bytes of dec $\neq \texttt{IP}_\text{c}$, or
last 4 bytes correspond to a timestamp outside allowed_time,
or $r \in \texttt{strike}$, or $\tau_t \notin \texttt{strike}_\texttt{rng}$, or
scid is unknown or corresponds to an expired $\texttt{scfg}^{\text{t}'<\text{t}}_\text{pub}$
or g, q of $\texttt{pub}_\text{c}$ are not the same as g, q of $\texttt{pub}_\text{s}$
$\texttt{ipms} \leftarrow y_c^{x_s}$
return xtrct_xpnd($\texttt{ipms}, \texttt{nonc}, \texttt{cid}, m, 40, 1$)

xtrct_xpnd($\texttt{pms}, \texttt{nonc}, \texttt{cid}, m, \ell, \texttt{init}$):
$\texttt{ms} \leftarrow \text{HMAC}(\texttt{nonc}, \texttt{pms})$
if $\texttt{init} = 1$, then str $\leftarrow$ "QUIC key expansion"
else, str $\leftarrow$ "QUIC forward secure key expansion"
$\texttt{info} \leftarrow (\text{str}, 0\text{x}00, \texttt{cid}, m, \texttt{scfg}^\text{t}_\text{pub})$
return the first ℓ octets (i.e. bytes) of
$\texttt{T} = (\texttt{T}(1), \texttt{T}(2), \cdots)$, where for all $i \in \mathbb{N}$,
$\texttt{T}(i) = \text{HMAC}(\texttt{ms}, (\texttt{T}(i-1), \texttt{info}, 0\text{x}0i))$ and $\texttt{T}(0) = \varepsilon$

Initial Data Exchange consists of two packet sequences m_4 and m_5. C and S exchange their initial data $M_c^1, \ldots, M_c^\imath$ and $M_s^1, \ldots, M_s^\jmath$ encrypted and authenticated using AEAD with ik by running $\text{pak}(ik, \texttt{sqn}_c, M_\alpha^i)$ for each α in $[\imath]$ or $\text{pak}(ik, \texttt{sqn}_s, M_\beta^i)$ for each β in $[\jmath]$ respectively. $\texttt{sqn}_c$ and $\texttt{sqn}_s$ correspond to the sequence numbers of packets sent by C and S respectively.

get_iv in QUIC outputs the iv which is the concatenation of either $\texttt{iv}_\text{c}$ and $\texttt{sqn}_s$ when S sends a packet or $\texttt{iv}_\text{s}$ and $\texttt{sqn}_c$ when C sends a packet. $\texttt{iv}_\text{c}$ and $\texttt{iv}_\text{s}$ are each 4 bytes in length, while $\texttt{sqn}_c$ and $\texttt{sqn}_s$ are each 8 bytes in length. Thus, each iv is 12 bytes in length.

Note that the sequence numbers in QUIC are generated per packet, always start with 1, and are independent of what that packet is carrying. $\imath$ and $\jmath$ correspond to the maximal number of message blocks that C and S can send prior to the *Key agreement* stage. Upon receipt of packets from S, C decrypts them and outputs their respective payloads concatenated together in the order of their sequence numbers with function process_packets. S does the same with packets it receives from C.

get_iv(H, κ):
$(k_c, k_s, \texttt{iv}_\text{c}, \texttt{iv}_\text{s}) \leftarrow \kappa$
if this is a client, then $src \leftarrow c$ and $dst \leftarrow s$
else $src \leftarrow s$ and $dst \leftarrow c$
$(\texttt{cid}, \texttt{sqn}) \leftarrow H$
return $(\texttt{iv}_{dst}, \texttt{sqn})$

pak($\kappa, \texttt{sqn}, m$):
$(k_c, k_s, \texttt{iv}_\text{c}, \texttt{iv}_\text{s}) \leftarrow \kappa$
if this is a client, then $src \leftarrow c$ and $dst \leftarrow s$
else $src \leftarrow s$ and $dst \leftarrow c$
pkt_info $\leftarrow (\texttt{IP}_{src}, \texttt{IP}_{dst}, \texttt{port}_{src}, \texttt{port}_{dst})$
$H \leftarrow (\texttt{cid}, \texttt{sqn})$
$\texttt{iv} \leftarrow \text{get\_iv}(H, \kappa)$
return $(\text{pkt\_info}, \mathcal{E}(k_{dst}, \texttt{iv}, H, m))$

process_packets($\kappa, p_1, ..., p_v$):
$(k_c, k_s, \texttt{iv}_\text{c}, \texttt{iv}_\text{s}) \leftarrow \kappa$
if this is a client , then $src \leftarrow c$ and $dst \leftarrow s$
else $src \leftarrow s$ and $dst \leftarrow c$
for each $\gamma \in [v]$
$(H_\gamma, c_\gamma) \leftarrow p_\gamma$
$\texttt{iv}_\gamma \leftarrow \text{get\_iv}(H_\gamma, \kappa)$
$m_\gamma \leftarrow \mathcal{D}(k_{src}, \texttt{iv}_\gamma, H_\gamma, c_\gamma)$
return $(m_1, \ldots, m_v)$

Key Agreement consists of one message m_6. Specifically, S generates new DH values and sends its new public DH values to the client by running s_hello($m_3, ik, \texttt{sqn}$), encrypted and authenticated using AEAD with ik.

s_hello($m_3, ik, \texttt{sqn}$):
$(ik_c, ik_s, \texttt{iv}_\text{c}, \texttt{iv}_\text{s}) \leftarrow ik$
$\tilde{x}_s \xleftarrow{\$} \mathbb{Z}_{q-1}$, $\tilde{y}_s \leftarrow g^{\tilde{x}_s}$, $\tilde{\texttt{pub}}_\text{s} \leftarrow (g, q, \tilde{y}_s)$
$H \leftarrow (\texttt{cid}, \texttt{sqn})$
$e \leftarrow \mathcal{E}(ik_c, (\texttt{iv}_\text{c}, \texttt{sqn}), H, (\texttt{scfg}^\text{t}_\text{pub}, \tilde{\texttt{pub}}_\text{s}, \texttt{stk}))$
return $(\texttt{IP}_\text{s}, \texttt{IP}_\text{c}, \texttt{port}_\text{s}, \texttt{port}_\text{c}, H, e)$

The client verifies the authenticity of the server's new DH public values upon receipt of this packet using ik and both parties at this point can derive the session key material k by running get_key_s(m_6) and get_key_c(m_6), which both use the xtrct_xpnd function defined earlier.

get_key_s(m):
$\texttt{pms} \leftarrow y_c^{\tilde{x}_s}$
return xtrct_xpnd($\texttt{pms}, \texttt{nonc}, \texttt{cid}, m, 40, 0$)

get_key_c(m):
  $(\mathtt{IP_s}, \mathtt{IP_c}, \mathtt{port_s}, \mathtt{port_c}, \mathtt{cid}, \mathtt{sqn}, e) \leftarrow m$
  abort if $\mathcal{D}(ik_c, (\mathtt{iv_c}, \mathtt{sqn}), (\mathtt{cid}, \mathtt{sqn}), e) = \perp$
  $\mathtt{pms} \leftarrow \tilde{y}_s^{x_c}$
  return xtrct_xpnd$(\mathtt{pms}, \mathtt{nonc}, \mathtt{cid}, m, 40, 0)$

Data Exchange consists of two packet sequences m_7 and m_8. C and S will use k to encrypt and authenticate their remaining data $M_c^{\iota+1}, \ldots, M_c^u$ and $M_s^{\jmath+1}, \ldots, M_s^w$, respectively, instead of ik for the rest of this session.

Similar to ik, $k = (k_c, k_s, \mathtt{iv})$ consists of two parts: the two 128-bit application keys (k_c, k_s) and the two 4-byte initialization vector prefixes $\mathtt{iv} = (\mathtt{iv_c}, \mathtt{iv_s})$. C uses k_s and $\mathtt{iv_s}$ to encrypt data that it sends to S, while using k_c and $\mathtt{iv_c}$ to decrypt data received from S, and vice versa.

B. 0-RTT Connection Establishment

If the client C has already had a connection with a server S in the time period τ_t, then C does not need to send the c_i_hello, but can instead initiate another connection request with the server via a c_hello packet containing the previously obtained stk and scid, as well as new cid, nonc, and $\mathtt{pub_c}$ (which should contain its new DH ephemeral public value). In this case, the c_hello function will be:

c_hello$(\mathtt{stk}, \mathtt{scfg}^t_{\mathtt{pub}})$:
  $\mathtt{cid} \xleftarrow{\$} \{0,1\}^{64}$
  $r \xleftarrow{\$} \{0,1\}^{160}$, $\mathtt{nonc} \leftarrow (\mathtt{current\_time_c}, r)$
  $x_c \xleftarrow{\$} \mathbb{Z}_{q-1}$, $y_c \leftarrow g^{x_c}$, $\mathtt{pub_c} \leftarrow (g, q, y_c)$
  pkt_info $\leftarrow (\mathtt{IP_c}, \mathtt{IP_s}, \mathtt{port_c}, \mathtt{port_s})$
  return (pkt_info, cid, 1, stk, scid, nonc, $\mathtt{pub_c}$)

Upon receipt of c_hello, S verifies that the nonc is fresh by checking it against its strike-register, that the stk is valid, and that scid is not unknown or expired. If the verification steps fail, S goes back to the 1-RTT case by generating and sending the s_reject as described in Section V-A, and then the rest of the protocol is exactly the same as described in Section V-A. If, however, these verification steps succeed, the rest of the protocol is exactly the same as in Section V-A, except that the packet sequence numbers account for the fact that there are two fewer packets.

VI. Security Model

We formally define the security model for QC protocols, which is one of our main technical contributions. Our model is an extension of the Authenticated and Confidential Channel Establishment (ACCE) security model for TLS to accommodate performance-driven protocols that do not run on top of TCP and have two stages for key agreement and data exchange. We call our model QACCE for Quick ACCE.

We consider a very strong attacker who can initiate possibly concurrent rounds of a protocol between various clients and servers and see the exchanged communication. Moreover, an attacker can corrupt servers, control clients, and drop or modify the packets exchanged by the honest parties. Our definition targets the major security goal of a communication protocol: secure channel, which means that data is exchanged in a private and authentic manner and cannot be re-ordered. The necessary goal of key security and (unilateral) authentication is also captured by the definition. Furthermore, the model addresses particular attacks such as IP spoofing attacks.

After presenting the formal definition (with informal explanations) we discuss the differences from the existing security models and the reasons for them.

In Section VI-B we explain how our formal model captures server impersonation attacks, attacks on secure channel, and such malicious behaviors as eavesdropping, man-in-the-middle attacks, forgeries, and DDoS attacks (e.g. due to IP spoofing). We also explain the levels of forward secrecy a protocol can provide. We hope our informal discussions help make our analyses useful for practitioners.

A. Security Definition

Security Experiment. Fix the security parameter λ and a QC protocol Π with associated server key generation protocol Kg, scfg_gen, an authenticated encryption with associated data scheme AEAD $= (\mathcal{E}, \mathcal{D})$ with key space $\{0,1\}^\lambda$ and additional authenticated data (which we will denote by H) space $\{0,1\}^*$.

We define the experiment $\mathbf{Exp}^{\mathrm{QACCE}}_{\Pi}(A)$ associated with the adversary A. We consider two sets of parties, clients and servers, $C = \{C_1, \ldots, C_\ell\}$ and $S = \{S_1, \ldots, S_\ell\}$, for parameter $\ell \in \mathbb{N}$ denoting the maximum possible number of servers or clients. The experiment first generates server key pairs $(pk_i, sk_i) \xleftarrow{\$} \mathsf{Kg}(\lambda)$, $k_{\mathsf{stk}} \xleftarrow{\$} \{0,1\}^{128}$, and $\mathtt{scfg}_i^t \xleftarrow{\$}$ scfg_gen(sk_i, τ_t, λ), for all time periods, for all $i \in [\ell]$.

To capture several sequential and parallel executions of the protocol we follow the standard approach and associate each party $P_i \in \{C \cup S\}$ with a set of stateful oracles $\pi^1_{p,i}, \ldots, \pi^d_{p,i}$, for parameter $d \in \mathbb{N}$ and $p \in \{c, s\}$, where each oracle $\pi^{r \in [d]}_{p,i}$ represents a process that executes one single instance of the protocol at party P_i and p indicates whether the party in question is a client or server. Intuitively, each oracle π^s_i of some party $P_i \in \{C \cup S\}$ models that party's IP address and a unique port number. We discuss the importance of this part more in Section VI-B. The experiment flips a bit $b^q_{p,i} \xleftarrow{\$} \{0,1\}$ for each oracle $\pi^q_{p,i}$.

Each server oracle gets the corresponding $\mathtt{scfg}_i^t$ at the beginning of each time period. We assume that at each point of the protocol's execution each party (adversary included) can tell what time period it is. We also assume that every server oracle is aware what protocol stage it is in for every client oracle that it is and/or has been exchanging messages with. With this assumption we are not required to keep track of the stages in the simulations in our proofs detailed in the full version [16]. Even though the server keeps local state and knows which stage it is in, it may have inaccurate view of the stage of the protocol because it is not guaranteed to

know the correct identity of the party it is talking with. We refer to oracles that set ik with each other as *peers*.

The adversary A is given the public keys of all servers $pk_1, \ldots, pk_\ell$ and can interact with all oracles of all parties by issuing queries. The values in parentheses are supplied by A, except when they are bold face. If the parameter in parentheses is an oracle, e.g. $\pi^q_{p,i}$, this means that A needs to supply the indices p, i, q specifying the oracle.

- $\texttt{connect}(\pi^q_{\mathbf{c},i}, \pi^r_{\mathbf{s},j})$, for $i, j \in [\ell], q, r \in [d]$.

As a result, $\pi^q_{c,i}$ outputs the initial connection request packet (first connection for that client party for that particular time period) that it would send specifically to oracle $\pi^r_{s,j}$ according to the protocol. The output of this query is not delivered to the recipient oracle, but is just given to A.

This query allows the adversary to ask a client oracle to start communicating with a particular server party for the first time between those parties for a particular time period.

- $\texttt{resume}(\pi^q_{\mathbf{c},i}, \pi^r_{\mathbf{s},j})$, for $i, j \in [\ell], q, r \in [d]$.

This query returns $\perp$ if ik corresponding to oracle $\pi^q_{c,i}$ is not set. Otherwise, $\pi^q_{c,i}$ outputs the 0-RTT connection request packet that it would send to an oracle $\pi^r_{s,j}$ according to the protocol. The output is given to A, who can deliver it to the destination oracle, modify it, or drop it.

This query allows the adversary to ask a particular client oracle to request a 0-RTT connection with a particular server party, if the client party corresponding to that oracle has communicated before with that server in a particular time period. Recall that every server party is aware of its communication status with respect to every client oracle that may contact it.

- $\texttt{send}(\pi^r_{p,j}, m)$, for $p \in \{c, s\}$, $j \in [\ell], r \in [d]$ and $m \in \{0,1\}^*$.

As a result, m is sent to $\pi^r_{p,j}$, which will respond with $\perp$ if the oracle is in data exchange phase. Otherwise, A gets the response, which is defined according to the protocol.

This query allows the adversary to send a specified packet m to a specified destination oracle. Note that the attacker must provide a header for the packet that specifies the source and destination IP addresses and port numbers as well as packet sequence numbers of its choice. The destination oracle can check this information. The adversary gets control of the resulting packet and can choose to modify, drop, or deliver it to an oracle.

- $\texttt{revealik}(\pi^q_{p,i})$, for $p \in \{c, s\}$, $i \in [\ell], q \in [d]$.

As a result, A receives the contents of variable ik for oracle $\pi^q_{p,i}$.

This query allows the adversary to learn the initial key set by the oracle of its choice.

- $\texttt{revealk}(\pi^q_{p,i})$, for $p \in \{c, s\}$, $i \in [\ell], q \in [d]$.

As a result, A receives the contents of variable k for oracle $\pi^q_{p,i}$.

This query allows the adversary to learn the final key set by the oracle of its choice.

- $\texttt{corrupt}(S_i)$, for $i \in [\ell]$.

A gets back sk_i and the current $\texttt{scfg}^t_i$ and any other state of S_i.

This query allows the adversary to corrupt the server of its choice and learn its long-term secrets including $\texttt{scfg}^t_i$ for the current time period.

- $\texttt{encrypt}(\pi^r_{p,j}, m_0, m_1, H, \texttt{init})$, for $p \in \{c, s\}$, $j \in [\ell], r \in [d]$, $m_0, m_1, H \in \{0,1\}^*$, and $\texttt{init} \in \{0,1\}$:

  return $\perp$ if $|m_0| \neq |m_1|$ or $\texttt{init} = 1$ and $\pi^r_{p,j}$ is not in the initial data exchange stage or if $\texttt{init} = 0$ and $\pi^r_{p,j}$ is not in the data exchange stage
  $p' \leftarrow \{c, s\} \setminus \{p\}$
  if $\texttt{init} = 1$
    IV $\leftarrow$ get_iv(ik, H), return $\perp$ if IV was used
    return $(H, \mathcal{E}(ik_{p'}, \text{IV}, H, m_{b^q_{p,j}}))$
  if $\texttt{init} = 0$
    IV $\leftarrow$ get_iv(k, H), return $\perp$ if IV was used
    return $(H, \mathcal{E}(k_{p'}, \text{IV}, H, m_{b^q_{p,j}}))$

  Above, $ik, k, ik_{p'}, p'$ belong to $\pi^r_{p,j}$.

This query, unlike the previous ones, deals with the initial and final data exchange phases (flag $\texttt{init}$ specifies which), while the previous ones concerned the initial and final key exchange phases. It is designed to follow the standard approach of capturing message privacy under chosen-message attack. It allows the adversary to obtain a randomly chosen ciphertext out of the two messages provided by the adversary. Just like in the security definition for AEAD, the attacker can select the header H. For QUIC it means that the adversary can specify the source and destination IP addresses and port numbers as well as packet sequence numbers of its choice. Unlike the AEAD security model, however, we do not let the adversary select the IV because in QUIC the IV depends on the secrets of a party and is not under the attacker's control. get_iv is the function that we require to produce initialization vectors used for encryption and appropriate headers. The initialization vector is not given to the adversary. The adversary is restricted to providing H whose destination IP address and port number correspond to $\pi^r_{p,j}$ and whose source IP address and port number correspond to an oracle $\pi^q_{p',i}$ in the experiment, for $p' \in \{c, s\} \setminus \{p\}$.

- $\texttt{decrypt}(\pi^r_{p,j}, \mathcal{C}, H, \texttt{init})$, for $p \in \{c, s\}$, $j \in [\ell], r \in [d]$, $\mathcal{C}, H \in \{0,1\}^*$, and $\texttt{init} \in \{0,1\}$:

  return $\perp$ if $\texttt{init} = 1$ and $\pi^r_{p,i}$ is not in the initial data exchange phase, or $\texttt{init} = 0$ and $\pi^r_{p,j}$ is not in the data exchange phase, or $(H, \mathcal{C})$ was output before by $\texttt{encrypt}(\pi^r_{p,j}, *, *, *, \texttt{init})$
  if $\texttt{init} = 1$
    IV $\leftarrow$ get_iv(ik, H),

if $\mathcal{D}(ik_p, \text{IV}, H, \mathcal{C}) \neq \perp$, return $b^r_{p,j}$ else return $\perp$
if $\texttt{init} = 0$
$\text{IV} \leftarrow \text{get\_iv}(k, H)$,
if $\mathcal{D}(k_p, \text{IV}, H, \mathcal{C}) \neq \perp$, return $b^r_{p,j}$ else return $\perp$

Above, $ik, k, ik_{p'}, p'$ belong to $\pi^r_{p,j}$.

This query also concerns the initial and final data exchange phases. It follows the standard approach to capture authenticity for AEAD schemes. The adversary's goal is to create a "new" valid ciphertext. If it succeeds, it is given the challenge bit and thus can win.

- $\texttt{connprivate}(\pi^q_{c,i}, \pi^r_{s,j})$, for $i, j \in [\ell], q, r \in [d]$.

As a result, the initial connection request is sent to $\pi^r_{s,j}$. The response, which is defined according to the protocol, is sent to $\pi^q_{c,i}$ and not shown to A. Any following response of $\pi^q_{c,i}$ is not shown to A.

This query is not part of the existing definitions. It models IP spoofing attacks, which the previous models did not consider. We explain its importance below when we discuss A's advantage.

After the adversary is done with queries it may output a tuple (p, i, q, b), for $p \in \{c, s\}$.

Before we proceed with the security definition we define the notion of a matching conversation [27] taking place between a client and a server. The scope of this concept is the initial and final key exchange phases only.

MATCHING CONVERSATIONS. For $p \in \{c, s\}$, $p' \in \{c, s\} \setminus \{p\}$, $i, j \in [\ell], q, r \in [d]$, we denote with $R^q_{p,i}$ the sequence of all messages used for establishing keys (during stages 1 and 3) sent and received by $\pi^q_{p,i}$ in chronological order, and we call $R^q_{p,i}$ the message record at $\pi^q_{p,i}$. With respect to two message records $R^q_{p,i}$ and $R^r_{p',j}$, we say that $R^q_{p,i}$ is a prefix of $R^r_{p',j}$, if $R^q_{p,i}$ contains at least one message, and the messages in $R^q_{p,i}$ are identical to and in the same order as the first $|R^q_{p,i}|$ messages of $R^r_{p',j}$. We say that $\pi^q_{p,i}$ has a *matching conversation* with $\pi^r_{p',j}$, if the following two conditions are both true:

- either $p = c$ and $p' = s$, or $p' = c$ and $p = s$;
- either $R^r_{p',j}$ is a prefix of $R^q_{p,i}$ and $\pi^q_{p,i}$ has sent the last message(s), or $R^q_{p,i}$ is a prefix of $R^r_{p',j}$ and $\pi^r_{p',j}$ has sent the last message(s).

Note that the notion of a matching conversation is not sufficient to define peers because, unlike in TLS, communicating parties in QUIC may set initial keys without having a matching conversation. This is why throughout our analysis the notion of peers is instead equivalent to the notion of one party *setting a key with* another party.

MEASURES OF A'S ATTACK SUCCESS.

- The *server impersonation advantage* of A $\mathbf{Adv}^{\texttt{s-imp}}_{\Pi}(A)$ is the probability that there exists an oracle $\pi^q_{c,i}$ such that k of this oracle is set and there is no oracle $\pi^r_{s,j}$ corresponding to a server party S_j such that $\pi^q_{c,i}$ has a matching conversation to $\pi^r_{s,j}$, no $\texttt{revealik}$ contained ik possibly set in the optional initial key agreement stage between $\pi^q_{c,i}$ and $\pi^r_{s,j}$, and S_j was not corrupted.

The above captures the attack when the adversary impersonates an honest server and makes a client think it sets a key shared with the server, but the adversary may have the shared key instead.

- The *channel-corruption advantage* of A $\mathbf{Adv}^{\texttt{ch-corr}}_{\Pi}(A)$ is $2\Pr\left[\, b = b^q_{p,i} \,\right] - 1$,

where if $p = s$, then it must be the case that $\pi^q_{s,i}$ has a matching conversation with some client oracle $\pi^r_{c,j}$, such that the following conditions hold

1) if S_i was corrupted, then no $\texttt{encrypt}(\pi^q_{s,i}, *, *, *, 1)$ and $\texttt{encrypt}(\pi^r_{c,j}, *, *, *, 1)$ queries were made for any $*$ after or during the same time period τ_t that S_i was corrupted,
2) if S_i was corrupted, then no $\texttt{encrypt}(\pi^q_{s,i}, *, *, *, *)$ and $\texttt{encrypt}(\pi^r_{c,j}, *, *, *, *)$ queries were made for any $*$ after S_i was corrupted, and
3) no $\texttt{revealik}(\pi^q_{s,i})$ and $\texttt{revealik}(\pi^r_{c,j})$ or $\texttt{revealk}(\pi^q_{s,i})$ and $\texttt{revealk}(\pi^r_{c,j})$ queries returned the key used to answer any $\texttt{encrypt}(\pi^q_{s,i}, *, *, *, *)$ and $\texttt{encrypt}(\pi^r_{c,j}, *, *, *, *)$ queries for any $*$ respectively;

and if $p = c$, then let $\pi^r_{s,j}$ be peer of $\pi^q_{c,i}$. The following conditions must be satisfied.

1) if S_j was corrupted, then no $\texttt{encrypt}(\pi^q_{c,i}, *, *, *, 1)$ and $\texttt{encrypt}(\pi^r_{s,j}, *, *, *, 1)$ queries were made for any $*$ after or during the same time period τ_t that S_j was corrupted,
2) if S_j was corrupted, then no $\texttt{encrypt}(\pi^q_{c,i}, *, *, *, *)$ and $\texttt{encrypt}(\pi^r_{s,j}, *, *, *, *)$ queries were made for any $*$ after S_j was corrupted, and
3) no $\texttt{revealik}(\pi^q_{c,i})$ and $\texttt{revealik}(\pi^r_{s,j})$ or $\texttt{revealk}(\pi^q_{c,i})$ and $\texttt{revealk}(\pi^r_{s,j})$ queries returned the key used to answer any $\texttt{encrypt}(\pi^q_{c,i}, *, *, *, *)$ and $\texttt{encrypt}(\pi^r_{s,j}, *, *, *, *)$ queries for any $*$ respectively.

The above captures the attacks in which information about groups of messages exchanged between the client and the server is leaked without the adversary corrupting the server party (1) before or (2) during the same time period as attempting the breach as well as without (3) revealing the initial and session keys ik and k. Thus, we capture a slightly weaker notion of forward secrecy by restricting the adversary to corrupt the appropriate server only after the time period when the adversary attempts the breach. We explain this subtlety further in Section VI-B.

- The *IP spoofing* of A $\mathbf{Adv}^{\texttt{ips}}_{\Pi}(A)$ is the probability that there exist oracles $\pi^q_{c,i}$ and $\pi^r_{s,j}$ such that at some time period τ_t A makes a $\texttt{send}(\pi^r_{s,j}, m')$ query, $\pi^r_{s,j}$ does not reject this query, S_j was not corrupted, m' is not an output resulting from any previous connection request query (done via $\texttt{connect}$ or $\texttt{resume}$ queries), and the

only other query A is allowed to make concerning $\pi^q_{c,i}$ during τ_t is the $\texttt{connprivate}(\pi^q_{c,i}, \pi^r_{s,j})$ query.

This goal captures attacks in which the adversary wins if it succeeds in having the server accept a connection request on behalf of a client who either did not request connection to that server or previously requested only an initial connection but did not request any further connections in the same time period. The adversary issues a connection query hoping it gets accepted by the server, possibly preceded by the only other allowed query in that time period: connection request (`connprivate`) whose output it cannot see.

SECURITY DEFINITION. We say that a QC protocol Π is QACCE-secure if its advantage $\mathbf{Adv}^{\texttt{QACCE}}_{\Pi}(A)$, defined as $\mathbf{Adv}^{\texttt{s-imp}}_{\Pi}(A)+\mathbf{Adv}^{\texttt{ips}}_{\Pi}(A)+\mathbf{Adv}^{\texttt{ch-corr}}_{\Pi}(A)$, is negligible (in λ) for any polynomial-time adversary A.

B. Security Model Discussion

COMPARISON TO THE EXISTING MODELS. Existing models do not fit QUIC. Namely, we could not simply compose key exchange [28] and authenticated encryption definitions because QUIC has additional initial key and data exchange stages. The work [11] extended the key exchange definition of [28] to treat multiple stages of key exchange, but QUIC does not achieve their definition. Moreover, even with their fix, the full security of QUIC will not follow from their results because QUIC's secure channel implementation is not independent from the key exchange phases.

Therefore, similarly to recent analyses of protocols such as TLS [8], [9] and EMV [29], [30] we chose to work with a dedicated definition that assess the security of a protocol as a whole. We followed the ACCE model for TLS but had to modify it to accommodate for dealing with setting and using the initial key, which was not present in TLS. Moreover, QUIC handles novel security goals that TLS did not address, such as some cryptographic protection for network packet handling and protection against IP spoofing. We comment on these in more detail below.

ON SECURING PACKETS. Any communication protocol that does not run on top of TCP risks having its packets be misordered and/or not delivered at all. QUIC, unlike TLS, does not run on top of TCP but instead runs on top of UDP, which does not provide any delivery guarantees. Since QUIC adds cryptographic protection to some tasks usually handled by TCP, it makes sense to capture this in our model. Thus, in our security definition we allow the adversary to intercept, delay, misorder, modify, and selectively drop any communication between a client and a server. Our model captures the fact that data in real life is transmitted in packets and that the adversary could in principle modify such packet fields as source and destination IP addresses and port numbers. Specifically, we give the adversary the ability to specify the precise oracles associated with certain parties as subjects of its queries to send and/or receive messages of the adversary's choice. Our security model does not, however, capture adversaries that simply drop (or delay for an unreasonably long time) all possible traffic because mitigating such attacks would require more sophisticated protocols than those captured by our QC protocol model that could detect and avoid failures.

ON SERVER IMPERSONATION. The server impersonation goal in our model captures attacks in which the adversary attempts to convince the client to set a session key that is in any way inconsistent with the key set by the server. That is, when using a secure protocol, a client knows that the final session key is shared only with the server the client talked to and no one else. We do not capture attacks of the same type with respect to initial keys in this goal. This is because it may not be possible in general, since the client may have to derive the initial key from the semi-permanent `scfg` that could be used for many client connection requests while it persists. This would allow, for example, the adversary to replay the values of `scfg` to clients that have not yet contacted the corresponding server, which could lead to some clients establishing an initial key without the server being aware of their connection request. This weakness may also be relevant to TLS variants that allow for stateless connection resumption, and we discuss it in more detail in Section VIII. Although for simplicity we do not to address this directly in our analysis, the requirement of having a matching conversation captures the basic mandate that the communicating parties may need to agree not only on the session key, but also on any other important communication parameters such as congestion control, key generation, encryption algorithms, etc. Thus, in principle, this goal not only captures the traditional man-in-the-middle attacks, but also more subtle attacks where the adversary may be interested in degrading the communication security and performance due to parties having inconsistent views of session parameters. For example, when the two parties disagree on congestion-avoidance parameters, a server may end up sending content at much lower or higher rates than requested by the client.

ON CHANNEL SECURITY. The channel corruption goal in our model captures the expected goals of data authenticity and confidentiality with forward secrecy, in a way that is similar to the models used to analyze TLS but with a few crucial additions that we detail below. The goal of authenticity implicitly captures attacker's misordering, selectively delaying, and dropping certain content as well as positive ACK attacks, all of which involve the adversary sending something on behalf of a participating party. The content of any packet that is dropped or delayed beyond a certain time threshold (possibly dictated by the congestion-avoidance parameters that may be optionally negotiated by the communicating parties that we discuss below) could be retransmitted unless its receipt is positively acknowledged by the receiver. Thus, to prevent content delivery an adversary

could in principle positively acknowledge the receipt of packets on behalf of the receiver, which is captured by the authenticity goal in our model. This security goal also captures positive ACK attacks, which involve the adversary or a rogue receiver sending acknowledgments for content that was not actually received to cause the sender to send too much content and overwhelm the resources of intermediate and/or receiving network(s).

ON FORWARD SECRECY. A QACCE-secure protocol guarantees that the final session keys are forward secure, i.e. obtaining a server's long-term secrets does not leak any information about the data that was previously exchanged and encrypted under these keys. However, the guarantees with respect to the initial keys are weaker because, for them, forward secrecy holds only if the server does not get corrupted during the time period when the `scfg` that was used to derive those keys is valid. This is because, in QUIC, servers use the same `scfg` to derive initial keys with all clients for the duration of that `scfg`'s validity.

Unlike in previous models used to study TLS, we also impose some additional restrictions on the adversary that prevent it from revealing the initial key and corrupting the server during the same time period as its encryption queries. This restriction is imposed on the adversary because initial keys are not forward secure, as they could be derived using semi-permanent values stored by the server in its corresponding `scfg`, which is changed only once per time period, during which it could be used for all client connection requests in that period. Thus, to account for this weakness, it is important that the adversary does not learn of any semi-permanent state captured in the server's `scfg` that could be used for establishing initial keys during its lifetime. This weakness may also be relevant to TLS variants that allow for stateless connection resumption [14].

ON RE-ORDERING ATTACKS. As we mentioned before, strong security for secure channel, in addition to data privacy and authenticity, must guarantee security against re-ordering attacks. In the ACCE model for TLS [8], [9] this is captured by requiring the authenticated encryption scheme satisfy the notion of stateful decryption [31]. That definition requires each ciphertext delivered out of order to be rejected.

This notion is not suitable for QUIC analysis. In TLS, if the adversary tampers with the packet order at the TCP level, all re-ordered packets will be rejected as the receiver will detect re-ordering by comparing the order with the one indicated by the TLS-layer sequence numbers. Hence, the notion of stateful decryption can be met. In QUIC, this is impossible, because it does not run on top of TCP. The receiver gets *all* information about the packet order from the sequence numbers. The receiver in QUIC cannot reject any packet, even if it "looks" out of order, until the end, when the messages could be sorted and the proper order could be determined. Thus, re-ordering the existing packets is prevented in TLS but cannot be prevented in QUIC. Yet, the final order of the messages should still be correctly determined by the receiver in QUIC. This difference calls for different treatment in the security model.

For simplicity, we chose to capture re-ordering attacks somewhat implicitly. Note that for any protocol which authenticates the sequence numbers (in QUIC the sequence number is part of the authenticated header H), re-ordering is enforced by the authentication security which is part of encryption breach security. Namely, changing the legitimate order of the packets will require the adversary to create a valid ciphertext with a new sequence number, and this constitutes a "forgery" of encryption in the current security definition. It is possible to treat re-ordering attacks more explicitly, but this would require making the model less general and more involved as we will have to fit the sequence numbers into the syntax and security definition.

ON IP SPOOFING. Since it may not be possible to authenticate a client, attacks where the adversary initiates multiple connections to a server on behalf of honest clients by spoofing its IP address are possible. Such DoS attacks can lead to exhaustion of a server's resources resulting from prohibitively high rates of superfluous derivations of session keys. Because TCP provides protection against such attacks with its three-way handshake, they are not considered when analyzing protocols that rely on TCP, such as TLS. However, such attacks must be addressed for protocols that do not run on top of TCP, such as QUIC, and the third goal in our security model captures them. In the IP spoofing goal, the adversary wins if it can trick the server into establishing a session key with a client that did not request it.

VII. QUIC SECURITY ANALYSIS

We state our main result about the security of QUIC.

Theorem 7.1: Consider the cryptographic core of $QUIC$, as defined in Section V, associated with the base signature scheme $\mathcal{SS} = (\mathsf{Kg}_s, \mathsf{Sign}, \mathsf{Ver})$, and an authenticated-encryption with associated-data scheme $\text{AEAD} = (\{0,1\}^\lambda, \{0,1\}^n, \mathcal{E}, \mathcal{D})$. Then QUIC is QACCE if $\mathcal{SS}$ is suf-cma and AEAD is ind-cpa- and auth-secure and the SCDH problem is hard, in the random oracle model.

REMARK. We treat HMAC as the random oracle. This is a very common assumption for security analyses. While it may not be appropriate in every case, as cautioned in [32], the standard use of HMAC with fixed keys for the key derivation function here seems fine. There are two uses of HMAC in the key derivation function, and it is important for the analysis that the first occurrence is the random oracle. The second one may satisfy a weaker notion, but we treat it as the random oracle for simplicity. It does not seem possible to get rid of the reliance on the random oracle in the first case though.

The Theorem follows from the following three lemmas.

Let ℓ be the number of servers (and clients), let d be the maximum number of oracles corresponding to any party

(i.e. the maximum number of connection sessions a party can initiate), let T be the number of time periods and let Q be the maximum number of decryption queries the adversary does in $\mathbf{Exp}_{\Pi}^{\texttt{QACCE}}$.

Lemma 7.2: For any efficient adversary A there exist efficient adversaries B, C, D, E such that

$$\begin{aligned}\mathbf{Adv}_{QUIC}^{\texttt{ch-corr}}(A) &\leq \ell\mathbf{Adv}_{\mathcal{SS}}^{\texttt{suf}}(B) + 2d\ell^2T\mathbf{Adv}_{\texttt{SCDH}}(C)\\ &+ 4\ell dTQ\mathbf{Adv}_{\text{AEAD}}^{\text{auth}}(D)\\ &+ 4\ell dT\mathbf{Adv}_{\text{AEAD}}^{\text{ind-cpa}}(E)\ .\end{aligned}$$

Lemma 7.3: For any efficient adversary A there exist efficient adversaries B, C, D such that

$$\begin{aligned}\mathbf{Adv}_{QUIC}^{\texttt{s-imp}}(A) &\leq \ell\mathbf{Adv}_{\mathcal{SS}}^{\texttt{suf}}(B) + 2d\ell^2T\mathbf{Adv}_{\texttt{SCDH}}(C)\\ &+ 7\ell dTQ\mathbf{Adv}_{\text{AEAD}}^{\text{auth}}(D)\ .\end{aligned}$$

Lemma 7.4: For any efficient adversary A there exists an efficient adversary B such that

$$\mathbf{Adv}_{QUIC}^{\texttt{ips}}(A) \leq \ell\mathbf{Adv}_{\text{AEAD}}^{\text{auth}}(B)\ .$$

The detailed proofs of the Lemmas can be found in the full version [16].

VIII. Performance Issues and Malice

In this section we discuss how simple attacks on QUIC packets during the handshake can introduce latencies, essentially countering one of the primary goals of the protocol: 0-RTT connection establishment. Persistent failure to establish a QUIC session could further result in a fall-back to TCP, defeating QUIC's purpose of minimizing latency while securing the transport layer. We discuss two types of attacks: the first exploits public, cachable information from either the server or client side, the second exploits unprotected fields on packets exchanged during the handshake protocol.

A. Replay Attacks

Once at least one client establishes a session with a particular server, an adversary could learn the public values of that server's `scfg` as well as the source-address token value `stk` corresponding to that client during their respective validity periods. The adversary could then replay the server's `scfg` to the client and the source-address token `stk` to the server, misleading in either case the other party. To launch both attacks an adversary would have to have access to the communication channel.

Server Config Replay Attack. An attacker can replay a server's public `scfg` to any other clients sending initial connection requests to that server while keeping the server unaware of such requests from clients. Thus, these clients establish an initial key without the server's knowledge, and when they attempt to communicate with the server, the server would not be able to recognize them and would reject their packets. While data confidentiality is not affected, the clients would experience additional latencies and waste computational resources deriving an initial key.

Source-Address Token Replay Attack. An attacker can replay the source-address token `stk` of a client to the server that issued that token on behalf of the client many times to establish additional connections. This action would cause the server to establish initial keys and even final forward-secure keys for each connection without the client's knowledge. Any further steps in the handshake would fail, but an adversary could create a DoS attack on the server by creating many connections on behalf of a many different clients and possibly exhausting the server's computational and memory resources.

Ironically, these attacks stem from parameters whose main purpose was to minimize latency. These attacks are more subtle than simply dropping QUIC handshake packets because they mislead at least one party into "believing" that everything is going well while causing it to waste time and resources deriving an initial key.

Resolving these types of attacks seems to be infeasible without reducing `scfg` and `stk` parameters to one-time use, because as long as these parameters persist for more than just a single connection, they could be used by the adversary to fake multiple connections while they remain valid. However, such restriction would prohibit QUIC from ever achieving 0-RTT connection establishment.

B. Packet Manipulation Attacks

Not all fields of QUIC packets are protected against adversarial manipulation. An attacker with access to the communication channel used by a client to establish a session with a particular server could flip bits of the unprotected parameters such as the connection id `cid` and the source-address token `stk` and lead the server and client to derive different initial keys which would ultimately lead connection establishment to fail. For a successful attack, the adversary has to make sure that all parameters modified in this way seem consistent across all sent and received packets with respect to any single party but inconsistent from the perspective of both parties participating in the handshake.

As shown in Section VII this type of attack does not raise concerns over the confidentiality and authenticity of communication that is encrypted and authenticated under the initial key, because even though the initial keys are different, they are not known by the adversary. Note also that if parties do not agree on an initial key, they cannot establish a session key in QUIC because the final server hello packet is encrypted and authenticated under the initial key. Therefore, these attacks also do not compromise the confidentiality and authenticity of communication encrypted and authenticated under the final key.

These packet manipulation attacks are smarter than just dropping QUIC handshake packets because the client and server progress through the handshake while having a

mismatched conversation, resulting in the establishment of inconsistent keys. This causes both parties to waste time and resources deriving keys and other connection state. In particular, the server performs all the processing required for a successful connection, unlike in attacks that simply drop QUIC handshake packets.

A simple strategy for mitigating this type of attack would be to have the server sign all such modifiable fields in its s_reject and s_hello packets (`cid` is unencrypted). However, this would incur the cost of computing a digital signature over all such modifiable parameters, which would in turn open another opportunity for a DoS attack in which the adversary, with IP spoofing, could send many initial connection requests on behalf of as many clients as it desires.

IX. Attack Results

In this section we discuss our implementations of the attacks we identified against QUIC in Section VIII. We target the Chromium implementation of QUIC[5] in our attacks, as this is the canonical implementation. Our attacks were developed in python using the scapy library.[6] We summarize our attacks, their properties, and impacts in Table I.

Replay Attacks.

Server Config Replay Attack. To conduct this attack, an attacker must first collect a copy of the target server's `scfg`. This can be done either by actively establishing a connection to the server or by passively listening for a client to attempt a connection. In either case, the server's `scfg` can be readily collected from a full, 1-RTT QUIC connection handshake.

Once the attacker has `scfg`, he waits for the target client to attempt to start a connection. When the attacker sees a c_hello message from the client, he can respond with a spoofed s_reject message using the collected `scfg` and randomly generated `stk` and `sno` values. Similar s_reject messages are the proper response to a client that either does not have a cached copy of the server's `scfg` or has a copy that is no longer valid. We assume that the attacker is closer to the client than the server is so that the s_reject message reaches the client prior to the response from the legitimate server. When the client receives this spoofed s_reject message, it promptly sends a new c_hello message using these new `scfg`, `stk`, and `sno` values.

When the real server receives this new c_hello message, it will attempt to validate it. However, the `stk` and `sno` values were randomly generated by the attacker and so are almost certain to fail the validation. In response to this failure, the server generates a new s_reject message containing `scfg` and new `stk` and `sno` values.

This new s_reject message provides the client with valid `stk` and `sno` values so another c_hello message could correctly complete the connection. However, when testing this attack, we found two further issues, the combination of which will always result in the connection terminating abnormally. The first issue is that each QUIC packet includes an entropy bit in its header and QUIC acknowledgment frames include a hash of these bits along with a list of unseen packets. The goal of this mechanism is to prevent Optimistic Ack attacks [2]. In our case, an acknowledgment frame will typically be included with the client's second c_hello message acknowledging the spoofed s_reject message. If the entropy bit in the attacker's spoofed s_reject message does not match the entropy bit in the server's real response, then the entropy hash in this acknowledgement will not validate and the server will abruptly terminate the connection.

The second issue is that a single QUIC connection provides multiple byte-streams for data transfer, and the QUIC handshake takes place within a special byte-stream reserved for connection establishment. This implies that all the c_hello, s_reject, and s_hello messages we have mentioned so far occur within the context of this byte-stream and have offset and length attributes. As a result, if the attacker's s_reject is not exactly the same size as the server's response, then this byte-stream is effectively broken. Any further messages from the server will be at offsets either above or below the client's position in the byte-stream. These messages will either be dropped or buffered forever. After ten seconds the client will abruptly terminate the connection because it is unable to complete the handshake.

In our tests, the combination of these two issues completely prevented the establishment of any QUIC connections. Connection attempts always terminated after either half a second, in the case of an entropy bit mismatch, or ten seconds, if the entropy bits matched, but the byte-stream was corrupted. Our python implementation requires that the attacker be about 20ms closer to the client than the server is, in order to create an s_reject message and have it reach the client before the server's legitimate response. However, with an optimized C implementation, this requirement could be significantly reduced.

Source-Address Token Replay Attack. The `stk` token is supposed to prevent packet spoofing by ensuring that a connection request originates at the IP address claimed. The `stk` is created by the server as part of the s_reject message. It contains the client's IP address and the current time, both encrypted. A client must present a valid `stk` in its c_hello message in order to perform a 0-RTT connection. However, the `stk` token must be presented prior to encryption being established. This means that any attacker who can sniff network traffic can collect `stk` tokens that can be used to spoof connection requests from a specific host for a limited period of time, by default 24 hours.

This attack operates by sniffing the network for s_reject messages from the target server. Each s_reject message contains a new `stk` being sent to some client. For each

[5]https://chromium.googlesource.com/chromium/src.git. We tested git revision `50a133b51fa9c6a3dc2b82ce9fedcf074859cd13` from October 1, 2014.

[6]http://www.secdev.org/projects/scapy/

Table I
DISCOVERED ATTACKS AND THEIR PROPERTIES

Attack Name	Type	On-Path	Traffic Sniffing	IP Spoofing	Impact
Server Config Replay Attack	Replay	No	Yes	Yes	Connection Failure
Source-Address Token Replay Attack	Replay	No	Yes	Yes	Server DoS
Connection ID Manipulation Attack	Manipulation	Yes	No	No	Connection Failure; server load
Source-Address Token Manipulation Attack	Manipulation	Yes	No	No	Connection Failure; server load
Crypto Stream Offset Attack	Other	No	Yes	Yes	Connection Failure

new `stk` seen, our attacker grabs the `stk`, the `scfg`, and the client's IP address and starts repeatedly spoofing 0-RTT connection attempts with random `cid`s from this client.

When the target server receives these requests, they appear to be legitimate 0-RTT connection requests. The `stk` will validate because the `stk` is replayed from a legitimate connection with an actual client at the spoofed IP address. As a result, the server will create a new connection for this request. This includes creating initial and forward-secure encryption keys and sending an s_hello message. At this point, the server believes it has completed connection establishment with the spoofed client.

In our tests, we used separate virtual machines for the attacker and server. We found that a single attacker starting with a single `stk` and sending packets at 200KB/sec was able to completely overwhelm our test server. The 2.4 GHz Intel(R) Xeon(R) CPU dedicated to our server was pegged at 100% utilization, and the operating system's out-of-memory killer eventually killed the server process after it exhausted the 3GB of memory allocated to the server's virtual machine.

It seems apparent that the QUIC server implementation in Chromium has no limitation on the number of connections that can be established from a single IP address. While we do not believe that this is the server implementation that Google uses in production, it is the only open-source QUIC server available. Additionally, much of the QUIC code is a library that we expect would be used by any production QUIC server. Note, however, that even if a limit on the number of connections from a single IP were added, this attack can inflate the number of connections to the server by this maximum number for *every* observed QUIC client.

MANIPULATION ATTACKS. Manipulation attacks subvert key agreement by causing the client and server to agree on different keys. This is done by modifying unprotected packet fields that are used as input to the key derivation process, in particular, the connection id `cid` or source-address token `stk`. We develop attacks against both of these parameters.

Connection ID Manipulation Attack. In this attack, the attacker is positioned on the path between the client and the server and re-writes the `cid` such that the client and server see different values. The handshake proceeds as normal, with the client requesting the `scfg`, if it does not have a cached copy, and then sending a c_hello message. This c_hello is processed by the server and an s_hello message sent in response. At this point, the server believes the connection has been successfully established. However, when the client receives the s_hello message sent by the server, it will fail to decrypt. This is because the `cid` is an input to the encryption key derivation process. Since the attacker changes the `cid`, the client and server will compute different encryption keys.

Unfortunately, decryption failure is not a sign of catastrophic handshake failure because it can be caused by reordering. In particular, packets encrypted with the forward-secure key will fail to decrypt prior to the reception of the s_hello message, which may be delayed due to reordering. As a result, packets failing decryption are buffered until the handshake completes. With the bad s_hello message buffered, the client will eventually timeout and retransmit its c_hello message. This process will repeat until the client's 10 second timer on connection establishment expires. At that point the connection will be terminated.

An error message will be sent to the server when the connection is terminated. However, this message will be encrypted with the initial encryption key, and thus the server will fail to decrypt it and will queue it for later decryption. Since it cannot decrypt the error message, the server will retain the connection state until the idle connection timeout expires. This timeout defaults to 10 minutes.

Source-Address Token Manipulation Attack. The goal of this attack is to prevent a client from establishing a connection, either denying access to the desired application or forcing the client to fall back to TCP/TLS. It requires an attacker positioned on the path between the client and the server who re-writes the `stk` such that the client and server see different values. It is important that the server always see the value it initially sent because it will validate `stk` later. To the client, however, `stk` is simply an opaque byte-string.

Any attempted connection request will proceed as normal, except that the attacker silently changes the `stk` values seen by client and server. The client requests the `scfg` from the server, which replies with the current `scfg` and an `stk` value. The client then sends a full c_hello to initiate the connection. The server receives and processes this c_hello and sends an s_hello message in response.

When the client receives this s_hello message sent by the server, it will fail to decrypt. This is because `stk` is an input into the encryption key derivation process, and the attacker has changed the `stk` value seen at the client. As a result,

the client and server will compute different encryption keys.

However, as mentioned previously, a decryption failure is not a sign of catastrophic handshake failure because this could happen due to reordering, if packets encrypted with the forward-secure key were received before the s_hello message. Hence, the client buffers the bad s_hello message for later decryption. Eventually the client times out and retransmits the c_hello message. This process will repeat until the client's 10 second timer on connection establishment expires. At that point the connection will be terminated.

The client will notify the server that it terminated the connection, but, unfortunately, this message will be transmitted encrypted with the initial encryption key. Hence, the server will be unable to process it and will continue to retain the connection state. This state will only be removed when the idle connection timeout expires, by default after 10 minutes.

We found that this attack effectively prevented all targeted QUIC connections. Further, all targeted connections experienced a 10 second delay before timing out.

OTHER ATTACKS While developing and testing the Server Config Replay Attack, we discovered an additional attack against QUIC. This attack results from QUIC treating handshake messages as part of a logical byte-stream, a detail abstracted out of the provable security analysis.

Crypto Stream Offset Attack. Recall that handshake messages are part of a logical byte-stream in QUIC. As a result, by injecting data into this byte stream an attacker is able to break the byte-stream and prevent the processing of further handshake messages. The attack results in preventing a client from establishing a connection using QUIC, either denying access to the desired application or forcing the client to fall back to TCP/TLS.

We create the attack by injecting a four character string into this handshake message stream. This injection is sufficient to prevent connection establishment. Our attacker listens for c_hello messages and responds with a spoofed reply containing the string "REJ\0" in the handshake message stream. As observed before, this breaks connection establishment because any messages from the server will now start at the wrong offset in the handshake message stream. Hence, they will be discarded or buffered indefinitely.

A connection that is attacked in this manner will either be terminated by the server because of an entropy bit mismatch or be timed out by the client after 10 seconds.

Note that an attacker requires very little information to launch this attack. No information is needed from the client's c_hello message, QUIC packet sequence numbers always start from 1, and the `cid` can be omitted from any packet other than the client's c_hello. As a result, all an attacker needs to launch this attack is knowledge of when a connection attempt will occur and the 4-tuple (server IP, client IP, server port, client port) involved. Of this 4-tuple, three items are already known: the server's IP, the client's IP, and the server's UDP port. If an attacker can guess the client's UDP port and when it will make a connection attempt, he can launch this attack completely blind.

In our tests, the ephemeral UDP port range was still too large to brute force within an RTT, at least with our python attacker. However, if the attacker can narrow the port range sufficiently, then an optimized C implementation could probably conduct this attack completely blind.

A. Attack Discussion

In this section we discuss how the attacks we found against QUIC relate to prior attacks on TCP and TLS. We find that attacking QUIC is not easier than TCP and TLS.

Source-Address Token Replay Attack. This QUIC attack is similar to the TCP SYN Flood attack where the attacker sends numerous spoofed TCP SYN packets to a server to overwhelm it and cause DoS [33]. The QUIC attack does almost the same thing, but the attacker is limited in the IP addresses he can use for spoofed packets. However, the impact of each spoofed packet is larger because QUIC needs to create encryption keys after receiving the initial packet.

The classic mitigation to SYN Flood is SYN Cookies, opaque tokens passed to the client by the server in the SYN-ACK and returned by the client on the final handshake ACK [33]. A SYN-Cookie encodes enough information so that the server does not need to keep state between the SYN and the final ACK and can serve as a proof that the client resides at its claimed IP address. The server creates the connection state structures only after the cookie is returned by the client, making it more difficult to overwhelm the server with spoofed connection requests.

An `stk` serves a similar purpose in preventing spoofed packets, with the difference being that its goal is to avoid the RTT incurred for a handshake. SYN-Cookies cannot be replayed because they are single use [33]. Because QUIC wants to support 0-RTT connections, it cannot make `stk`s single use, instead it limits their time and IP address validity. This allows an attacker to replay them.

QUIC Manipulation Attacks. These QUIC attacks are similar to the SSL Downgrade attack against a modern TLS implementation. In both cases, a Man-In-The-Middle attacker modifies packet fields and the attack is not discovered until the end of the handshake, after key generation and multiple RTTs.

SSL Downgrade works against SSL connections where both endpoints have SSL versions less than SSL 3.0 enabled. The goal is to downgrade the connection to an older, less secure version of SSL [34]. Basically, the attacker rewrites the connection request to indicate that the client only supports an older version of SSL, often version 2.0. The server and client then establish an SSL 2.0 connection, which the attacker can presumably compromise.

SSL 3.0 adds protection against this attack by adding a keyed hash of all the handshake messages to the Finished message and requiring the receiver to verify this hash [34].

This defense is effective, but the attack will only be detected at the end of the handshake.

Our QUIC Manipulation Attacks have similar outcome where the attack only becomes apparent at the end of the handshake when the keys generated by client and server do not match. Thus, the connection fails after a timeout, and the client may fall back to TCP/TLS. Since QUIC is designed to provide much lower latency for connection initiation than TCP/TLS, this compromises one of QUIC's main goals.

As discussed in section VIII-B, one simple mitigation would be to sign all modifiable fields in the server's s_reject and s_hello messages. However, this introduces signature computation overhead and a possible DoS attack.

QUIC Crypto Stream Offset Attack. This attack is similar to the TCP ACK Storm attack in that both result in the inability to transfer any more data over the target byte-stream and are caused by an attacker inserting data into the byte-stream.

The TCP ACK Storm attack [35] requires an attacker who can observe a TCP ACK packet of the target connection and then spoof data-bearing packets to both the client and the server. This data will be received and processed by the client and server and both will increase their ACK numbers as a result. Unfortunately, when an ACK is eventually sent by either client or server, it will appear to acknowledge data that the other side has not yet sent. TCP will drop such packets and send a duplicate ACK. At this point, the TCP byte-stream is effectively broken; no more data can be transferred because all packets will have invalid ACK numbers.

In much the same way, injection of data into a QUIC handshake stream disrupts the stream offsets and prevents any further handshake negotiation. This eventually results in connection timeout. Although a byte-stream is a convenient abstraction, it does not appear to be a good fit for handshake data. A message-stream, or sequence of messages, would be less prone to disruption in this manner.

X. Conclusions and Future Work

In this paper we provide the provable-security treatment of QUIC and assess its performance guarantees in the presence of adversaries. We provide a formal definition of a *Quick Connections (QC)* protocol, formally define a novel security model Quick ACCE (QACCE) appropriate for QC protocols, and show that QUIC satisfies QACCE under reasonable assumptions on its underlying building blocks.

Our analysis also reveals, however, that in the presence of attackers, QUIC may be unable to attain one of its main goals: 0-RTT connections. The adversary can make QUIC fall-back to TCP/TLS or cause the client and server to have an inconsistent view of their handshake which could lead to inconsistent states and more latency. Furthermore, such simple attacks could also be used to mount DoS attacks.

Our security definition is general and we plan to use our models to analyze other performance-driven security protocols, such as TLS version 1.3. Our work provides insights into the pitfalls of designing performance-driven secure protocols. In the future, we hope to explore methodologies for addressing the weaknesses of the QUIC protocol, which we have presented in this paper, and which may also be relevant to other protocols in this domain.

XI. Acknowledgements

We thank the anonymous reviewers for their comments. We thank Marc Fischlin and Bogdan Warinschi for useful discussions. We thank Adam Langley, Jim Roskind, Jo Kulik, Alyssa Rzeszutek Wilk, Ian Swett, Fedor Kouranov, and Robbie Shade for help with QUIC protocol details. We thank Andrew Newell for first introducing us to QUIC. Alexandra Boldyreva and Robert Lychev were supported in part by NSF CNS-1318511 and CNS-1422794 awards. Cristina Nita-Rotaru was supported in part by NSF CNS-1421815 award.

References

[1] T. Dierks and C. Allen, "The TLS protocol version 1.0," RFC 2246 (Proposed Standard), Internet Engineering Task Force, Jan. 1999.

[2] J. Roskind, "Quick UDP internet connections: Multiplexed stream transport over UDP," 2012. [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit

[3] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan, "Towards a SPDY'ier mobile web?" in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. ACM, 2013, pp. 303–314.

[4] R. Stewart, "Stream control transmission protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007.

[5] B. Ford, "Structured streams: A new transport abstraction," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. ACM, 2007, pp. 361–372.

[6] J. Roskind, "Experimenting with QUIC," The Chromium Blog, 2013. [Online]. Available: http://blog.chromium.org/2013/06/experimenting-with-quic.html

[7] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, Jan. 2012.

[8] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *CRYPTO*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 273–293.

[9] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the TLS protocol: A systematic analysis," in *CRYPTO*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, 2013, pp. 429–448.

[10] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Z. Bguelin, "Proving the TLS handshake secure (as it is)," 2014, IACR Cryptology ePrint Archive 2014: 182 (2014).

[11] M. Fischlin and F. Günther, "Multi-stage key exchange and the case of google's QUIC protocol," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014, pp. 1193–1204.

[12] R. Clayton, S. J. Murdoch, and R. N. Watson, "Ignoring the great firewall of China," in *Privacy Enhancing Technologies*. Springer, 2006, pp. 20–35.

[13] E. Rescorla, "New handshake flows for TLS 1.3," 2014. [Online]. Available: http://tools.ietf.org/html/draft-rescorla-tls13-new-flows-01

[14] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "Transport layer security (TLS) session resumption without server-side state," RFC 5077 (Proposed Standard), Internet Engineering Task Force, Jan. 2008.

[15] A. Langely, "Google, Personal communication," 2014.

[16] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is QUIC? Provable security and performance analyses," 2015, Full version of this paper. IACR Cryptology ePrint Archive 2015.

[17] A. R. Wilk, J. Kulik, F. Kouranov, and A. Westerlund, "Google QUIC team, Personal communication," 2014.

[18] W. Aiello, S. M. Bellovin, R. Canetti, J. Ioannidis, A. D. Keromytis, and O. Reingold, "Just fast keying: Key agreement in a hostile Internet," in *ACM Transactions on Information and System Security*, ser. TISSEC, vol. 7, no. 2. ACM, May 2004, pp. 1–30.

[19] B. Blanchet, "A computationally sound mechanized prover for security protocols," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006, pp. 140–154.

[20] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Springer, 2011, pp. 71–90.

[21] C. Fournet, M. Kohlweiss, and P. Strub, "Modular code-based cryptographic verification," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. ACM, 2011, pp. 341–350.

[22] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin, "Proving the TLS handshake secure (as it is)," in *CRYPTO*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds., vol. 8617. Springer, 2014, pp. 235–255.

[23] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 445–459.

[24] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02. ACM, 2002, pp. 98–107.

[25] M. Abdalla, M. Bellare, and P. Rogaway, "The oracle Diffie-Hellman assumptions and an analysis of DHIES," in *Topics in Cryptology – CT-RSA*, ser. Lecture Notes in Computer Science, D. Naccache, Ed., vol. 2020. Springer, 2001, pp. 143–158.

[26] D. A. McGrew and J. Viega, "The security and performance of the Galois/counter mode (GCM) of operation," in *Progress in Cryptology-INDOCRYPT 2004*. Springer, 2004, pp. 343–355.

[27] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols." in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS '93. ACM, 1993, pp. 62–73.

[28] ——, "Entity authentication and key distribution," in *CRYPTO*, ser. Lecture Notes in Computer Science, D. Stinson, Ed. Springer, 1994, vol. 773, pp. 232–249.

[29] C. Brzuska, N. P. Smart, B. Warinschi, and G. J. Watson, "An analysis of the EMV channel establishment protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. ACM, 2013, pp. 373–386.

[30] EMVCo LLC, "EMV ECC key establishment protocols," 2012. [Online]. Available: http://www.emvco.com/specifications.aspx?id=243

[31] M. Bellare, T. Kohno, and C. Namprempre, "Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 2, pp. 206–241, 2004.

[32] Y. Dodis, T. Ristenpart, J. P. Steinberger, and S. Tessaro, "To hash or not to hash again? (In)differentiability results for H^2 and HMAC." in *CRYPTO*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds. Springer, 2012, vol. 7417, pp. 348–366.

[33] W. Eddy, "TCP SYN flooding attacks and common mitigations," RFC 4987 (Informational), Aug. 2007.

[34] J. Clark and P. C. van Oorschot, "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 511–525.

[35] R. Abramov and A. Herzberg, "TCP ack storm DoS attacks," in *Future Challenges in Security and Privacy for Academia and Industry*, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds. Springer, 2011, pp. 29–40.