A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer

Antoine Delignat-Lavaud
Microsoft Research

Cédric Fournet
Microsoft Research

Bryan Parno
Carnegie Mellon University

Jonathan Protzenko
Microsoft Research

Tahina Ramananandro
Microsoft Research

Jay Bosamiya
Carnegie Mellon University

Joseph Lallemand
INRIA Nancy Grand-Est, LORIA

Itsaka Rakotonirina
INRIA Nancy Grand-Est, LORIA

Yi Zhou
Carnegie Mellon University

ABSTRACT

We investigate the security of the QUIC record layer, as standardized by the IETF in draft version 24. This version features major differences compared to Google's original protocol and prior IETF drafts. We model packet and header encryption, which uses a custom construction for privacy. To capture its goals, we propose a security definition for authenticated encryption with semi-implicit nonces. We show that QUIC uses an instance of a generic construction parameterized by a standard AEAD-secure scheme and a PRF-secure cipher. We formalize and verify the security of this construction in F*. The proof uncovers interesting limitations of nonce confidentiality, due to the malleability of short headers and the ability to choose the number of least significant bits included in the packet counter. We propose improvements that simplify the proof and increase robustness against strong attacker models. In addition to the verified security model, we also give concrete functional specification for the record layer, and prove that it satisfies important functionality properties (such as successful decryption of encrypted packets) after fixing more errors in the draft. We then provide a high-performance implementation of the record layer that we prove to be memory safe, correct with respect to our concrete specification (inheriting its functional correctness properties), and secure with respect to our verified model. To evaluate this component, we develop a provably-safe implementation of the rest of the QUIC protocol. Our record layer achieves nearly 2 GB/s throughput, and our QUIC implementation's performance is within 21% of an unverified baseline.

1 INTRODUCTION

The majority of the Web today relies on the network stack consisting of IP, TCP, TLS and HTTP. This stack is modular but inefficient: TLS starts only when the TCP handshake is complete, and HTTP must wait for completion of the TLS handshake. There have been recent efforts to cut this latency overhead (using, e.g., TCP fast open and TLS 1.3 0-RTT), but further gains require breaking the classic OSI model. QUIC was introduced by Google in 2012 [35] to increase performance by breaking layer abstractions, combining features that are part of TCP (fragmentation, re-transmission, reliable delivery, flow control), TLS (key exchange, encryption and authentication), and application protocols (parallel data streams) into a more integrated protocol, as shown in Figure 1. For example, consider two independent features that require an extra round-trip: source address validation (when a server wants to check that the source IP of an initial packet is not spoofed, currently part of the TCP handshake), and a TLS hello-retry request (when a server asks the client to send its key share over a different Diffie-Hellman group). Using QUIC, it is possible to combine both features in a single message, saving a full network round-trip.

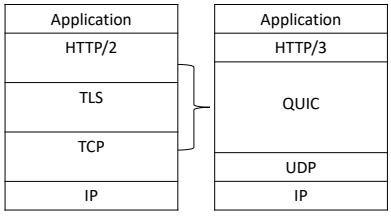


Figure 1: Modularity of current networking stack vs. QUIC

From a security standpoint, a fully-integrated secure transport protocol offers the potential for a single, clean security theorem, which avoids the pitfalls that arise when composing adversary models from different layers. Mismatched assumptions have caused several surprising cross-layer attacks against HTTPS. For instance, the CRIME [44] and BREACH [40] attacks illustrate the risk of composing application-layer compression with transport-layer encryption against adversaries that can both control the application plaintext and observe encrypted TLS traffic. Interestingly, transport compression exists in Google's version of QUIC, but was removed entirely by the IETF. Another example is cross-layer truncation attacks, such as the Cookie Cutter attack [12], where applications may perform dangerous side-effects based on incomplete received data due to a TCP error. With QUIC, it becomes possible to consider a single adversarial model for the application's transport and, in principle, to show that an application is secure against the UDP/IP message interface, which is very hard to achieve with TLS or IPSec.

Although QUIC was originally designed and implemented by Google for its own products, it is currently undergoing standardization by the IETF [33]. An explicit goal of the QUIC working group has been to ensure that QUIC inherits all the security properties of TLS 1.3, thus avoiding the lengthy formal protocol analysis effort that stretched out the TLS 1.3 standardization process for 4 years.

Submission to ACM CCS 2020, Due 20 January 2020, Orlando, TBD
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

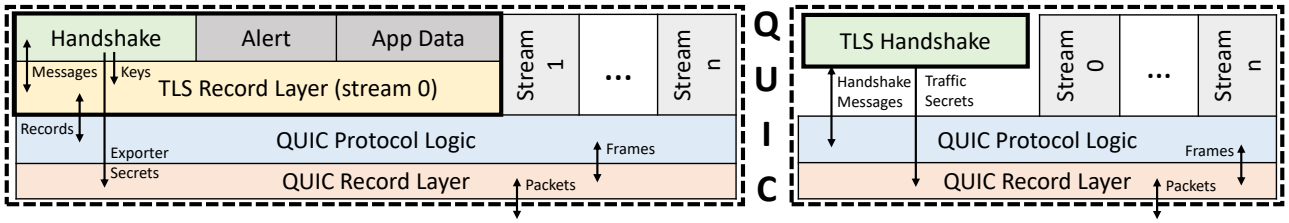


Figure 2: Internal modularity between TLS 1.3 and QUIC before (left) and after (right) draft 12. In the latter, the QUIC Protocol Logic is responsible for deriving keys from the TLS Handshake's traffic secrets.

Unfortunately, as we highlight in this paper (§2), the working group has failed to achieve that goal; the latest IETF drafts have progressively opened up many internal TLS abstractions, and thus, increasingly diverged from the context under which TLS 1.3 is proved secure. Entire features of TLS (including the record layer [49], version negotiation, the end-of-early-data message, hello retry, rekeying, and some key derivations) have been replaced in QUIC, often under different assumptions; furthermore, new cryptographic constructions, which have received little academic scrutiny, have been added. The standard has also drifted significantly away from Google's original version of QUIC, to the point that little of the early security analysis work on Google's QUIC ([27, 34, 37]) is relevant to the IETF version. Careful new analysis is required to precisely understand the security properties of QUIC data streams.

This paper aims to model and mechanically verify the security of the new features of the IETF's QUIC, focusing on what we refer to as the QUIC "record layer", i.e., the portion that handles packet formatting and encryption. This is an important step towards full end-to-end verification of the soon-to-be-standardized protocol (we discuss remaining steps below).

We begin by giving a new security definition (drawing heavily on prior work [8, 38]) to precisely capture the record layer's security goals (§3.2). We discovered and reported that early drafts of IETF's QUIC failed to achieve this definition; our feedback resulted in a new draft (#17) with an updated construction. We use F* [47], an ML-like language designed for verification, to develop a mechanized version of our security definition as well as a detailed functional specification for the new record layer construction. We mechanically prove the new construction is cryptographically secure, assuming the underlying primitives are. We also prove a number of useful properties of the specification (e.g., that packet numbers and headers are encoded injectively, and that decryption inverts encryption). These proofs identified two flaws in the IETF's reference implementation, and they uncovered interesting limitations of the IETF's security goal, as well as brittleness in the construction (e.g., places where easy-to-make implementation mistakes may undermine security). Hence, we propose improvements that simplify the proof and increase robustness against strong attacker models (§3.4).

Given our specification, we then develop a high-performance, low-level implementation (§4) of the record layer that offers cryptographic security, runtime safety, and functional correctness. Finally, we develop a proof-of-concept implementation of the QUIC protocol logic verified for memory safety, which we use to evaluate our verified implementation of the record layer, leveraging previously verified artifacts for the TLS handshake [14] and cryptographic primitives [41], to produce a verified implementation of the full IETF QUIC protocol. However, this is not yet an end-to-end verification of the protocol, as the proofs for the protocol logic would have to be significantly expanded and strengthened to connect the guarantees from the TLS handshake with those of the record layer and hence produce a single application-facing theorem for the entire implementation. We defer this to future work.

While we prove strong correctness and security properties for QUIC, like any verification result, our guarantees rely on the correctness of our formalized cryptographic definitions and of our verification tools. As we show in §5, while the performance of our record layer implementation is quite strong (GB/s), our protocol logic limits the performance of our overall QUIC reference implementation, leaving us 21% slower than our unverified baseline.

All of our specifications, implementations, and proofs are open source and available on GitHub.

2 QUIC BACKGROUND

Because of its integrated nature, it is hard to summarize QUIC. We introduce the aspects relevant to our security analysis, focusing on confidentiality, integrity and authentication goals, but refer to the IETF drafts [33, 49] for details. We also highlight deviations from QUIC's original goal of treating TLS as a black box.

TLS Handshake Interface In its early days, the IETF version of QUIC was designed as a strict encapsulation of TLS 1.3; that is, QUIC packets up to draft 12 contained full TLS records (as shown in Figure 2). The TLS record layer [11] has its own header format and supports fragmentation, encryption, padding, and multiplexing between handshake, alert and application data fragments. To reduce the redundancy between the TLS record layer and the QUIC record layer (e.g., double encryption and fragmentation of handshake messages), in current drafts, the QUIC protocol logic directly interacts with the TLS handshake and carries its messages directly in QUIC packets. The TLS 1.3 handshake negotiates the connection ciphersuite and parameters, authenticates the peer, and establishes fresh traffic secrets (used to derive record keys). It expects a duplex bytestream for reading and writing its messages, and a control interface for returning secrets and signaling multiple key changes:

- A first, optional early traffic secret is available immediately after handling a client hello message containing a pre-shared key (PSK). This allows the client to send application data in its very first message (no round-trip, "0-RTT"), but at a cost: 0-RTT messages are not forward secure and can be replayed.

- The (client and server) handshake traffic secrets are available after both hello messages are exchanged. A man-in-the-middle may corrupt the handshake traffic secret, so TLS messages are only secret against a passive attacker.
- The application ("1-RTT") traffic secrets are available once the server completes the handshake.

The record layer must be careful not to use traffic secrets until the handshake indicates that it is safe to do so. For instance, the application traffic secret is typically available on the server before the client's finished message is received, and the server must not try to decrypt 1-RTT packets before the finished has been checked.

Connection Establishment QUIC connections exchange sequences of encrypted packets over UDP. There are four main types of packets: initial packets (INIT) carry the plaintext TLS messages: client hello, server hello, and hello-retry request. Like all QUIC packets, they are encrypted, but their traffic secret is derived from public values, so the encryption is mostly for error detection. 0-RTT packets are encrypted using a key derived from the TLS early traffic secret, and similarly, handshake packets (HS) and 1-RTT packets are encrypted using keys derived from handshake traffic secrets and application data traffic secrets, respectively.

Packet Headers QUIC packets consist of a header and a payload. The type of a packet is indicated in the header. Initial, 0-RTT and handshake packets use long headers while 1-RTT packets use short headers, as depicted in Figure 3. Long headers include an explicit payload length to allow multiple packets to be grouped into a single UDP datagram. In contrast, the length of packets with short headers must be derived from the encapsulating UDP length. Hence, UDP datagrams contain at most one such packet, at the end.

Connection Identifiers Multiple QUIC connections can share the same IP address and UDP port by using separate connection IDs. This is particularly useful for servers, e.g., to route connections behind a load balancer. Clients may also use this feature to work around port restrictions and NAT congestion. Long headers include both a source and destination connection ID (of variable length). In its initial packet, the client picks its own source ID and the initial ID of the server as destination. Servers are expected to overwrite this initial choice by picking a new source ID in their response. Once the connection is established, the connection IDs are presumed to be authentic and of known length (either fixed, or encoded in the ID itself). Hence, short headers omit the source ID and the length of the destination ID. Connections IDs are a clear risk for privacy, since they correlate individual packets with sessions. QUIC encourages long-lived connections that can be migrated from one network path to another. For instance, if a mobile device switches from cellular to Wi-Fi, it is possible to migrate the connection on the new network while maintaining the current connection state, thus preventing the overhead of a re-establishing the connection. To manage this privacy risk, the connection peers have the ability to declare and retire connection IDs, and a privacy-conscious client may even change their ID in every packet.

Packet Numbers and Stream Encryption TLS over TCP relies on the in-order delivery of message fragments, and thus encrypts these fragments using a nonce computed from their implicit fragment sequence number. In contrast, QUIC packets are sent over UDP and may be dropped or delivered out of order. Therefore,

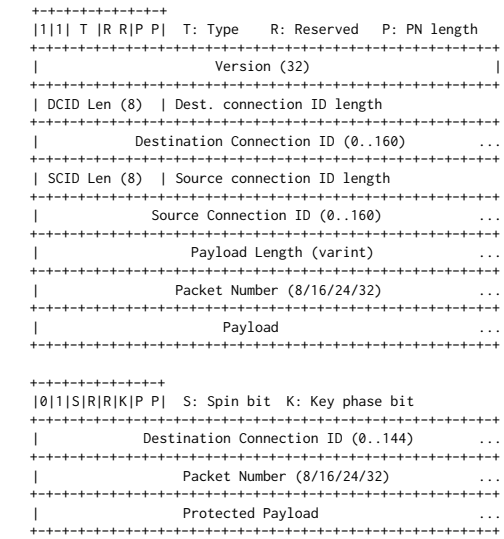


Figure 3: QUIC long (top) and short (bottom) packet headers. Initial, retry, handshake and 0-RTT packets use long headers, whereas 1-RTT packets use short headers.

encryption nonces cannot be implicit, which causes both communication overhead (full nonces are usually 12 bytes) and privacy concerns: if full nonces are sent on the wire, they can be used to map packets to connections and their users. QUIC connections can be very long-lived, and the most significant bits of the nonce become increasingly precise at identifying users. To address these concerns, QUIC's record layer introduces a new semi-implicit nonce construction that combines two ideas: only the least significant (and least privacy sensitive) bits of the nonce are sent on the wire; and those bits are further encrypted under a special header-protection key. This construction is detailed in §3.1.

In TLS over TCP, traffic secrets protect disjoint, successive authenticated encryption streams, with explicit termination messages to transition between keys and prevent truncation (end of early data message for 0-RTT, finished messages for the handshake, and the re-keying messages and closure alerts for 1-RTT). In QUIC, multiple keys may be active at the same time, which makes the logic for selecting and discarding keys much more involved. This adds new dangerous pitfalls that must be actively prevented: for instance, servers may receive 1-RTT data before the handshake is complete (this data is not yet safe to process), or clients may reply to 1-RTT messages with 0-RTT ones (the specification forbids this, but it is up to implementations to enforce). Normally, each direction of each traffic secret maintains its own packet counter; however, since only the client may send 0-RTT packets, they are acknowledge in 1-RTT packets, which means packet numbers are shared between 0-RTT and 1-RTT.

Transport Parameter Negotiation QUIC uses a special TLS extension to negotiate an extensible list of transport parameters (set or negotiated by the client and server). TLS guarantees that if the handshake completes, both parties agree on the parameters.

Version Negotiation QUIC defines a mechanism for the server to indicate that it wishes to use a different version than the one

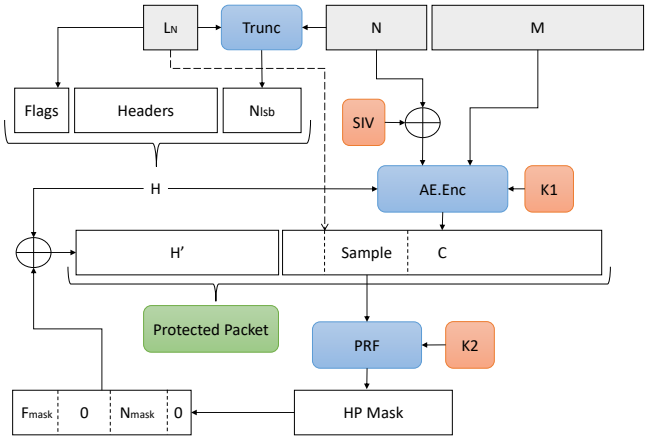


Figure 4: Overview of the QUIC packet encryption (QPE) construction with header protection (HP), parametrized by the AEAD scheme AE **and keyed pseudo-random function** PRF

proposed by the client. This mechanism uses a special version negotiation packet type, which is specified in a version-independent document [48] and contains the list of versions that the server is willing to use. Surprisingly, while previous drafts attempted to authenticate the version negotiation using the transport parameter negotiation through the TLS handshake, this feature has been removed in current drafts. Instead the specification states: *"Version Negotiation packets do not contain any mechanism to prevent version downgrade attacks. Future versions of QUIC that use Version Negotiation packets MUST define a mechanism that is robust against version downgrade attacks"*. It is unclear how future versions of QUIC will prevent version downgrade attacks. Regardless of which version the client supports, an attacker may always inject a version negotiation packet that only indicates version 1 support.

3 QUIC RECORD LAYER SECURITY

This section focuses on QUIC's new record layer. We first describe the cryptographic construction used by the record layer to encrypt both packet payloads and selected parts of packet headers (§3.1). We then present our cryptographic definition of security for single packets (§3.2) and then generalize it to streams of packets (§3.3). Finally, we discuss (§3.4) vulnerabilities that we discovered in earlier QUIC drafts, how the construction changed as a result, frailties that persist in the most recent draft, and our suggested improvements.

3.1 Background: QUIC Packet Encryption (QPE)

As explained in §2, QUIC packets consist of a header and a payload. The payload is encrypted using a standard algorithm for authenticated encryption with additional data (AEAD, abbreviated AE below), negotiated by TLS as part of its record ciphersuite. AEAD also takes the header (shown in Figure 3) in plaintext as additional authenticated data (AAD) and thus protects its integrity.

QUIC also applies a new and rather convoluted *header protection* scheme before sending the packet on the network. As discussed in §2, because QUIC uses UDP, it must include a packet number in each packet to cope with packet drops and reorderings. These packet numbers, however, pose a privacy risk, as a passive network observer can correlate packets and sessions, facilitating traffic analysis, particularly given QUIC's support for very long-lived, migratable sessions. Employing random nonces would waste too much space (because of the risk of birthday collisions, which cause catastrophic integrity failure, at least 128 bits would be required instead of the 8 to 32 bits of packet number that QUIC advocates). The situation is further complicated by QUIC's decision to use variable-length packet numbers, which means the length must also be hidden. Instead, QUIC settled on the more optimized (but somewhat exotic) construction shown in Figure 4 (the full implementation details of the construction appear in Appendix 8.1). In the discussion below, we refer to this construction as QPE[AE, PRF], since it is parameterized by the particular authenticated encryption (*AE*) and PRF algorithms selected during TLS ciphersuite negotiation.

The main inputs to the construction are the packet number N, the plaintext message M, the flags and QUIC-specific headers H, and the number L_N of least-significant bytes of N to send on the wire (between 1 and 4). The key materials consist of three cryptographic secrets: K_1 for the AEAD scheme AE, K_2 for the PRF, and SIV, a 12-byte static IV which is combined with the full 62-bit packet number N to form the AEAD nonce. The construction performs a normal AEAD encryption (using the message M as the plaintext, and the header H as the additional data).

QUIC then computes a header protection (HP) mask which it uses to hide the truncated packet number N_{lsb} and some bits of the flags—including those encoding the length L_N. The mask is computed by applying the PRF to a fixed-size fragment (called the *sample*) of the ciphertext C. The mask is split in two parts: the first byte is used to encrypt L_N, and the next 4 bytes are truncated to L_N and applied at the offset of the encrypted packet number. Because the L_N is hidden, it is difficult for the packet's recipient to know the boundary between the protected headers H' and the ciphertext C (and thus where the sample starts). This could be mitigated by choosing the last bytes of C for the sample, but QUIC solves this problem differently: the position of the sample is computed by *assuming* that $L_N = 4$, hence skipping the first $4 - L_N$ bytes of C. This requires all ciphertexts to be at least 20 bytes long, instead of 16 if the end of C was used. Since most AEAD tags are only 16 bytes, QUIC effectively requires a minimum of 4 bytes of plaintext in every packet (which can be achieve by adding padding frames). Since L_N is confidential, header decryption is also difficult to implement in constant time (we revisit this issue in §4.4).

3.2 Defining QUIC-Packet-Encryption Security

Although QPE[AE, PRF] is a new construction, it is similar to other constructions that have been proposed for nonce-hiding encryption, that is, constructions where the nonce is not an input of the decryption function but is instead embedded into the ciphertext. For instance, QPE is comparable to the HN1 construction of Bellare et al. [8], which comes with a security definition (AE2, §8.2) that captures the fact that the embedded nonce is indistinguishable from random, and a provable reduction to standard assumptions.

In line with their work, we define a security notion for partially nonce-hiding encryption that aims to capture the security goal of QUIC's approach to packet encryption. For simplicity, however, it

does not reflect the fact that header protection also applies of other parts of the header (i.e., the other flags, such as the key-phase bit).

Notation Following the well-established approach of Bellare and Rogaway [9], we define the security of a functionality F as the probability (called the *advantage*) $\epsilon_{\mathsf{G}}(\mathcal{A})$ that an adversary $\mathcal{A}$, interacting with a game $\mathsf{G}^b(F)$ (whose oracle are parametrized by a random bit b), guesses the value of b with better than random chance, i.e. $\epsilon_{\mathsf{G}}(\mathcal{A}) = |2\Pr[\mathsf{G}^b(\mathcal{A}) = b] - 1|$. By convention, when $b = 0$, the oracles of G behave exactly like the functions in the functionality F; we also refer to G^0 as the *real functionality*. In contrast, the oracles of G^1 capture the perfect intended behavior, or the *ideal functionality*, which is typically expressed using shared state. The reason we insist on real-or-ideal indistinguishability games will be more apparent in §4, as we formalize type-based security verification using idealized interfaces. We refer the reader to Brzuska et al. [18] for a detailed introduction to the methodology.

Definitions To illustrate our notation, consider the standard security definition for a keyed pseudo-random functionality F, which offers a single function, namely compute : $\{0,1\}^{\ell_K} \times \{0,1\}^{\ell} \to \{0,1\}^{\ell}$. The real functionality (b=0) just evaluates the function. The ideal functionality (b=1) is a random function, implemented using a private table T to memoize randomly sampled values of length ℓ.

Game $\mathsf{PRF}^b(\mathsf{F})$

$T \leftarrow \emptyset$

$k \xleftarrow{\$} \{0,1\}^{\mathsf{F}.\ell_K}$

Oracle Compute(X)

if $b = 1$ **then**

  if $T[X] = \perp$ **then** $T[X] \xleftarrow{\$} \{0,1\}^{\mathsf{F}.\ell}$

  return $T[X]$

return $\mathsf{F}.\mathsf{compute}(k, X)$

Our security definition for packet encryption is a refinement of the standard notion of authenticated encryption security, AE1, shown below, for a symmetric encryption scheme SE1. In the definition, ideal encryption is implemented by a lookup in a global table indexed by the nonce, ciphertext and header.

Game $\mathsf{AE1}^b(\mathsf{SE1})$

$T \leftarrow \emptyset;\ k \xleftarrow{\$} \mathsf{SE1}.\mathsf{gen}()$

Oracle Decrypt(N, C, H)

if $b = 1$ **then**

  $M \leftarrow T[N, C, H]$

else

  $M \leftarrow \mathsf{SE1}.\mathsf{dec}(k, N, C, H)$

return M

Oracle Encrypt(N, M, H)

assert $T[N, \_, \_] = \perp$

if $b = 1$ **then**

  $C \xleftarrow{\$} \{0,1\}^{|M|+SE1.\ell_T}$

  $T[N, C, H] \leftarrow M$

else

  $C \leftarrow \mathsf{SE1}.\mathsf{enc}(k, N, M, H)$

return C

In AE1, the correct nonce must be known by the recipient of the message in order to decrypt. In TLS, the nonce is obtained by counting the packets that have been received, but this does not work in QUIC where packets are delivered out of order. Instead, the recipient only knows an approximate range (which depends on L_N), while the fine-grained position in that range (the encrypted packet number in the packet headers) is obtained by header decryption.

We formalize the security goal via security definition AE5 for encryption with variable-sized, semi-implicit nonces. The idea behind this definition has appeared before, and is informally described in an unpublished note by Namprempre, Rogaway and Shrimpton [38] where it is referred to as AE5 in reference to the 5 inputs of the encryption function; however, to our knowledge, the definition has never been formalized, and no construction has been proposed or proved to implement this definition.

Game $\mathsf{AE5}^b(\mathsf{E})$

$T \leftarrow \emptyset;\ k \xleftarrow{\$} \mathsf{E}.\mathsf{gen}()$

Oracle Decrypt(N_m, C, H)

if $b = 1$ **then**

  $M \leftarrow T[N, C, H]$ **for** N s.t. $\mathsf{msb}(N) = N_m$

else

  $M \leftarrow \mathsf{E}.\mathsf{dec}(k, N_m, C, H)$

return M

Oracle Encrypt(N, L_N, M, H)

assert $T[N, \_, \_] = \perp$

if $b = 1$ **then**

  $C \xleftarrow{\$} \{0,1\}^{L_N+|M|+\mathsf{E}.\ell_T}$

  $T[N, C, H] \leftarrow M$

else

  $C \leftarrow \mathsf{E}.\mathsf{enc}(k, N, L_N, M, H)$

return C

Notice that while E.enc takes in the full nonce N and its encoding length L_N, E.dec only takes in the latter. Our definition generalizes both standard AE1 (when $L_N = 0$), and nonce-hiding encryption (AE2) when $L_N = |N|$.

Security Theorem Our main security result reduces the security of QUIC's packet encryption construction, QPE[AE,PRF], to the AE1 security of AE and the security of the PRF.

Theorem 1 (QPE Security). *Given an adversary $\mathcal{A}$ against the* $\mathsf{AE5}^b(\mathsf{QPE}[\mathsf{AE},\mathsf{PRF}])$ *game, we construct adversaries $\mathcal{A}'$ against* $\mathsf{AE1}^b(\mathsf{AE})$ *and $\mathcal{A}''$ against* $\mathsf{PRF}^b(\mathsf{PRF})$ *such that:*

$$\epsilon^{\mathsf{QPE}}_{\mathsf{AE5}}(\mathcal{A}) \le \epsilon^{\mathsf{AE}}_{\mathsf{AE1}}(\mathcal{A}') + \epsilon^{\mathsf{PRF}}_{\mathsf{PRF}}(\mathcal{A}'') + \frac{q_e(q_e-1)}{2^{\mathsf{PRF}.\ell+1}}$$

where q_e is the number of encryptions performed, and $\mathsf{PRF}.\ell$ is the output length of the PRF.

§4.5 describes our mechanized proof in more detail, but at a high level, we consider a variant of $\mathsf{AE5}^b$ where the game stops if, after sampling C (in the ideal case), the oracle detects that there is some other ciphertext C' in T such that $C[L_N..L_N + F.\ell] = C'[L'_N..L'_N + F.\ell]$. In other words, in this variant of the game, the sample values extracted from the ciphertext for header protection are unique. The probability of any collision between any independent, randomly-distributed samples after q_e queries to the encryption oracle is $\frac{q_e(q_e-1)}{2^{\mathsf{PRF}.\ell+1}}$. For the rest of the proof, we *implement* the adversaries $\mathcal{A}'$ and $\mathcal{A}''$ based on idealized implementations of the AE1 and PRF games, which we describe in §4.5.

For simplicity, we wrote single-instance versions of the AE5 and PRF games. In our mechanization of the proof, all games are multi-instance, but the state of each instance is independent. The bound of Theorem 1 generalizes to the multi-instance setting by multiplying the collision term by the number of honest instances.

3.3 Packet Stream Security

The definition of AE5 only provides partial packet number confidentiality (for the least significant bits). We now introduce a notion of nonce-hiding stream encryption security, for a stateful encryption scheme SE that encrypts message M and protects header H. Note that unlike standard stream encryption security notions, this allows messages to be decrypted in a different order than they are encrypted, but with a restriction: messages that are too far from the most recent successfully decrypted packet will fail to decrypt. We assume L_N is fixed for this definition; i.e., the number of encrypted packet-number bits is fixed between packets.

Game $\mathsf{NHSE}^b(L_N, \mathsf{SE})$
$c_e \leftarrow 0; c_d \leftarrow 0;\ T \leftarrow \emptyset;$
$S \xleftarrow{\$} \mathsf{SE.gen}()$

Oracle $\mathsf{Encrypt}(M, H)$
if $b = 1$ **then**
  $C \xleftarrow{\$} \{0,1\}^{L_N+|M|+\mathsf{E}.\ell_T}$
  $T[C, H] \leftarrow (c_e, M)$
  $c_e \leftarrow c_e + 1$
else
  $C, S' \leftarrow \mathsf{SE.enc}(S, M)$
  $S \leftarrow S'$
return C

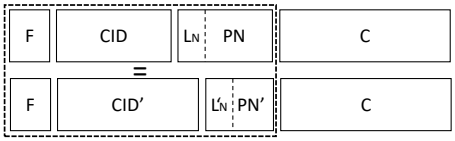
Oracle $\mathsf{Decrypt}(C, H)$
if $b = 1$ **then**
  $c, M \leftarrow T[C, H]$
  if $|c - c_d| >= 2^{L_N-1}$ **then**
    return $\perp$
  $c_d \leftarrow max(c, c_d)$
else
  $M, S' \leftarrow \mathsf{SE.dec}(S, C, H)$
  $S \leftarrow S'$
return M

The actual QUIC stream encryption (QSE) construction is similar to the ideal functionality of NHSE: a counter c_e is used to send packets with QPE, and a reference c_d stores the highest decrypted packet number received. In our verified implementation, we prove constructively that for an adversary $\mathcal{A}$ of NHSE^b[QSE], we can program an adversary $\mathcal{A}'$ for $\mathsf{AE5}^b$[QPE] with the same advantage.

3.4 History and Improvements to QPE

Implicit-Nonce Length Malleability While the table T in AE5 is not indexed by L_N, the definition still implies that decryption must authenticate the implicit nonce length L_N chosen by the sender. To illustrate this point, we consider the construction QPE'[AE,PRF] used in draft 16 of IETF's QUIC, where L_N is encoded in the most-significant 2 bits of the packet number field of the QUIC headers (instead of in the 2 least-significant bits of the flags as done in QPE[AE,PRF] in more recent drafts).

Recall that in short headers, the destination connection ID (CID) is concatenated with the packet number, and that the length of the CID is implicit. Since L_N is variable, an adversary may try to cause the sender and recipient of a packet to disagree on the position where the CID ends and the packet number begins:



Conveniently for the attacker, the XOR between the mask and the packet number is easily malleable: by flipping the first two encrypted bits, the attacker can force the receiver to interpret the length differently. This yields an easy way to win the AE5 game: the attacker first calls $\mathsf{Enc}(N, 2, M, F||CID)$ then $\mathsf{Dec}(N', C', F||CID||X)$ for each of the 256 possible values of the byte X, where $N' = N[0..11]$ and $C' = C[1..|C|]$. One of the values of X will result in the correct AAD; hence $\mathcal{A}$ can return 0 if any of the decryptions succeeds (because of the mismatched N' length, the nonce of the decryption can work in the ideal variant).

Although this attack may be hard to exploit in practice, it raises the question of whether QUIC expects the peer connection ID to be authenticated by the TLS handshake. The specification does not state it as an explicit goal; however, some working group members argue that the authenticity of the connection IDs follows from their inclusion in headers authenticated through AAD. This claim is disputable: the CIDs are negotiated in the initial messages, whose keys are derived from public values. In draft 16, an active adversary can inject a retry to client to change the server's CID. If the attacker tampers with one byte of CID, the attack succeeds with 2^{-8} probability, which is practical on every packet.

We submitted these observations to the IETF [39], and proposed to concatenate the 2 bits of L_N with the 62-bit packet number when constructing the AEAD nonce. The goal of the change was to ensure that L_N is authenticated regardless of potential malleability issues in the formatting of the AAD headers. However, in draft 17, the implemented change was to move L_N to the least significant bits of the flags, which is sufficient to prove the security of the construction but requires the processing of the mask to depend on L_N. The question of CID authentication remains open. In order to prevent an adversary from injecting a retry to the client, a new transport parameter was introduced that records the initial destination CID. The presence of this transport parameter indicates that a retry occurred, and peers are expect to check agreement on the initial CID. However, it is not clear that this is sufficient to guarantee agreement on the CID. For instance, if the server sends a legitimate retry message, the attacker may modify the new server CID, and the change is not detectable by the client. We have found that some implementations prevent this attack (by authenticating the new server CID in the returned token), but it is not mandated by the specification.

Improving the QUIC Construction Although Theorem 1 provides useful guarantees, we are still concerned about weaknesses in the QPE construction:

- The authentication of L_N depends on the AAD security of the payload, which in turns depends on the non-malleability of the formatting of QUIC headers. This is brittle in short headers, especially as some implementations may use unsafe representations of their CID, such as null-terminated strings.
- The construction collapses if the receiver uses the decrypted packet number before the successful decryption of the whole payload. While the QUIC specification explicitly forbids this behavior, we observe in practice that many implementations do not bother decrypting the payload if they know in advance that the decryption will fail. This shortcut provides a timing oracle to abuse the malleability of the XOR encryption of the packet number—allowing an attacker to do efficient range checks by flipping the last 2 bits of the flags.
- The construction is difficult to implement in constant time. The natural implementation strategy is to first decrypt L_N, and then truncate the rest of the mask to L_N.

Interestingly, Bellare et al. [8] propose another construction called HN2 (shown in Figure 11) that uses a block cipher (or pseudo-random permutation) instead of a PRF and an XOR. The idea of this construction is to encrypt with the block cipher the concatenation of the packet number and AEAD ciphertext.

We propose a variant of QPE based on HN2 in Figure 5. This variant makes it much more difficult for an adversary to selectively flip bits in the packet number. The security proof is also simpler, as the collision term is accounted for by the idealization of the PRP.

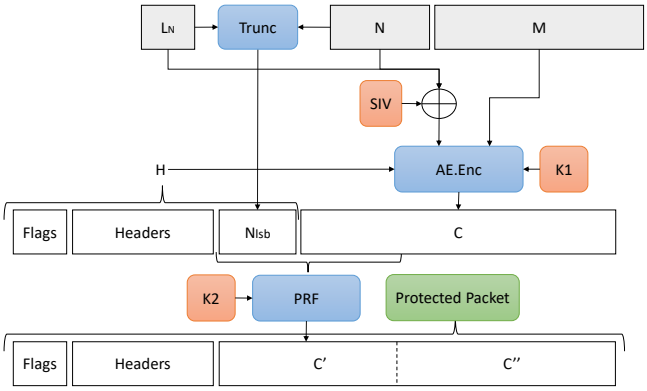


Figure 5: Our proposed HN2-based construction for QPE

4 VERIFIED CORRECTNESS AND SECURITY

We contribute a reference implementation of the QUIC record layer and support it with machine-checked proofs of its intended functional and security properties, as described in §3. Our verified specification, code, and proofs are done within the F⋆ proof assistant.

4.1 F⋆ (review) and Initial QUIC Definitions

F⋆ is a language in the tradition of ML, equipped with dependent types and an effect system, which allows programmers to prove properties of their code. A popular flavor is proofs by refinement, wherein a complex *implementation* is shown to be functionally equivalent to a concise, crisp *specification*. F⋆ relies on a weakest precondition calculus to construct proof obligations that can then be discharged using a mixture of automated (via Z3 [23], an SMT solver), semi-automated (via user tactics), and manual proofs.

To execute an F⋆ program it must be *extracted* to OCaml or F#. If the run-time portions of a program fit within Low⋆, a low-level subset of F⋆, then the program can be compiled to portable C code via a dedicated compiler, KreMLin [42]. This allows programmers to rely on the full power of F⋆ for proofs and specifications, knowing that they are erased at extraction-time; only executable chunks of code need to be in Low⋆. The Low⋆ methodology has been successfully used by the HACL⋆ cryptographic library [51], the EverCrypt cryptographic provider [41], the EverParse verified parser library [43] and several others. We now illustrate basic F⋆ concepts, using truncation and recovery of packet numbers as an example of how to author specifications in F⋆. A subsequent section (§4.4) dwells on Low⋆ implementations.

Truncated packet numbers occupy 1 to 4 bytes, depending on the user's choice. Packet numbers, once decoded and recovered, are in the range $[0; 2^{62})$. The truncated-number length is included in the header as a two-bit integer, while packet numbers mandate a 62-bit integer. Both are defined as refinements of natural integers:

```
type nat2 = n:nat{n < 4}        type nat62 = n:nat{n < pow2 62}
```

One can define the upper bound on the value of a truncated packet number (named below "npn" for "network packet number") given the length of its encoding in bytes. To this end, we define bound_npn, a function of a single argument pn_len, using a let-binding.

```
let bound_npn (pn_len: nat2) = pow2 (8 * (pn_len + 1))
```

Using these definitions, we define truncation as a function of the target length (in bytes) and the full packet number:

```
let truncate_pn (pn_len: nat2) (pn: nat62):
  npn:nat{ npn < bound_npn pn_len } = pn % bound_npn pn_len
```

Undoing the truncation for incoming headers is more involved, since it is clearly not injective. Hence, QUIC uses an expansion operation whose correctness is guaranteed when the packet number to recover is within a window of width bound_npn pn_len centered on the maximal packet number received so far. We express it in F⋆ via the following predicate:

```
let in_window (pn_len:nat2) (max_pn:nat62) (pn:nat62) =
  let h = bound_npn pn_len in
  (max_pn+1 < h/2 ∧  pn < h) ∨
  (max_pn+1 ≥  pow2 62 − h/2 ∧  pn ≥  pow2 62 − h) ∨
  (max_pn+1 − h/2 < pn ∧  pn ≤  max_pn+1 + h/2)
```

The first and second clauses of the disjunction shift the window when it under- or overflows the interval $[0, 2^{62})$. Proving the correctness of packet number expansion revealed two errors in the IETF reference implementation [33, Appendix A]: an off-by-one in the 3rd case, and an overflow in the second case. Both are fixed in the editor's copy following our report. Below, we give the types of the patched function and its verified inverse property, which ensures it returns the full packet number if it is within the window.

```
val expand_pn : pn_len:nat2 →
  max_pn:nat{max_pn+1 < pow2 62} →
  npn:nat{npn < bound_npn pn_len} →
  pn:nat62{in_window pn_len max_pn pn}
val lemma_parse_pn_correct : pn_len:nat2 →
  max_pn:nat{max_pn+1 < pow2 62} → pn:nat62 →
  Lemma (requires in_window pn_len max_pn pn)
    (ensures expand_pn pn_len max_pn (truncate_pn pn_len pn) = pn)
```

As expressed by the precondition of lemma_parse_pn_correct, the sender must choose the value of pn_len carefully for the predicate in_window pn_len max_pn pn to hold, which in turn ensures npn will expand to the intended packet number. However, the sender cannot predict the exact value of max_pn, which is the highest packet number received by the receiver. She can only know a range for this number: it must be greater than the last acknowledged packet number last_min, and lower than the last sent packet number last_max. To be certain that the chosen pn_len will always lead to the intended expanded packet number pn, the sender must therefore ensure that for any value max_pn∈[last_min,last_max], in_window pn_len max_pn pn holds. This condition can be checked by the following function:

```
let in_all_windows (pn_len:nat2) (last_min last_max pn:nat62) : bool =
  let h = bound_npn pn_len in
  (pow2 62 − h ≤  pn || last_max+2 − h/2 ≤  pn) &&
  (pn ≤  h−1 || pn ≤  last_min+1 + h/2)
```

We prove that this function has the intended behavior. The sender then simply has to pick the shortest pn_len that passes this check. Doing so yields a function with the following signature

```
val decide_pn (last_min last_max pn:nat62)) : Pure (option nat2)
  (requires last_min ≤  last_max)
  (ensures (function None →
    ∀ (pn_len:nat2). ¬ (in_all_windows pn_len last_min last_max pn)
```

```
| Some pn_len →
  in_all_windows pn_len last_min last_max pn ∧
  (∀ (pn_len':nat2{pn_len' < pn_len}).
    ¬ (in_all_windows pn_len' last_min last_max pn))))
```

whose post-condition expresses that it returns the optimal (i.e. shortest) possible pn_len. Note that there might not always exist a suitable one, for instance if the range [last_min, last_max] is too wide, in which case the function returns None.

4.2 F⋆ Specification of Packet Encryption

We outline the remainder of our specification in a bottom-up fashion, starting with parsers and serializers and leading to a high-level specification of packet encryption and decryption.

Parsing For parsing and serializing, we use and extend the EverParse framework [43]. EverParse is an existing combinator library for specifying, generating, and extracting parsers and serializers, written in F⋆/Low⋆. EverParse focuses on binary formats like those of QUIC, and extracts to zero-copy validators and serializers in C.

For this work, we extended EverParse with a notion of bit fields, which it previously lacked. This allowed us to express the variable-length encoding used by the QUIC spec within EverParse. We also expressed packet-number truncation and recovery using EverParse, which yielded a more concise and efficient proof of correctness.

We expressed the rest of packet-header parsing and serializing using EverParse combinators, yielding an automatic proof **(i)** of parser correctness, i.e., the parser and serializer are inverses of each other, and **(ii)** of injectivity, ensuring there is at most one possible binary representation of QUIC headers, given the packet number window and the choice of lengths of connection identifiers for short packets. To prove the uniqueness of the binary representation, we needed to impose a minimum-length restriction on the representation of variable-length integers.

From Wire Formats to Abstract Headers Parsers and serializers operate on sequences of bytes (using the F⋆ type bytes), as well as vlbytes $min\ max$, an EverParse refinement of bytes of variable length ℓ such that $min \le \ell \le max$, used below to represent connection IDs. At the boundary of EverParse-based specifications, we abstract over raw bytes and switch to high-level, structured values, using an inductive type with separate cases for long and short headers (reflecting the header format of Figure 3).

```
type header =
| MLong: version: U32.t →
  dcid: vlbytes 0 20 → scid: vlbytes 0 20 →
  spec: long_header_specifics → header
| MShort:
  spin: bool → key_phase: bool → dcid: vlbytes 0 20 →
  packet_number_length: packet_number_length_t →
  packet_number: uint62_t → header
```

The type long_header_specifics, elided here, contains the encoded packet-number length, the truncated packet number, and the payload length, with a special case for retry packets. The remainder of our QUIC specifications, including formatting, parsing, as well as the correctness and injectivity lemmas rely on the high-level header type. We discuss our proof of the correctness of conversion between high-level and low-level header representations in §4.3.

Side-channel-Resistant Header Protection Leveraging the header type above, we specify header protection, using a custom-built F⋆ library of specification helpers for manipulating byte sequences with binary operators. Further specification refinements are needed. For packet-number masking, we refine the initial specification into a more operational one that avoids a common implementation pitfall that results in a side-channel leak. We then verify the low-level implementation against the side-channel-free specification.

Agile Cryptography The QUIC specification inherits a large body of cryptographic primitives mandated by the TLS 1.3 standard: HKDF expansion and derivation, AEAD, and the underlying block ciphers for the packet-number mask.

Rather than rewrite this very substantial amount of specification, we reuse EverCrypt [41], an existing cryptographic library written in F⋆/Low⋆. EverCrypt specifies and implements all major cipher suites and algorithms, including HKDF for SHA2 and all major variants of AEAD (ChachaPoly, AES128-GCM, AES256-GCM).

More importantly, EverCrypt offers agile, abstract specifications, meaning that our QUIC specification is *by construction* parametric over the algorithms chosen by the TLS negotiation. Lemmas such as lemma_encrypt_correct (§4.3) are parameterized over the encryption algorithm ("ea"), and so is our entire proof. This means our results hold for *any* existing or future AEAD algorithm in EverCrypt.

EverCrypt uses a simple model for side-channel resistance where any data of type bytes is secret; our QUIC spec uses information-flow labels instead to distinguish plain texts and cipher texts. We omit these technicalities.

4.3 Functional Correctness Properties

We have proven F⋆ lemmas showing that our specification (and hence, our verified implementation described in §4.4) of the draft 24 specification has the expected properties, including:

(1) correctness of the packet-number decoding (§4.1)
(2) correctness and injectivity of header parsing
(3) correctness of header and payload decryption for packets.

We elaborate on the latter two proofs below. In the process of developing these proofs, we uncovered several issues with the current IETF draft. For example, as described in Section 4.1, while specifying packet-number recovery, we discovered that the QUIC draft omits an overflow condition on the window size. We also demonstrated that the whole QUIC specification is parameterized over destination connection ID lengths, meaning that non-malleability depends on proper authentication of connection IDs. We have proposed simple fixes, notably embedding the length L_N of the truncated packet number into the AEAD nonce (§3.4).

Header Parsing Proofs First, header parsing is correct, meaning that parse_header inverts format_header.

```
val lemma_header_parsing_correct: ... → Lemma
  (parse_header cid_len last (format_header (append h c)) = Success h c))
```

For safety reasons [43], parsers should also be injective (up to failure). The parse_header function enjoys this property but only for a given connection-ID length.

```
val lemma_header_parsing_safe: ... → Lemma (requires ... ∧
   parse_header cid_len last b1 = parse_header cid_len last b2)
  (ensures parse_header cid_len last b1 = Failure ∨ b1 = b2)
```

Encryption Correctness Based on our parsing correctness lemma, we can prove the correctness of packet encryption. Our proof is based on an intermediate lemma about header encryption, and uses the idempotence property of XOR and the functional correctness lemma of EverCrypt's specification of AEAD.

```
val lemma_encrypt_correct:
  a:ea (* the AE algorithm negotiated by TLS *) →
  k:lbytes (ae_keysize a) (* the main key for this AE algorithm *) →
  siv: lbytes 12 (* a static IV for this AE algorithm *) →
  hpk: lbytes (ae_keysize a) (* the header key for this AE algorithm *) →
  h: header  (* Note the condition on the CID length below *) →
  cid_len: nat {cid_len ≤ 20 ∧ (MShort? h ⇒ cid_len = dcid_len h)} →
  last: nat{last+1 < pow2 62 } →
  p: pbytes' (is_retry h) { has_payload_length h ⇒
    U64.v (payload_length h) == length p + AEAD.tag_length a } →
  Lemma (requires is_retry h ∨ in_window (pn_length h − 1) last (pn h))
(ensures decrypt a k siv hpk last cid_len (encrypt a k siv hpk h p) = OK h p)
```

The final lemma is rather verbose due to the number of parameters and conditions, but merely states that decrypt inverts encrypt. It explicitly inherits all limitations of previously-defined functions: the window condition for packet-number decoding, and the need to provide the correct length of the connection ID.

4.4 Low-Level Record-Layer Implementation

We now briefly describe the low-level implementation of our QUIC record layer, leveraging both the EverCrypt and EverParse libraries. We follow the proof by refinement methodology (§4.1) and show that our Low★ implementation is functionally equivalent to the specification from §4.2. By virtue of being written in Low★, the code is memory-safe and these guarantees carry over to the generated C code [42]. §5 reports code statistics and performance.

Overview Verifying code in Low★ differs greatly from authoring specifications (§4.2). First, all computations must be performed on machine integers, meaning that computations such as in_window must be rewritten to avoid overflowing or underflowing intermediary sub-expressions — a common source of pitfalls in unverified code. Second, all sequence-manipulating code must be rewritten to use arrays allocated in memory. This requires reasoning about liveness, disjointness and spatial placement of allocations, to prevent runtime errors (e.g. use-after-free, out-of-bounds access).

Parsers Our specification of message formats is expressed using the EverParse library. Just like for specifications, we have extended EverParse's low-level parsers and serializers with our new features (bounded integers; bitfields), and we have written low-level, zero-copy parsers and serializers for QUIC message formats directly using the EverParse library. We unfortunately were unable to use the front-end language for EverParse, dubbed QuackyDucky [43], resulting in substantial manual effort; we hope future versions of the tool will capture the QUIC format.

Internal State Our QUIC library follows an established pattern [41] that revolves around an indexed type holding the QUIC state. This state is kept abstract from verified clients and, interestingly, from C clients as well, using an incomplete struct type to enforce C abstraction. Clients do not know the size of the C struct, meaning that they cannot allocate it and are forced to go through the allocation function we provide. We go to great lengths to offer idiomatic C-like APIs where functions modify out-params and return error codes, which requires extra memory reasoning owing to the double indirection of the out parameters.

Encryption and Decryption When called, encrypt outputs the encrypted data and the freshly-used packet number into two caller-allocated out-params. The decrypt function fails if and only if the spec fails, consumes exactly as much data as the spec, and when it succeeds, fills a caller-allocated struct. To maximize performance, our decryption implementation operates in-place and performs no allocation beyond temporaries on the stack.

4.5 Type-Based Cryptographic Security

To prove the cryptographic security of our implementation, we first convert the security games from §3.2 into F★ programs parametrized by the Boolean b, following the style of previous work [11, 13, 30]. As an example, consider the PRF^b game: as shown below, we can represent instances using an abstract type key, which is implemented as a random key k if b is false, or a PRF table T that maps input blocks to random outputs when b is true. We also index instances by an id type, such that the adversary can declare which instances are honest and which are corrupted, before they are created. We also assume a safety predicate let safe (i:id) = honest i && b.

```
(* PRF: Idealized Interface *)
let len = 16 (* Block size *)  let klen = 16 (* Key size *)
abstract type key (i:id)
val ideal: i:id{safe i} → key i → map (lbytes len) (lbytes len)
val real: i:id{¬ (safe i)} → key i → lbytes klen
val create: i:id{safe i} → ST (key i)
val coerce: i:id{¬ (safe i)} → lbytes klen → ST (key i)
val compute: i:id → k:key i → x:lbytes len → ST (lbytes len)
 (ensures fun mem0 y mem1 →
   if safe i then r == lookup (ideal k) x mem1
   else r == Spec.Cipher.compute (real k)  x)
(* PRF: Implementation *)
let key i = if safe i then lbytes klen  else map (lbytes len) (lbytes len)
let compute i k x = if safe i then
   if lookup (ideal k) x = None then extend (ideal k) x (sample len);
   lookup (ideal k) x
 else Spec.Cipher.compute (real k) x
```

The interface declares keys as abstract, hiding both the real key value and the ideal log, and relies on the log to specify the effects of computing a block for safe instances. Non-ideal instances are specified using a pure functional specification of Spec.compute.

The idealized interface of AE works similarly (an ideal instance is a table that stores randomly sampled ciphertexts), and has been documented in previous work [11]. We give an overview in Appendix 8.3, where we also discuss the confidentiality model.

Proof of Theorem 1 The idealized interfaces of AE and PRF constitute our cryptographic assumptions, from which we implement an idealized interface that reflects the stream-security (NHSE^b) game (§3.3). As explained in §3.2, we introduce an additional assumption on the AE interface: we require the *sample* to uniquely identify all ciphertexts. This step is justified on paper by adding the

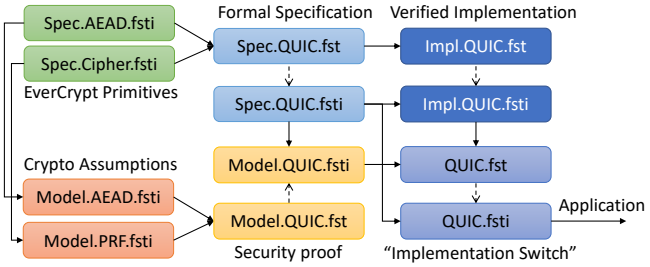


Figure 6: Integrated security model and verified implementation. In F⋆, an 'fsti' file is an interface for the corresponding 'fst' file, similar to a '.h' and '.c' files in C.

probability of sample collision to the advantage. We give below a much abstracted overview of the post-condition of decrypt, implemented using the ideal state of AE and PRF (Appendix 8.3 contains the real interface). We leave to the reader the exercise of mapping the post-condition to the decrypt oracle in NHSE[b].

```
val decrypt (k:id) (w:writer k) (r:reader w) (qp:quic_packet k) :
 ST (option (l:plainlen & plain (fst k) l))
 (ensures fun mem0 res mem1 → safe k ⇒
  match PRF.lookup (get_prf k w) (get_sample qp) mem0 with
  | None → None? res
  | Some (sample, mask) →
   let headers = header_decrypt r qp mask in
   let ln, npn = get_header_pn headers in
   let n = expand_pn ln (reader_max_pn r) npn in
   match AE.lookup_nonce (get_ae k w) n h0 with
  | None → None? res
  | Some (aad, ln', p', c') →
    if aad = headers && l' = ln && get_sample c' = sample then
     res = Some (|l',p'|) ∧ update_max r h0 n h1
   else None? res)
```

4.6 Verified Correctness and Security

As shown in Figure 6, our implementation of the idealized interface for the QUIC record layer (`Model.QUIC.fst`) uses our high-level specification (`Spec.QUIC.fsti`) in the real case (when b=0). However, we would like to extend the security guarantees to the low-level implementation (`Impl.QUIC.fsti`). This is accomplished with a technique we call an *implementation switch*, which is a perfect idealization step that replaces the call to the low-level implementation with a call to the security model. Note that the adversary for this step is much more powerful than in usual cryptographic games, because it can observe timing and memory access patterns in addition to the input/output behavior of the function. We assume that the execution of specification code is not observable, while the timing and memory access patterns of the low-level code are. EverCrypt guarantees by typing that the secret inputs are abstract, which we inherit in our record implementation. We implement the switch (`QUIC.fst`) by calling the low-level implementation on public inputs (to generate the correct observable side-effects), then replacing the value written to the output buffer by the output of calling the model on the high-level representation of the low-level inputs (i.e., arrays and machine integers are replaced by sequences and mathematical integers). Since both the model and low-level implementation share the same specification, and the observable side-effects are independent of secret inputs, to the adversary, the switch is indistinguishable to the adversary.

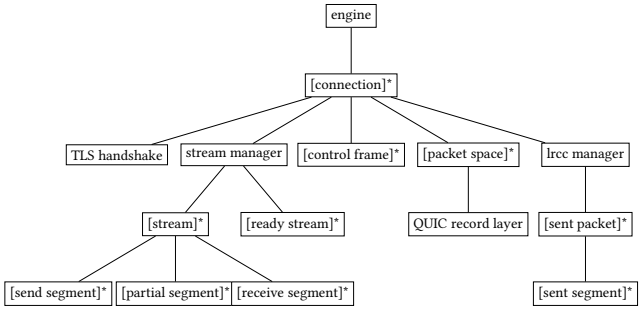


Figure 7: (Simplified) Hierarchy of Structures in our QUIC protocol logic, which utilizes the TLS handshake and QUIC record layer. `[X]*` represents a repetition of structure `X`, via a doubly-linked list, array, extendable vector, or sequence.

5 OUR QUIC REFERENCE IMPLEMENTATION

To evaluate if our verified record layer (§4) satisfies the needs of the QUIC protocol, we have developed a provably-safe reference implementation of QUIC on top of the record layer. We have also developed an example server and client that utilize our implementation to perform secure file transfers.

5.1 Implementation Overview

Our prototype consists of three modules – the TLS handshake, the QUIC record layer, and the QUIC protocol logic. The TLS handshake is based on prior verified TLS work [14] but has been updated to perform the TLS 1.3 handshake; it provides symmetric keys for bulk data transfer. These keys are used by our QUIC record layer (§4), which handles the encryption and decryption of QUIC traffic, as well as packet-level parsing/serialization. Using these two modules, the QUIC protocol logic implements the rest of the protocol, including for example, connection and stream management. This module is verified for safety, laying the groundwork for functional correctness in the future.

Our prototype's API is compliant with draft 24 of IETF QUIC. More specifically, it allows an application to interact with connections and streams as follows: open a connection as a client; listen for connection as a server; control and configure various resources such as number of permitted streams; and open/close a stream; write to/read from a stream.

5.2 The QUIC Protocol Logic Module

The QUIC protocol logic implements stream multiplexing, connection management, frame decoding/encoding, loss recovery, congestion control, and other functionality required by the IETF QUIC standard. Our implementation is centered around the various structures in Figure 7. The `engine` represents a prototype instance that manages `connection` instances (a client instance contains one `connection`; a server instance contains zero or more, depending on the number of connected clients). A `connection` contains multiple `stream` instances. It manages stream multiplexing, loss recovery, and the interactions with the TLS handshake and QUIC record

layer. Finally, the `stream` maintains the sending/receiving states of a stream. Each structure contains many other supporting structures.

Our QUIC protocol logic is verified for memory safety, type safety, termination, and the absence of integer overflows. This prevents, e.g., buffer overflows, type-safety flaws, use-after-free, and null-pointer accesses. We also prove correctness of some key data structures, e.g., a doubly-linked list and an expandable array.

Although the other modules in our reference implementation are written and verified in F*, we write and verify our QUIC protocol logic in Dafny [36], an imperative, objected-oriented verification language. While F*'s higher-order, ML-inspired design is convenient for reasoning about cryptographic properties, the QUIC protocol logic primarily manages stateful data structures in a classically imperative fashion, which better matches Dafny's design. Indeed, in an early phase of this project, it required several person-months to implement and verify a generic doubly-linked list library in F*, while it required only three hours to do so in Dafny. Dafny was better able to handle multiple heap updates and the complex invariants needed to prove and maintain correctness.

To support the compilation of our QUIC protocol logic, we have extended Dafny to add a C++ backend. C++ offers multiple benefits. First, it simplifies integration with the C code compiled from the F* code of the other two modules. Second, it enables performance optimizations that are harder to realize in Dafny's higher-level backends for C#, Java, JavaScript, or Go. Finally, C++ (as opposed to C) is a convenient compilation target for Dafny, since it includes a standard collections library, support for reference-counted smart pointers, and platform agnostic threading.

Our development includes ~500 lines of trusted Dafny code modeling the effects of calls to the other two modules; the pre- and post-conditions are carefully matched with their F* implementations. We model calls to the underlying OS (e.g., for UDP) similarly.

5.3 Proof Challenges and Techniques

We briefly summarize our overall proof strategy, challenges we encountered, and techniques for coping with those challenges.

To prove the safety of our QUIC protocol logic, we establish and maintain validity invariants throughout our codebase. Specifically, for each structure used in the code, we define a `valid` predicate, which ensures that structure's safety.

Structures lower in the hierarchy (Figure 7) mostly contain only primitive types. Hence, we define validity directly through type refinement. For example, to represent a frame of data in a stream, we use a `frame` datatype, which stores a byte array, a length, and an offset into the stream. A type refinement on the datatype ensures the array is not null, that the length is accurate, and that the length plus the offset will not cause an integer overflow.

Structures higher in the hierarchy contain nested substructures, which complicates our validity definitions. These now need to ensure: **(i)** validity of all substructures, **(ii)** the disjointness of substructures. For example, a `stream` manages the receive/send buffers through multiple doubly-linked lists, all of which must be valid. Further, these lists should not be aliased, which also means the nodes of the lists must be completely disjoint.

A standard technique used to handle such complex data structure reasoning is to maintain, in parallel to the actual data, a "ghost" representation (i.e., one used only for proof purposes and that will not appear in the compiled code). The ghost representation is typically a set of all substructures. This facilitates a succinct invariant about the disjointness of each member of the set, and it allows the parent data structure to define its ghost representation simply as the union of its children's ghost representations. Unfortunately, we found that data structures containing four or more nested structures (e.g., the connection object) quickly overwhelmed the Dafny verifier. The underlying challenge appears to be the complex set reasoning which arises from needing to repeatedly flatten sets-of-sets-of-objects into sets-of-objects.

Ultimately, we take advantage of Dafny's ability to do type-based separation, i.e., to define types that are known to be incomparable. Rather than homogenizing the distinct sub-structures into a single object-based representation, we maintain ghost representations of each distinct type. This requires additional proof annotations, but it makes the verifier's reasoning about validity much simpler, since non-aliasing of instances of different class is "free".

Even with the aforementioned discipline, any mutation of subcomponents (however deep) requires reproving the validity of all layers above it. Hence we carefully structure our code in multiple layers: the innermost performs the actual mutation, and the outer layers simply expose these changes at higher and higher levels.

A final technique we employ is the careful use of immutability. Immutable data structures simplify proof reasoning, since any immutable value is independent of the state of the heap, and thus will remain valid regardless of how the heap changes. On the other hand, used indiscriminately, immutability imposes a performance cost due to excessive data copies.

To balance these concerns, we typically use immutable structures at lower levels and mutable types elsewhere. This simplifies reasoning at the upper levels (which are already quite complex) without unduly hurting performance, since the upper levels can manipulate, say, linked lists of immutable lower-level structures, avoiding unnecessary copies. Even at the lower levels, we sometimes find it convenient to keep a structure in a mutable form while constructing it (e.g., while reading from a stream), and then "freeze" it in an immutable form to simplify reasoning. Since these data structures are not subsequently mutated, we lose little performance.

6 EVALUATION

We first evaluate the effort required to build and verify our QUIC reference implementation. Next, we measure the performance of our record layer, the main focus of our work. Finally, while the main goal of our verified-safe protocol logic is to demonstrate that our record layer suffices to implement QUIC, we also, as a point of comparison, evaluate the overall performance of our QUIC prototype. We perform all of these measurements on a Linux desktop with an Intel i9-9900k processor with 128GB memory. When measuring on the network, we connect to a Linux desktop with the same configuration over a 1 Gigabit Ethernet LAN.

6.1 Verification Effort

Table 8 summarizes the size and verification time for our verified components. Overall, our record layer consists of about 10K lines

Modules	LoC	Verif.	C/C++ LoC
Verified Record Layer (§4)			
QUIC.Spec.*	2,570	5m12s	-
QUIC.Impl.*	2,011	6m32s	-
QUIC.Model.*	1,317	1m12s	-
LowParse.Bitfields.*	1,770	1m29s	-
LowParse.Bitsum.*	2,168	2m05s	-
Total	9,836	16m30s	-
QUIC Reference Implementation (§5)			
Connection mgmt	4,653	14m12s	-
Data Structures	651	9s	-
Frame mgmt	1,990	1m50s	-
LR & CC	758	11s	-
Stream mgmt	1,495	3m25s	-
Misc	118	2s	-
FFI	558	9s	1461
Server & Client	-	-	648
Total	10,223	19m46s	2,109

Figure 8: Summary of our verified codebase

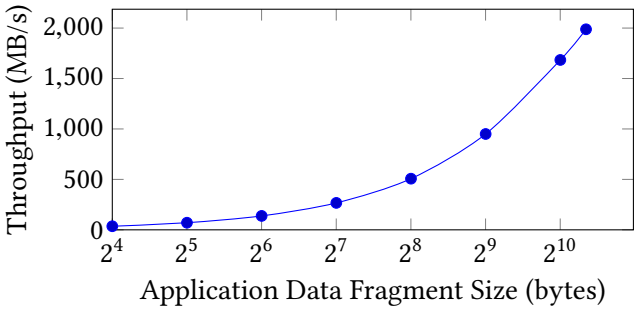


Figure 9: Record-layer performance: throughput of encryption/decryption of packets with various fragment sizes.

of F⋆ code, which extract to ~6K lines of C. A significant portion of this total consists of extensions to the EverParse libraries to support the bit-level combinators that describe header formats. Our verifiably-safe implementation of the QUIC protocol logic consists of 10K lines of Dafny code, which compiles to ~13K lines of C++. Additionally, we have ~800 lines of trusted C++ code to connect the record layer and the protocol logic, and another ~700 lines of trusted C++ code to connect the protocol logic to the underlying OS platform (shown collectively as FFI in the table).

Overall, we estimate that about 20 person-months went into this effort, which includes the overhead of training multiple new team members on our tools, methods, and QUIC.

6.2 QUIC Record Layer

Our core contribution, the QUIC record layer, performs full stateful encryption/decryption of packets, including header processing and protection. To evaluate its performance, we measure the application data throughput for varying packet-content sizes.

Figure 9 shows our results. At the typical MTU (1300 bytes), our implementation supports 1.98 GB/s of QUIC application data, which is ~2.4 times slower than raw AEAD.

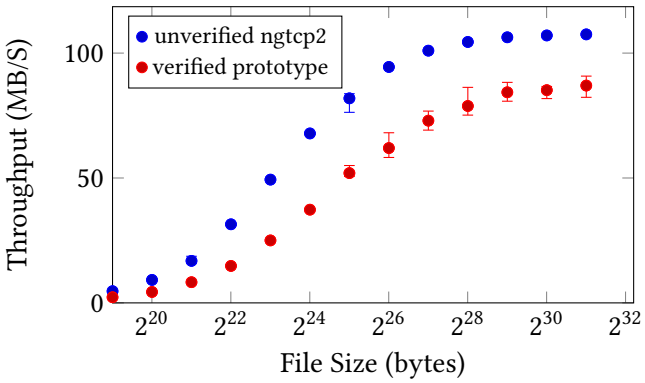


Figure 10: QUIC prototype performance: comparison of file transfer throughput with ngtcp2 on a 1 Gbps LAN.

6.3 File Transfer Performance

To evaluate the overall performance of our QUIC reference implementation, we use it to transfer files over the network, ranging in size from 512KB to 2GB, and measure throughput, comparing against an unverified baseline There are many unverified implementations of QUIC in various languages; we picked `ngtcp2`[50] as our baseline because it is a popular and fast implementation written in C++. Using a C++ baseline avoids performance differences due to differing runtimes. Figure 10 shows our results. Unsurprisingly, our implementation is slower than the carefully optimized `ngtcp2`. On smaller file sizes, our prototype is about twice as slow, but on large ones, it only lags by ~21%. Profiling shows this is largely due to our naïve coarse-grained locking strategy, which we plan to refine.

7 RELATED WORK

Some papers attempt to model the security of Google QUIC [27], but the results available for IETF QUIC are more limited [19]. QPE is one of many extensions of nonce-based authenticated encryption with additional data [45]. The use of AEAD to build stateful encryption [16, 46], stream-based channels [28], and concrete applications to protocols such as the TLS record layer [11, 17] or SSH [7] have been extensively studied. However, an important goal of the current QUIC packet encryption construction is nonce confidentiality, which is achieved by keeping some of the nonce implicit (an idea that appeared in the CAESAR competition, and received a proposed security definition [38]) and encrypting the explicit part, for which several related constructions have been proposed with security proofs [8]. Our work combines these results with the modular type-based verification method for cryptographic proofs of Fournet et al. [30] to create an efficient verified implementation, building on the verified EverCrypt [41] crypto library. An important limitation of the methodology is that only perfect indistinguishability steps can be mechanically verified. Other tools, notably EasyCrypt [4], have relational semantics that can reason about advantages in game hops, and have been used to fully prove the security of complex constructions such as RSA-PSS [1]. However, writing fast implementations is more difficult in EasyCrypt. The preferred approach for implementation security has been to use general-purpose C verification tools and prove the security of an extracted model [25], in contrast to our implementation switching strategy.

We stress that the scope of our security analysis is limited to the QUIC record layer, which is insufficient to conclude that QUIC is a secure-channel protocol. In contrast, considerable work has gone towards proving that TLS 1.3 provides a secure channel. For example, Dowling et al. [24] present a detailed cryptographic model of the handshake; Bhargavan et al. [10] present a computational model verified in CryptoVerif; and Cremers et al. [22] present a symbolic model verified in Tamarin. These are recent instances of the broader field of tool-assisted security proofs for cryptographic protocols and their implementations [3, 5, 6, 20, 26, 31, 32]. Readers can refer to the surveys of Barbosa [2], Blanchet [15] and Cortier [21].

8 CONCLUSIONS

This paper is the first step towards a provably secure and safe implementation of the IETF standard QUIC protocol. Despite some weaknesses, we have proved the security of QUIC packet encryption construction and built the first high-performance, low-level implementation (with proofs of correctness, runtime safety, and security). We have also built a safe implementation of the QUIC transport on top of our verified packet encryption component and the verified miTLS handshake. Our next steps are to write a functional specification of the transport and verify the correctness of our implementation, integrate the TLS handshake security model with the record layer, and expose an idealized interface to the QUIC transport that captures application data stream security [29].

ACKNOWLEDGEMENTS

Work at CMU was supported in part by grants from a Google Faculty Fellowship, the Alfred P. Sloan Foundation, the Department of the Navy, Office of Naval Research under Grant No. N00014-17-S-B001, and the National Science Foundation and VMware under Grant No. CNS-1700521. We thank Felix Günther, Markulf Kohlweiss and anonymous reviewers for their feedback.

REFERENCES

[1] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: Efficient provably secure machine code from high-level implementations," in *Computer & Communications Security*, ser. CCS '13. ACM, 2013, p. 1217–1230.

[2] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," 2020.

[3] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, and F.-X. Standaert, "Maskverif: Automated verification of higher-order masking in presence of physical defaults," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2019, pp. 300–318.

[4] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, *EasyCrypt: A Tutorial*. Springer, 2014, pp. 146–166.

[5] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"," in *Computer Security Foundations (CSF)*. IEEE, 2018, pp. 328–343.

[6] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 293–310.

[7] 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, p. 206–241, May 2004. [Online]. Available: https://doi.org/10.1145/996943.996945

[8] M. Bellare, R. Ng, and B. Tackmann, "Nonces are noticed: AEAD revisited," in *Advances in Cryptology – CRYPTO 2019*, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 235–265.

[9] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology – EUROCRYPT 2006*, 2006, pp. 409–426.

[10] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 483–502.

[11] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J.-K. Zinzindohoué, "Implementing and proving the TLS 1.3 record layer," in *IEEE Security & Privacy*. IEEE, 2017.

[12] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, , A. Pironti, and P.-Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 98–113.

[13] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Cryptographically verified implementations for TLS," in *ACM Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 459–468.

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

[15] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *Principles of Security and Trust (POST)*. Springer, 2012, pp. 3–29.

[16] A. Boldyreva, J. P. Degabriele, K. G. Paterson, and M. Stam, "Security of symmetric encryption in the presence of ciphertext fragmentation," in *EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 682–699.

[17] C. Boyd, B. Hale, S. F. Mjølsnes, and D. Stebila, "From stateless to stateful: Generic authentication and authenticated encryption constructions with application to tls," in *CT-RSA 2016*, K. Sako, Ed. Cham: Springer International Publishing, 2016, pp. 55–71.

[18] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss, "State separation for code-based game-playing proofs," in *ASIACRYPT 2018*, ser. Lecture Notes in Computer Science, vol. 11274. Springer, 2018, pp. 222–249.

[19] S. Chen, S. Jero, M. Jagielski, A. Boldyreva, and C. Nita-Rotaru, "Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC," in *ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Springer International Publishing, 2019, pp. 404–426.

[20] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran *et al.*, "Continuous formal verification of Amazon s2n," in *Computer Aided Verification (CAV)*. Springer, 2018, pp. 430–446.

[21] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *Journal of Automated Reasoning*, vol. 46, no. 3-4, pp. 225–259, 2011.

[22] C. Cremers, M. Horvat, S. Scott, and T. v. d. Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *IEEE Security and Privacy*, May 2016, pp. 470–485.

[23] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," 2008.

[24] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM, 2015, pp. 1197–1210.

[25] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," *Journal of Computer Security*, vol. 22, no. 5, pp. 823–866, 2014.

[26] H. Eldib, C. Wang, and P. Schaumont, "SMT-based verification of software countermeasures against side-channel attacks," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2014, pp. 62–77.

[27] M. Fischlin and F. Günther, "Multi-stage key exchange and the case of google's QUIC protocol," in *ACM CCS*. ACM, 2014, pp. 1193–1204.

[28] M. Fischlin, F. Günther, G. A. Marson, and K. G. Paterson, "Data is a stream: Security of stream-based channels," in *CRYPTO 2015*, R. Gennaro and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 545–564.

[29] M. Fischlin, F. Günther, G. A. Marson, and K. G. Paterson, "Data is a stream: Security of stream-based channels," in *Advances in Cryptology - CRYPTO 2015*, 2015, pp. 545–564.

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

[31] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "IODINE: Verifying constant-time execution of hardware," in *Usenix Security*, 2019, pp. 1411–1428.

[32] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *Usenix Operating Systems Design and Implementation (OSDI)*, 2014, pp. 165–181.

[33] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," IETF draft, 2019.

[34] T. Jager, J. Schwenk, and J. Somorovsky, "On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption," in *22nd ACM Conference on Computer and Communications Security*, 2015, pp. 1185–1196.

[35] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, and J. Iyengar, "The QUIC transport protocol: Design and internet-scale deployment," in *SIG on Data Communication*. ACM, 2017, pp. 183–196.

[36] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the Conference on Logic for Programming, Artificial Intelligence,*

and Reasoning (LPAR), 2010.
[37] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is QUIC? provable security and performance analyses," in *IEEE Security and Privacy*. IEEE, 2015, pp. 214–231.
[38] C. Namprempre, P. Rogaway, and T. Shrimpton, "AE5 security notions: Definitions implicit in the CAESAR call," Cryptology ePrint Archive, Report 2013/242, 2013, https://eprint.iacr.org/2013/242.
[39] K. Oku, "Client's initial destination CID is unauthenticated," QUIC WG issue tracker, 2019. [Online]. Available: https://github.com/quicwg/base-drafts/issues/1486
[40] A. Prado, N. Harris, and Y. Gluck, "SSL, gone in 30 seconds: a BREACH beyond CRIME," *Black Hat USA*, vol. 2013, 2013.
[41] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet *et al.*, "Evercrypt: A fast, verified, cross-platform cryptographic provider," Cryptology ePrint Archive, Report 2019/757. https://eprint. iacr. org/2019/757, Tech. Rep., 2019.
[42] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *PACMPL*, vol. 1, no. ICFP, pp. 17:1–17:29, Sep. 2017.
[43] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "EverParse: verified secure zero-copy parsers for authenticated message formats," in *28th USENIX Security Symposium*, 2019, pp. 1465–1482.
[44] J. Rizzo and T. Duong, "The CRIME Attack," September 2012.
[45] P. Rogaway, "Authenticated-encryption with associated-data," in *CCS'02*. ACM, 2002, pp. 98–107.
[46] P. Rogaway and Y. Zhang, "Simplifying game-based definitions: Indistinguishability up to correctness and its application to stateful AE," in *CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds., 2018, pp. 3–32.
[47] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43nd ACM Symposium on Principles of Programming Languages, POPL 2016*, 2016, pp. 256–270.
[48] M. Thomson, "Version-independent properties of QUIC," IETF draft, 2019.
[49] M. Thomson and S. Turner, "Using TLS to secure QUIC," IETF draft, 2019.
[50] T. Tsujikawa, "ngtcp2 project is an effort to implement IETF QUIC protocol," GitHub, 2019. [Online]. Available: https://github.com/ngtcp2/ngtcp2
[51] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *ACM Conference on Computer and Communications Security*. ACM, 2017, pp. 1789–1806.

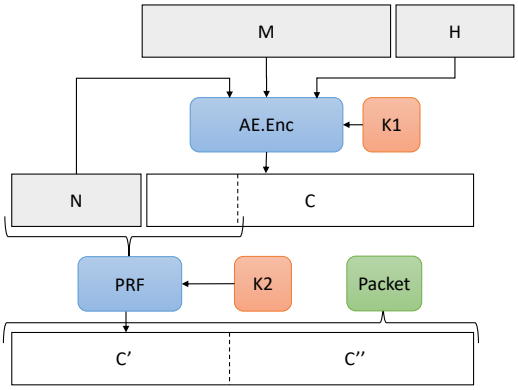


Figure 11: The HN2[AE, PNE] construction

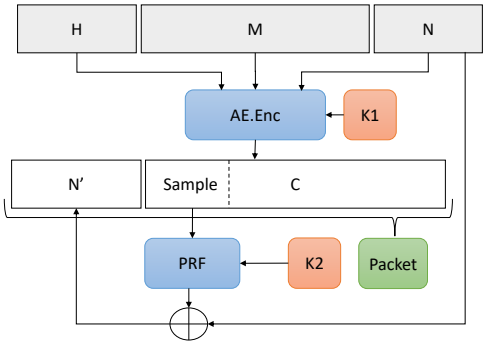


Figure 12: The HN1[AE,PNE] construction

APPENDIX

8.1 Details of QPE[AE,PNE] construction

Keygen()
$N_s \leftarrow\$ \{0,1\}^{\mathsf{AE.nlen}}$
$K_1 \leftarrow\$ \mathsf{AE.keygen}()$
$K_2 \leftarrow\$ \mathsf{PNE.keygen}()$
return (N_s, K_1, K_1)

Encode(N, L_N)
return $N \& (2^{8L_N} - 1)$

Dec(K, N_h, P, ℓ_H)
$N_s, K_1, K_2 \leftarrow K$
$H', C' \leftarrow \mathsf{split}(P, 1 + \ell_H)$
$S \leftarrow C'[4..20]$
$B \leftarrow \mathsf{PNE.compute}(K_2, S)$
$F \leftarrow H'[0] \oplus (B[0] \& 15)$
$L_N \leftarrow 1 + F \& 3$
$C'', C \leftarrow \mathsf{split}(C', L_N)$
$N' \leftarrow C'' \oplus B[1..1 + L_N]$
$N \leftarrow \mathsf{Decode}(N_h, N', L_N) \oplus N_s$
$H \leftarrow F \,\|\, H'[1..\ell_H] \,\|\, N'$
$M \leftarrow \mathsf{AE.dec}(K_1, N, H, C)$
return M

Decode(N', N_h, L_N)
$W \leftarrow 2^{8L_N}$; $X \leftarrow N_h + 1$
$N \leftarrow N' + (X \& (W - 1))$
if $N \le X - W/2$
  $\wedge\, N < 2^{62} - W$
 return $N + W$
if $N > X + W/2 \wedge N \ge W$
 return $N - W$
return N

Enc(K, N, H, M)
$N_s, K_1, K_2 \leftarrow K$
$L_N \leftarrow 1 + H[0] \& 3$
$N' \leftarrow \mathsf{Encode}(N, L_N)$
$N_0 \leftarrow N \oplus N_s$
$C \leftarrow \mathsf{AE.enc}(K_1, N_s, H \,\|\, N', M)$
$S \leftarrow C[(4 - L_N)..(20 - L_N)]$
$B \leftarrow \mathsf{PNE.compute}(K_2, S)$
$H' \leftarrow H \,\|\, (N' \oplus B[1..1 + L_N])$
$H'[0] \leftarrow H[0] \oplus (B[0] \& 15)$
return $H' \,\|\, C$

8.2 Nonce-hiding encryption

Figure 12 and 11 shows the HN1 and HN2 constructions of Bellare et al. [8], which are proved secure with respect to AE2 below, assuming that AE is AE1-secure and PNE is PRF-secure:

Game $\mathsf{AE2}^b(\mathsf{SE2})$
$T \leftarrow \emptyset$; $k \xleftarrow{\$} \mathsf{SE2.gen}()$

Oracle Decrypt(C, H)
if $b = 1$ **then**
 $M \leftarrow T[\_, C, H]$
else
 $M \leftarrow \mathsf{SE2.dec}(k, C, H)$
return M

Oracle Encrypt(N, M, H)
assert $T[N, \_, \_] = \perp$
if $b = 1$ **then**
 $C \xleftarrow{\$} \{0,1\}^{|N|+|M|+SE2.\ell_T}$
 $T[N, C, H] \leftarrow M$
else
 $C \leftarrow \mathsf{SE2.enc}(k, N, M, H)$
return C

8.3 Idealized interfaces for QPE

AEAD Assumption We give a simplified abstract version of the idealized AEAD interface, which implements the $\mathsf{AE1}^b$ game:

```
abstract type key (i:id)
val ideal: #i:id{safe i} → key i →
  map (nonce × cipher × header) (plain i)
val real: #i:id{¬ (safe i)} → key i → lbytes klen

val keygen: i:id{fresh i} → ST (key i)
  (ensures fun mem0 k h1 → safe i ⇒ ideal k mem1 = ∅)
val encrypt: #i:id → k:key i →
  n:nonce → h:header → p:plain i → ST cipher
  (ensures fun mem0 c mem1 →
    if safe i then ideal k mem1 == extend (ideal k mem0) (n,c,h) p
    else c == Spec.AEAD.encrypt (real k) n h p)
val decrypt: #i:id → k:key →
  n:nonce → h:header → c:cipher → ST (option plain)
  (ensures fun mem0 r mem1 →
    if safe i then r == lookup (ideal k mem0) (n,c,h)
    else r == Spec.AEAD.decrypt (real k) n h c)
```

The post-condition of decryption clearly guarantees plaintext integrity. To model confidentiality, the idea is to rely on type parametricity of idealized plaintexts. The intuition is to prove that the ideal implementation of encryption and decryption never access the actual representation of the plaintext, by making the type of plaintext abstract (conditionally on safety). This guarantees that ideal plaintexts are perfectly (i.e. information theoretically) secure. Therefore, AE^b is also parametrized by a module Plain^b that defines abstract plaintexts plain i, with an interface that allows access to their concrete byte representation only when ¬ (safe i) (for real encryption). In practice, different applications want to use different

plaintext types (sometimes, type separation is desirable within the same application, for instance to separate encrypted session tickets from encrypted retry cookies in TLS). Hence, our actual idealized AEAD interface is also parametric in the type of (conditionally abstract) plaintexts, which must be set when an instance is created.

Implementation of AE5 reduction To prove the reduction of AE5-security of QPE to AE1-security and PRF-security, we need to implement the ideal functionality of AE5 using the ideal state of PRF and AE. In practice, our code assumptions are significantly more complex than the abstract overview of PRF and AE, to capture agility, plaintext type parametricity, ideal state invariants, and complex length conditions. We give below the real interface of our verified implementation of QPE encryption and decryption. The full source code of the verified implementation is available online.

```
val encrypt
 (#k:id)
 (w:stream_writer k)
 (#hl:headerlen)
 (hd:quic_header k hl)
 (nl:pnlen {hl + nl ≤ v AEAD.aadmax})
 (#l:plainlen {hl + nl + l + v AEAD.taglen ≤ pow2 32 − 1 ∧
  nl + l + v AEAD.taglen ≥ samplelen + 4})
 (p:plain (fst k) l):
 ST (quic_packet k hl (nl + l))
 (requires fun h0 →
  wincrementable w h0 ∧
  invariant w h0)
 (ensures fun h0 (ph,nec) h1 →
  let (i,j) = k in
  let aw = writer_aead_state w in
  let ps = writer_pne_state w in
  invariant w h1 ∧
  wctrT w h1 == wctrT w h0 + 1 ∧
  (safe k ⇒ (
   let (ne,c) = split #k #(nl+l) nec nl in
   let rpn = rpn_of_nat (wctrT w h0) in
   let npn = npn_encode j rpn nl in
   let alg = ((AEAD.wgetinfo aw).AEAD.alg) in
   let nce = create_nonce #k #alg (writer_iv w) rpn in
   let ad = Bytes.append (bytes_of_quic_header hd) npn in
   let s : PNE.sample = sample_quic_protect nec in
   let nn = pne_plain_of_header_pn hd npn in
   let cc = pne_cipher_of_pheader_epn ph ne in
   AEAD.wlog aw h1 ==
    Seq.snoc
     (AEAD.wlog aw h0)
     (AEAD.Entry #i #(AEAD.wgetinfo aw) nce ad #l p c) ∧
      PNE.table ps h1 ==
        Seq.snoc
          (PNE.table ps h0)
          (PNE.Entry #j #pne_plain_pkg s #(nl+1) nn cc))) ∧
  modifies (loc_union (footprint w) (loc_ae_region ())) h0 h1)

val decrypt
 (#k:id)
 (#w:stream_writer k)
 (r:stream_reader w)
 (#hl:headerlen {hl+5 ≤ v AEAD.aadmax})
 (#nll:pnplainlen{hl + nll + v AEAD.taglen ≤ pow2 32 − 1})
 (qp:quic_packet k hl nll) :
 ST (option (l:plainlen & plain (fst k) l))
 (requires fun h0 → rinvariant r h0 ∧ invariant w h0)
 (ensures fun h0 res h1 →
  let (i,j) = k in
  let ar = reader_aead_state r in
  let aw = writer_aead_state w in
  let ps = reader_pne_state r in
  rinvariant r h1 ∧
  modifies (rfootprint r) h0 h1 ∧
  (None? res ⇒ expected_pnT r h1 == expected_pnT r h0) ∧
  (safe k ⇒
   (let (ph,nec) = qp in
   let s = sample_quic_protect nec in
   let cp = pne_cipherpad_of_pheader_quicprotect ph nec in
   match PNE.entry_for_sample_cipherpad s cp ps h0 with
    | None → None? res
    | Some (PNE.Entry _ #ll n cc) →
     let nl:pnlen = ll − 1 in
     let (hd,npn) = header_pn_of_pne_plain #k #hl ph #ll n in
     let rpn = npn_decode #j #nl npn (expected_pnT r h0) in
     let alg = ((AEAD.rgetinfo ar).AEAD.alg) in
     let n = create_nonce #k #alg (reader_iv r) rpn in
     let ad = Bytes.append (bytes_of_quic_header hd) npn in
     match AEAD.wentry_for_nonce aw n h0 with
      | None → None? res
      | Some (AEAD.Entry _ ad' #l' p' c') →
       if ad' = ad && l' = nll − nl && snd (split nec nl) = c' then
        (res = Some (|l',p'|) ∧
          update_pnT r h0 rpn_of_nat (U64.v rpn + 1) h1)
       else None? res)))
```