A Security Model & Verified Implementation of the QUIC Record Layer

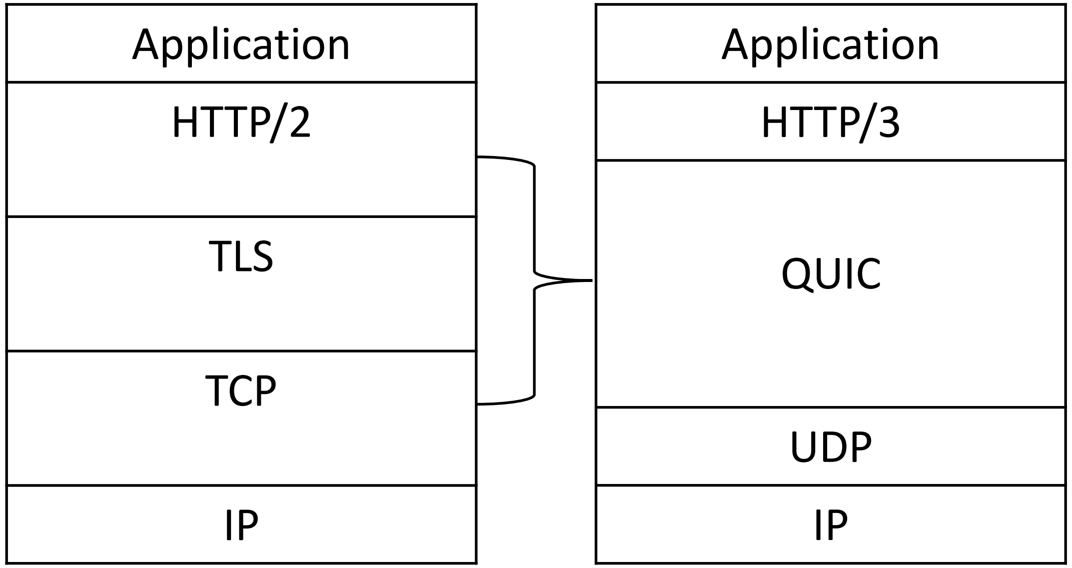
Antoine Delignat-Lavaud, Cédric Fournet (MSR-C),

Joseph Lallemand, Itsaka Rakotonirina (LORIA),

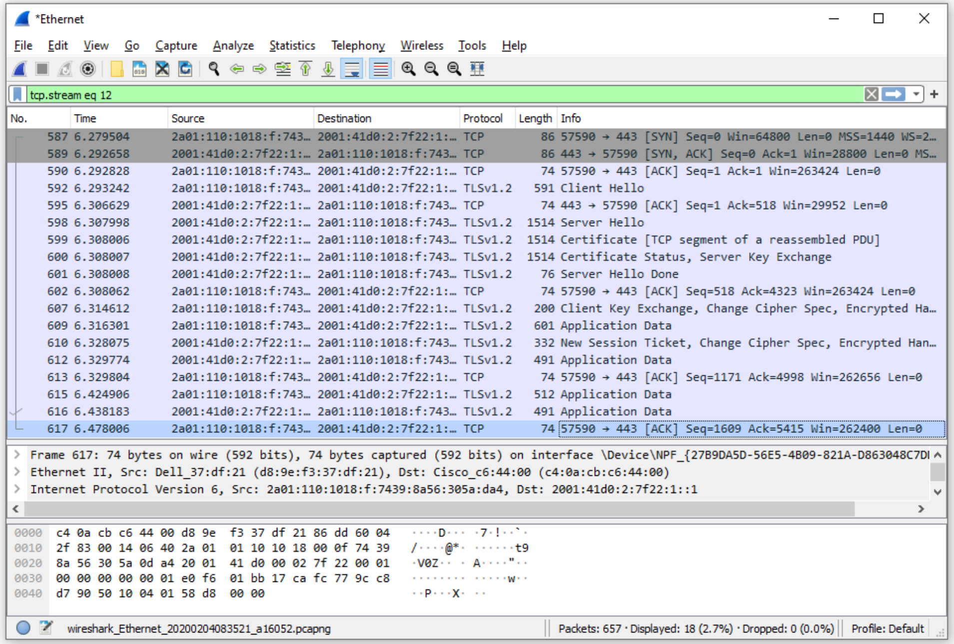
Jonathan Protzenko, Tahina Ramananandro (MSR-R)

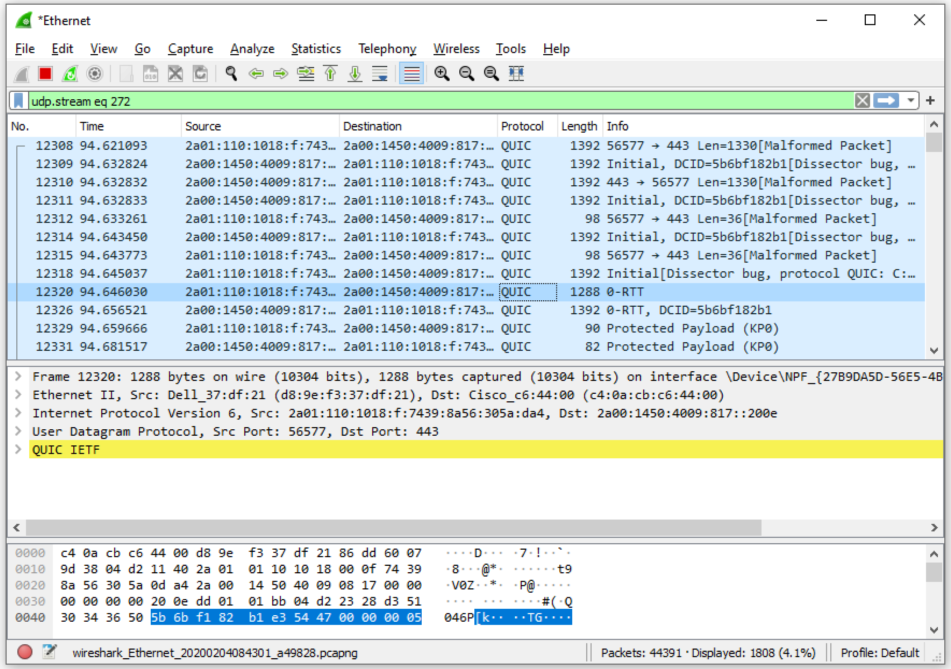
Bryan Parno, Jay Bosamiya Yi Zhou (CMU)

What is QUIC?



QUIC is a redesign of the Internet's networking protocol stack that trades off modularity for performance

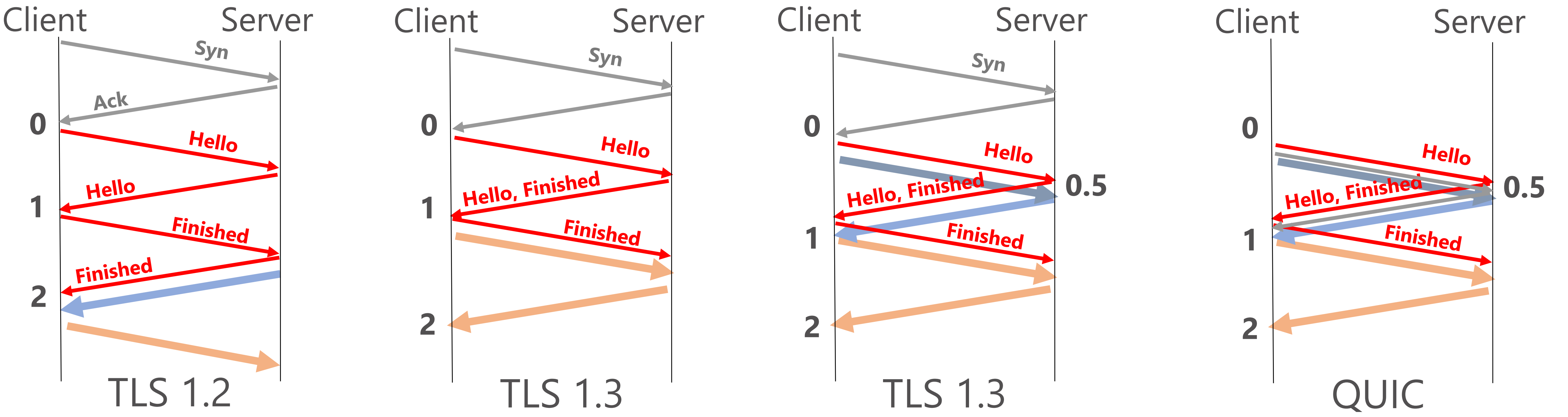




Connection Establishment Latency

Current HTTPS stack over TCP

New networking stack: UDP, QUIC, HTTP/3



“Two” roundtrips before sending application data

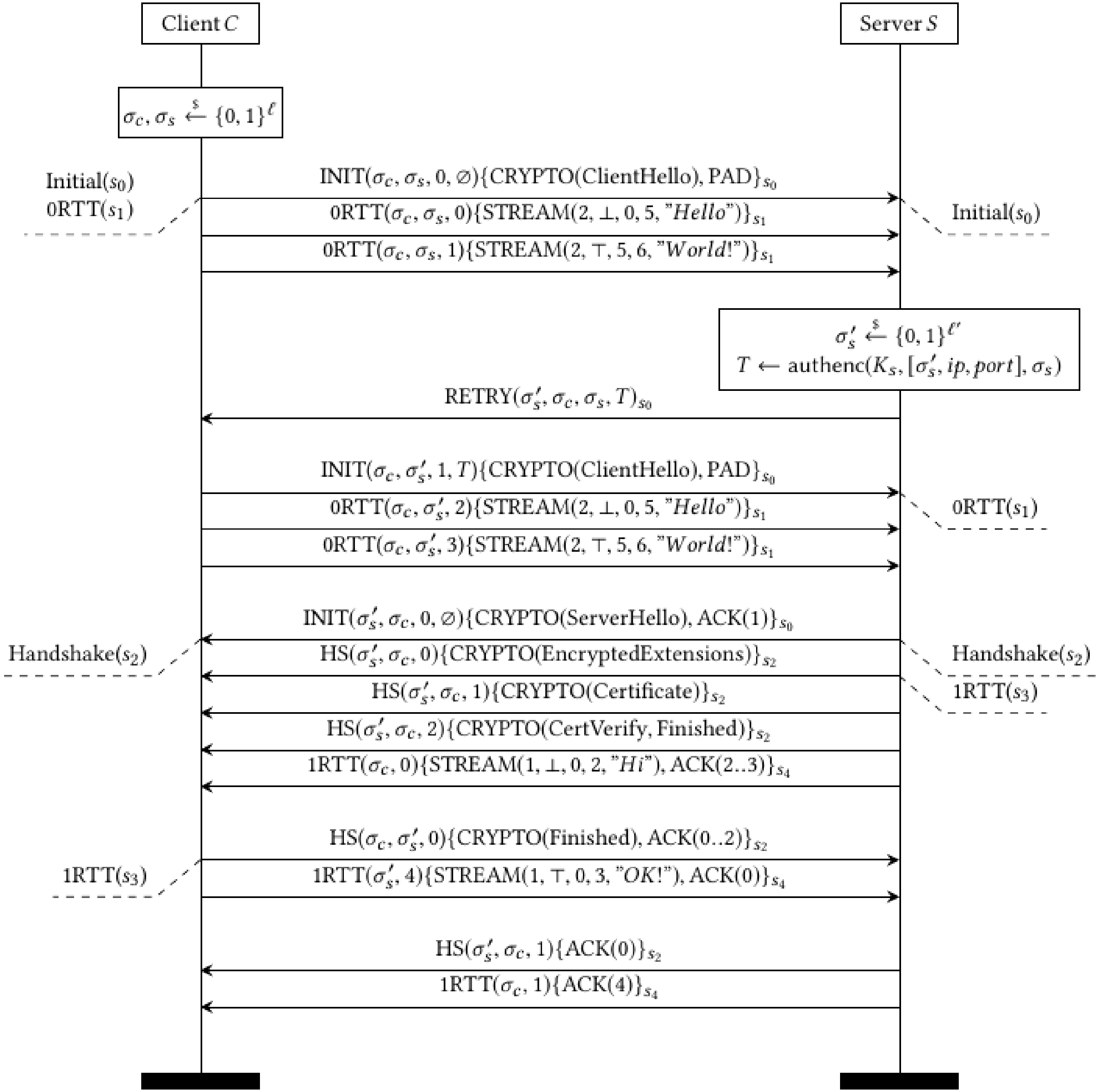
“One” roundtrip before sending application data

“Zero” roundtrip before sending application data

Cuts an extra roundtrip, enables more multiplexing

How QUIC works

- UDP datagrams contain one or more QUIC packets
- Packet types: INITIAL, 0RTT, RETRY, HS, 1RTT
- Packets contain list of frames
- CRYPTO frames contain TLS handshake messages
- STREAM frames carry app data, ACK acknowledgements



QUIC Packet Format

Version Negotiation

Connection multiplexing over same IP/port

```
+-+-+-+-+-+-+-+-+
|1|1| T |R R|P P|   T: Type    R: Reserved    P: PN length
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Version (32)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| DCID Len (8)  | Dest. connection ID length
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Destination Connection ID (0..160)              ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| SCID Len (8)  | Source connection ID length
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Source Connection ID (0..160)                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Payload Length (varint)                  ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Packet Number (8/16/24/32)                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Payload                            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Long header packets (INIT, 0RTT, HS, RETRY)

```
+-+-+-+-+-+-+-+-+
|0|1|S|R|R|K|P P|  S: Spin bit  K: Key phase bit
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Destination Connection ID (0..144)              ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Packet Number (8/16/24/32)                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Protected Payload                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Short header packets (1-RTT)

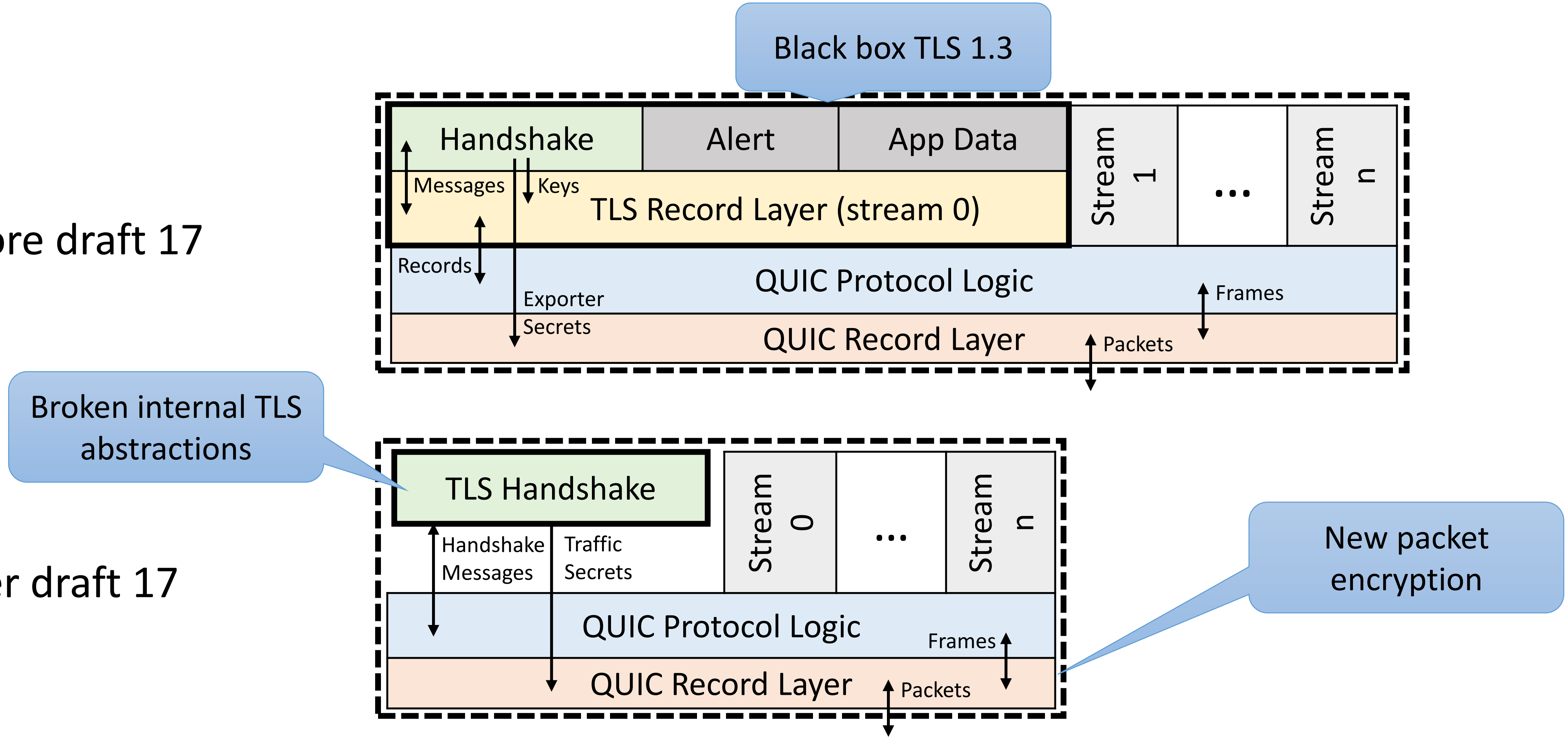
Implicit length

Variable length, truncated packet numbers

Internal QUIC Modularity

Before draft 17

After draft 17



Some Open Security Problems in QUIC

- **New custom construction for encrypting packets** (replaces TLS record layer)
- Status of mixed 0-RTT/1-RTT data streams (no end_early_data)
- QUIC re-keying (replaces TLS re-keying)
- QUIC retry (duplicates TLS hello retry)
- Version negotiation (duplicates TLS version negotiation)
- CID negotiation (somewhat duplicates TLS nonces)
- Exporting handshake & 1-RTT traffic secrets

QUIC Record Layer: Verification Goals

Functional Correctness

- Write complete formal specification of QUIC packet encryption
- Prove non-malleability of packet formatting
- Prove correctness of decryption

Cryptographic Security

- Create ideal model of packet stream encryption functionality
- Prove type-based reduction to core crypto assumptions

Verified Implementation

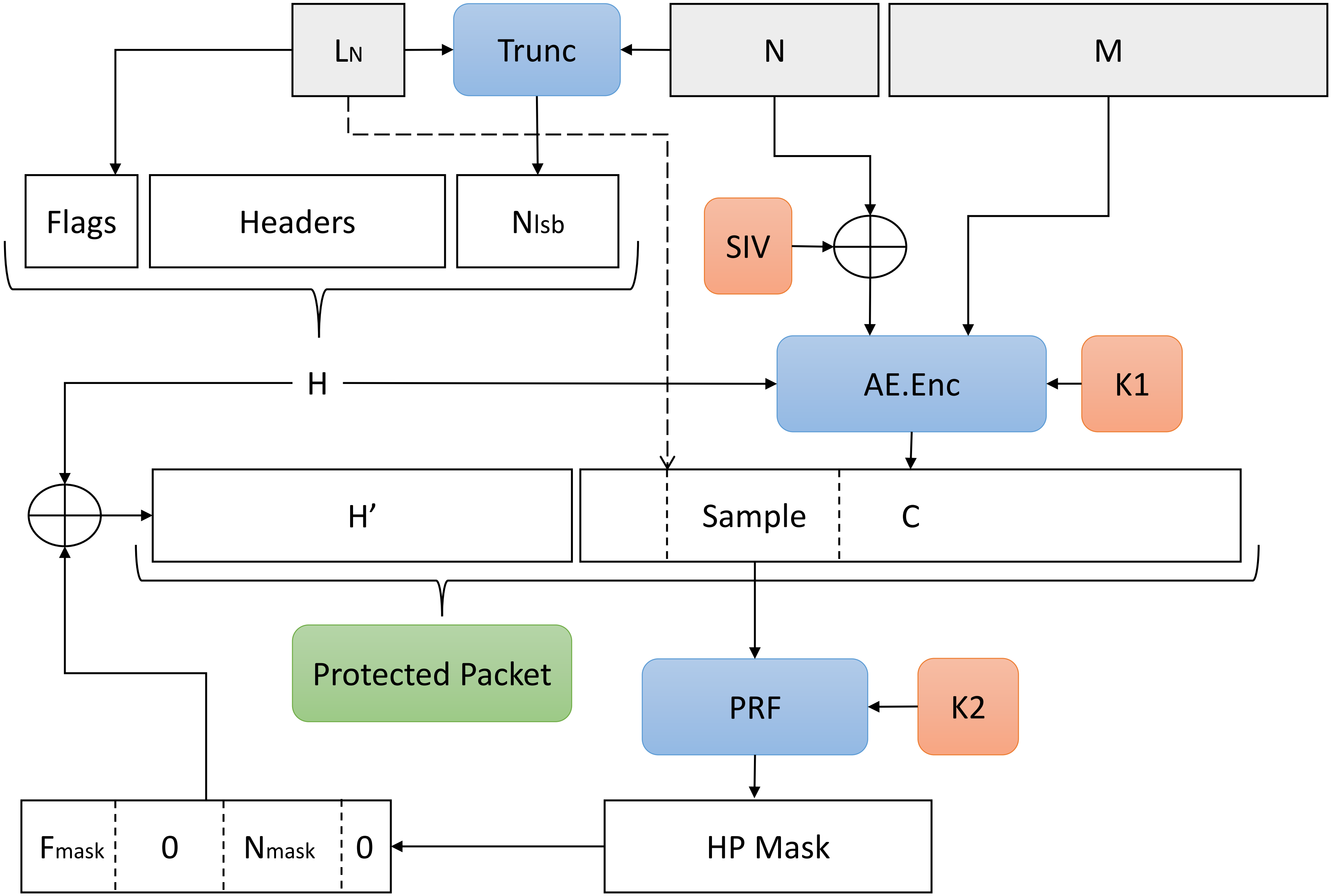
- Write implementation that extracts to high-performance C code
- Prove implementation is runtime safe
- Prove it is correct wrt the formal specification

Overview of QUIC Record Layer

- Payload (list of frames) is AEAD encrypted
- Header used as associated data
- What to do with the nonce?
  - No implicit sequence number (packets arrive out of order)
  - Explicit nonces are too long (12 bytes)
  - Sequential nonces correlate packets to connections
- Idea: use stateful encryption, send only least significant bits with packet and reconstruct most significant bits from state
  - Still, truncated packet number correlate packets to sessions
  - So, QUIC encrypts them without any space overhead

QUIC Packet Encryption

- The main idea is to use part of the payload ciphertext as an input to a PRF that generates a keystream block
- The construction is notably complicated by the fact that the truncated packet number is variable sized
- To model the security, we assume that the only goal is to protect the truncated packet number and its length



Modelling Single Packet Encryption

Standard AE security definition

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

$T \leftarrow \varnothing;\ k \xleftarrow{\$} \mathrm{SE1.gen}()$

Oracle $\mathrm{Decrypt}(N, C, H)$

if $b = 1$ **then**

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

else

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

return M

Oracle $\mathrm{Encrypt}(N, M, H)$

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

if $b = 1$ **then**

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

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

else

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

return C

Idealized Version

Real Version

Modelling Single Packet Encryption

Nonce-hiding AE definition

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

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

Oracle $\mathrm{Decrypt}(N_m, C, H)$

if $b = 1$ **then**

$\quad M \leftarrow T[N, C, H]$ **for** N

$\qquad\qquad$ s.t. $\mathrm{msb}(N) = N_m$

else

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

return M

Oracle $\mathrm{Encrypt}(N, L_N, M, H)$

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

if $b = 1$ **then**

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

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

else

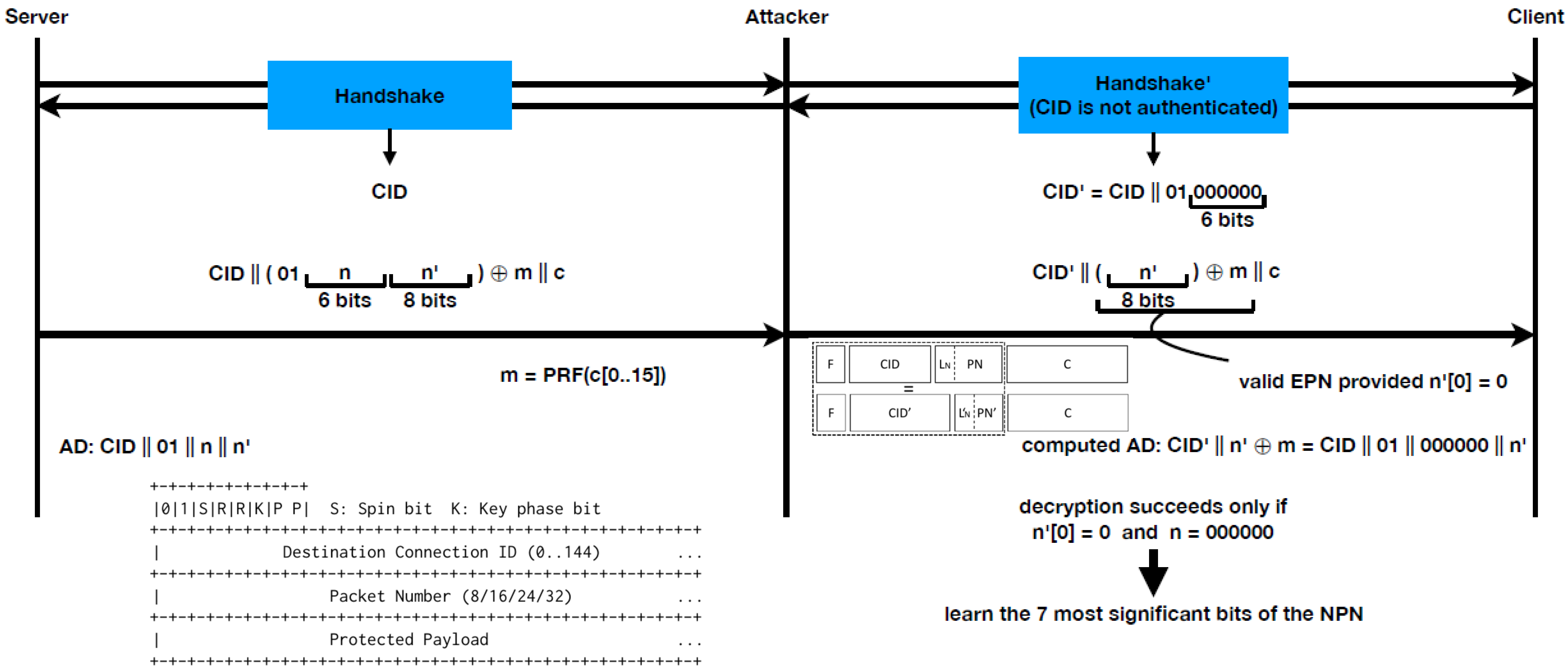
$\quad C \leftarrow \mathsf{E}.\mathrm{enc}(k, N, L_N, M, H)$

return C

Ln extra bytes of ciphertext to represent encrypted truncated nonce

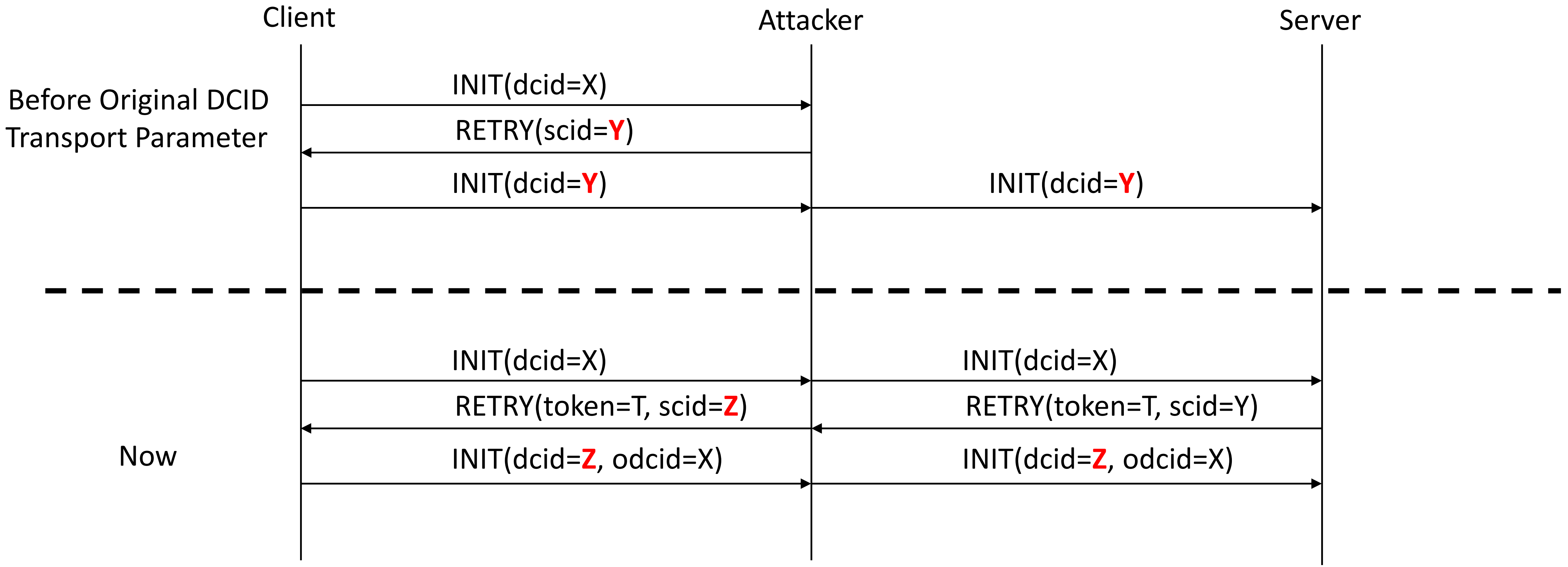
We store the full nonce in the ideal encryption table, but lookup is based on Nm

An attack on pre-draft 13 packet encryption




```
+-+-+-+-+-+-+-+-+
|0|1|S|R|R|K|P P|  S: Spin bit  K: Key phase bit
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Destination Connection ID (0..144)            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Packet Number (8/16/24/32)               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Protected Payload                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Digression: are CID Authenticated in QUIC?



N.B. some implementation use Y as IV for encryption of T, but this is not required by specification

Specifying QUIC Packet Formats

everparse

- We define combinators for correct parsers:

```
type parser t = b:bytes -> option (t * n:nat{n <= length b})
type serializer (p: parser t) = f:(t -> bytes) {
   forall x. p (f x) == Some (x, length (f x))
}
```

- With a well-known monadic structure:

```
val return: parser unit
val bind: parser t -> (t -> parser t') -> parser t'
val seq: parser t -> parser t' -> parser (t * t')
val map: f:(t -> t') -> parser t -> parser t'
```

New Bitfield & Bitsum combinators

- We can slice arbitrary integer types into a list of bit-aligned fields
- We can also define tagged unions based on a field of a bitfield
- For instance, a packet is long or short based on 2 msb bits of flags
- Similarly, we can express length dependencies, e.g. 2 lsb bits of flags encode the packet number length

```
let rec valid_bitfield_widths (lo: ℕ) (hi: ℕ { lo ≤ hi })
  (l: list ℕ) : Tot bool (decreases l) =
  match l with
  | [] → lo = hi
  | sz :: q → lo + sz ≤ hi && valid_bitfield_widths (lo + sz) hi q

noextract
let rec bitfields (#tot: pos) (#t: Type0) (cl: uint_t tot t)
  (lo: ℕ) (hi: ℕ { lo ≤ hi ∧ hi ≤ tot })
  (l: list ℕ { valid_bitfield_widths lo hi l })
  : Tot Type0 (decreases l) =
  match l with
  | [] → unit
  | [sz] → bitfield cl sz
  | sz :: q → bitfield cl sz & bitfields cl (lo + sz) hi q

noextract
let rec bitsum'_type'
  (#tot: pos)
  (#t: eqtype)
  (#cl: uint_t tot t)
  (#bitsum'_size: ℕ)
  (b: bitsum' cl bitsum'_size)
: Tot Type0
  (decreases (bitsum'_size))
= match b with
  | BitStop _ → unit
  | BitField sz rest → (bitfield cl sz & bitsum'_type' rest)
  | BitSum' key key_size e payload →
    (key: enum_key e & bitsum'_type' (payload key))
```

Example: flags

```
inline_for_extraction
type header_form_t =
  | Long
  | Short

[@filter_bitsum'_t_attr]
inline_for_extraction
noextract
let header_form : enum header_form_t (bitfield uint8 1) = [
  Long, 1uy;
  Short, 0uy;
]

[@filter_bitsum'_t_attr]
inline_for_extraction
noextract
let fixed_bit : enum unit (bitfield uint8 1) = [
  (), 1uy;
]

inline_for_extraction
type long_packet_type_t =
  | Initial
  | ZeroRTT
  | Handshake
  | Retry

[@filter_bitsum'_t_attr]
inline_for_extraction
noextract
let long_packet_type : enum long_packet_type_t (bitfield uint8 2) = [
  Initial, 0uy;
  ZeroRTT, 1uy;
  Handshake, 2uy;
  Retry, 3uy;
]
```

```
inline_for_extraction
noextract
let reserved_bits : enum unit (bitfield uint8 2) = [
  (), 0uy;
]

[@filter_bitsum'_t_attr]
inline_for_extraction
noextract
let packet_number_length : enum packet_number_length_t (bitfield uint8 2) = [
  1ul, 0uy;
  2ul, 1uy;
  3ul, 2uy;
  4ul, 3uy;
]

[@filter_bitsum'_t_attr]
inline_for_extraction
noextract
let rrpp : bitsum' uint8 4 =
  BitSum' _ _ reserved_bits (λ _ →
    BitSum' _ _ packet_number_length (λ _ →
      BitStop ()))

[@filter_bitsum'_t_attr]
inline_for_extraction
noextract
let first_byte : bitsum' uint8 8 =
  BitSum' _ _ header_form (function
    | Short →
      BitSum' _ _ fixed_bit (λ _ →
        BitField (* spin bit *) 1 (
          BitSum' _ _ reserved_bits (λ _ →
            BitField (* key phase *) 1 (
              BitSum' _ _ packet_number_length (λ _ →
                BitStop ()
              )
            )
          )
        )
      )
    | Long →
      BitSum' _ _ fixed_bit (λ _ →
        BitSum' _ _ long_packet_type (function
          | Retry → BitField (* unused *) 4 (BitStop ())
          | _ → rrpp
        )
      )
  )
```

Theorem: QUIC Header Format

- Correctness of header parsing
- (conditional) Non-malleability of modified draft 14 format with PN length in flags

```
noeq
type h_result =
| H_Success:
  h: header →
  c: bytes →
  h_result
| H_Failure

val parse_header: cid_len: ℕ { cid_len ≤ 20 } → last: ℕ { last + 1 < pow2 62 } → b:bytes →
GTot (r: h_result {
  match r with
  | H_Failure → T
  | H_Success h c →
    is_valid_header h cid_len last ∧
    Seq.length c ≤ Seq.length b ∧
    c `Seq.equal` Seq.slice b (Seq.length b - Seq.length c) (Seq.length b)
})

val lemma_header_parsing_correct:
  h: header →
  c: bytes →
  cid_len: ℕ { cid_len ≤ 20 } →
  last: ℕ { last + 1 < pow2 62 } →
  Lemma
  (requires (
    is_valid_header h cid_len last
  ))
  (ensures (
    parse_header cid_len last S.(format_header h @| c)
    == H_Success h c))

// N.B. this is only true for a given DCID len
val lemma_header_parsing_safe: cid_len: ℕ → last: ℕ → b1:bytes → b2:bytes → Lemma
  (requires (
    cid_len ≤ 20 ∧
    last + 1 < pow2 62 ∧
    parse_header cid_len last b1 == parse_header cid_len last b2
  ))
  (ensures parse_header cid_len last b1 == H_Failure ∨ b1 = b2)
```

Theorem: QUIC Header Encryption Correctness

Inherited CID length condition

N.B. correctness is conditional on the state of the recipient

```
// Header protection only
val header_encrypt: a:ea →
  hpk: lbytes (ae_keysize a) →
  h: header →
  c: cbytes' (is_retry h) →
  GTot packet

noeq
type h_result =
| H_Success:
  h: header →
  cipher: cbytes' (is_retry h) →
  rem: bytes →
  h_result
| H_Failure

// Note that cid_len cannot be parsed from short headers
val header_decrypt: a:ea →
  hpk: lbytes (ae_keysize a) →
  cid_len: ℕ { cid_len ≤ 20 } →
  last: ℕ { last + 1 < pow2 62 } →
  p: packet →
  GTot (r: h_result { match r with
  | H_Failure → ⊤
  | H_Success h c rem →
    is_valid_header h cid_len last ∧
    S.length rem ≤ S.length p ∧
    rem `S.equal` S.slice p (S.length p - S.length rem) (S.length p)
  })

// This is just functional correctness, but does not guarantee security:
// decryption can succeed on an input that is not the encryption
// of the same arguments (see QUIC.Spec.Old.*_malleable)
val lemma_header_encryption_correct:
  a:ea →
  k:lbytes (ae_keysize a) →
  h:header →
  cid_len: ℕ { cid_len ≤ 20 ∧ (MShort? h ⟹ cid_len == dcid_len h) } →
  last: ℕ { last + 1 < pow2 62 ∧ ((¬ (is_retry h)) ⟹ in_window (U32.v (pn_length h) - 1)
last (U64.v (packet_number h))) } →
  c: cbytes' (is_retry h) { has_payload_length h ⟹ U64.v (payload_length h) == S.length c
} →
  Lemma (
    header_decrypt a k cid_len last (header_encrypt a k h c)
    == H_Success h c S.empty)
```

Decryption window condition

- Incorrectly specified in QUIC up to draft 23
- We corrected the reference packet number decoding function
- Patched in draft 24

```
let bound_npn' (pn_len:ℕ { pn_len < 4 })
  : Tot (y: ℕ {y == pow2 (8 `op_Multiply` (pn_len + 1)) }) =
  assert_norm (pow2 8 == 256);
  assert_norm (pow2 16 == 65536);
  assert_norm (pow2 24 == 16777216);
  assert_norm (pow2 32 == 4294967296);
  match pn_len with
  | 0 → 256
  | 1 → 65536
  | 2 → 16777216
  | 3 → 4294967296

let in_window (pn_len: ℕ { pn_len < 4 }) (last pn:ℕ) =
  let h = bound_npn' pn_len in
  (last+1 < h/2 ∧ pn < h) ∨
  (last+1 ≥ U64.v uint62_bound - h/2 ∧ pn ≥ U64.v uint62_bound - h) ∨
  (last+1 - h/2 < pn ∧ pn ≤ last+1 + h/2)
```

Theorem: QUIC Packet Encryption Correctness

```
val lemma_encrypt_correct:
  a: ea →
  k: lbytes (AEAD.key_length a) →
  siv: lbytes 12 →
  hpk: lbytes (ae_keysize a) →
  h: header →
  cid_len: ℕ { cid_len ≤ 20 ∧ (MShort? h ⟹ cid_len == dcid_len h) } →
  last: ℕ{last+1 < pow2 62 } →
  p: pbytes' (is_retry h)  { has_payload_length h ⟹ U64.v (payload_length h) == S.length p
+ AEAD.tag_length a } → Lemma
  (requires (
    (¬ (is_retry h)) ⟹ (
      in_window (U32.v (pn_length h) - 1) last (U64.v (packet_number h))
  )))
  (ensures (
    decrypt a k siv hpk last cid_len
      (encrypt a k siv hpk h p)
    == Success h p Seq.empty
  ))
```

Going back to Security Model

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:*

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

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

Packet stream security definition

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**

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

$\quad T[C, H] \leftarrow (c_e, M)$

$\quad c_e \leftarrow c_e + 1$

else

$\quad C, S' \leftarrow \mathsf{SE.enc}(S, M)$

$\quad S \leftarrow S'$

return C

Oracle $\mathsf{Decrypt}(C, H)$

if $b = 1$ **then**

$\quad c, M \leftarrow T[C, H]$

$\quad$ **if** $|c - c_d| >= 2^{L_N - 1}$ **then**

In-window condition

$\quad\quad$ **return** $\perp$

$\quad c_d \leftarrow max(c, c_d)$

Decryption state

else

$\quad M, S' \leftarrow \mathsf{SE.dec}(S, C, H)$

$\quad S \leftarrow S'$

return M

Proof: code-based assumptions

```
(* 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
```

```
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)
```

Proof: code-based reduction

- We need a stronger AE assumption that guarantees that each sample is unique in the ciphertext table (paper step)
- Then, decryption can be implemented by a lookup to the PRF table based on the sample value for decrypting the header
- If the nonce and AAD match the result is a lookup in the AE table based on reconstructed nonce & plain header

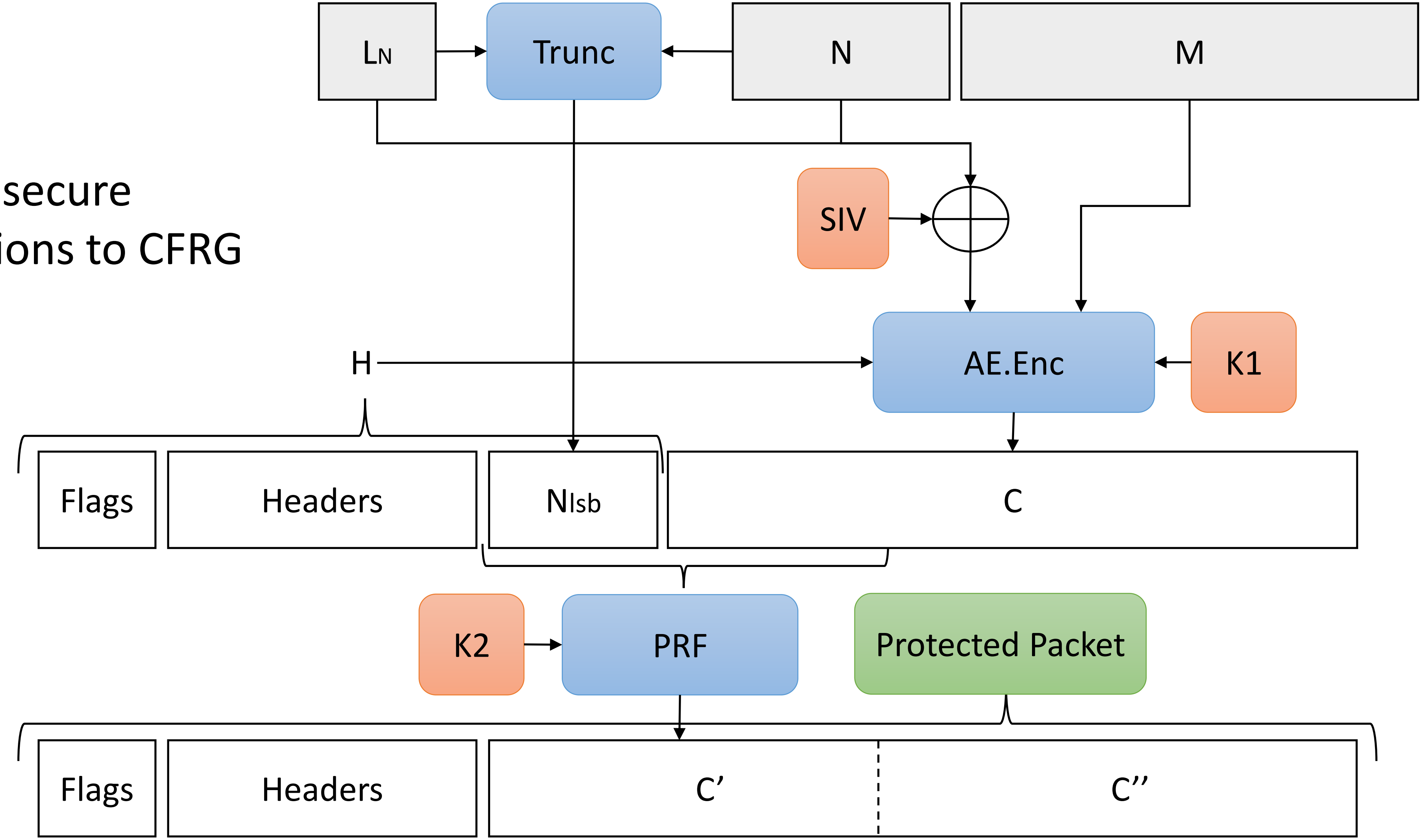
```
val encrypt
 (#k:id)
 (w:stream_writer k)
 (#hl:headerlen)
 (hd:quic_header k hl)
 (nl:pnlen {hl + nl ≤ v AEAD.aadmax})
 (#l:plainlen {hl + nl + l + v AEAD.taglen ≤ pow2 32 − 1 ∧
   nl + l + v AEAD.taglen ≥ samplelen + 4})
 (p:plain (fst k) l):
 ST (quic_packet k hl (nl + l))
 (requires fun h0 →
   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)
```

Remaining issues with QPE

- Some implementations skip the payload decryption based on the value of the decrypted packet number
- Very tricky to implement constant-time decryption: first decrypt 2 lsb of flags, then truncate mask, then decrypt PN… unsafe
- Authentication of Ln still depends on the header formatting, we prefer to include Ln in nonce (2 msb are not used)

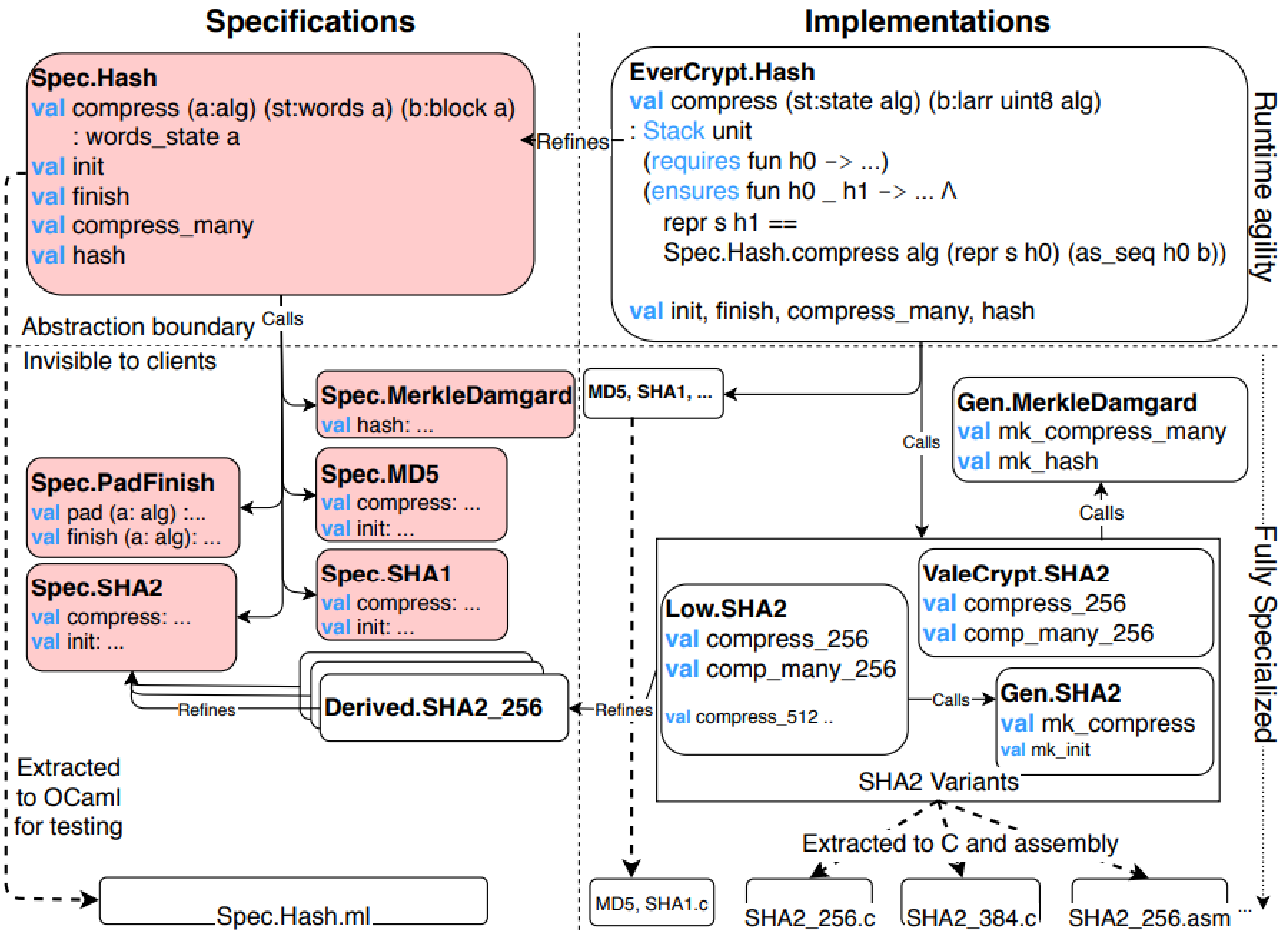
A simplified construction

Idea: propose provably secure nonce-hiding constructions to CFRG



Low-level Verified Implementation

- We use the EverCrypt implementation & Specification for AEAD (EverCrypt.AEAD) and PRF (EverCrypt.Cipher)
- Abstract crypto state for stream encryption instances
- Abstract specifications
- Multiple implementations



Low* Interface

```
inline_for_extraction noextract
let create_in_st (i:index) =
  r:HS.rid →
  dst: B.pointer (B.pointer_or_null (state_s i)) →
  initial_pn:u62 →
  traffic_secret:B.buffer U8.t {
    B.length traffic_secret = Spec.Hash.Definitions.hash_length i.hash_alg
  } →
  ST error_code
    (requires λ h0 →
      // JP: we could require that ``dst`` point to NULL prior to calling
      // ``create`` (otherwise, it's a memory leak). Other modules don't enforce
      // this (see AEAD) so for now, let's make the caller's life easier and not
      // demand anything.
      ST.is_eternal_region r ∧
      B.live h0 dst ∧ B.live h0 traffic_secret ∧
      B.disjoint dst traffic_secret)
    (ensures (λ h0 e h1 →
      match e with
      | UnsupportedAlgorithm →
          B.(modifies loc_none h0 h1)
      | Success →
          let s = B.deref h1 dst in
          not (B.g_is_null s) ∧
          invariant h1 s ∧

          B.(modifies (loc_buffer dst) h0 h1) ∧
          B.fresh_loc (footprint h1 s) h0 h1 ∧

          g_initial_packet_number (B.deref h1 s) == U64.v initial_pn
      | _ →
          ⊥))
```

Memory Safety

Functional Correctness

```
val encrypt: #i:G.erased index → (
  let i = G.reveal i in
  s: state i →
  dst: B.buffer U8.t →
  dst_pn: B.pointer u62 →
  h: header →
  plain: B.buffer U8.t →
  plain_len: U32.t →
  Stack error_code
    (requires λ h0 →
      // Memory & preservation
      B.live h0 plain ∧ B.live h0 dst ∧ B.live h0 dst_pn ∧
      header_live h h0 ∧
      B.(all_disjoint [ footprint h0 s; loc_buffer dst; loc_buffer dst_pn; header_footprint
h; loc_buffer plain ]) ∧
      invariant h0 s ∧
      incrementable s h0 ∧
      B.length plain == U32.v plain_len ∧ (
      let clen = if is_retry h then 0 else U32.v plain_len + Spec.Agile.AEAD.tag_length
i.aead_alg in
      (if is_retry h then U32.v plain_len == 0 else 3 ≤ U32.v plain_len ∧ U32.v plain_len <
QSpec.max_plain_length) ∧
      (has_payload_length h ⟹ U64.v (payload_length h) == clen) ∧
      B.length dst == U32.v (header_len h) + clen
    ))
    (ensures λ h0 r h1 →
      match r with
      | Success →
        // Memory & preservation
        B.(modifies (footprint_s h0 (deref h0 s) `loc_union` loc_buffer dst `loc_union`
loc_buffer dst_pn)) h0 h1 ∧
        invariant h1 s ∧
        footprint_s h1 (B.deref h1 s) == footprint_s h0 (B.deref h0 s) ∧ (
        // Functional correctness
        let s0 = g_traffic_secret (B.deref h0 s) in
        let open QUIC.Spec in
        let k = derive_secret i.hash_alg s0 label_key (Spec.Agile.AEAD.key_length
i.aead_alg) in
        let iv = derive_secret i.hash_alg s0 label_iv 12 in
        let pne = derive_secret i.hash_alg s0 label_hp (ae_keysize i.aead_alg) in
```

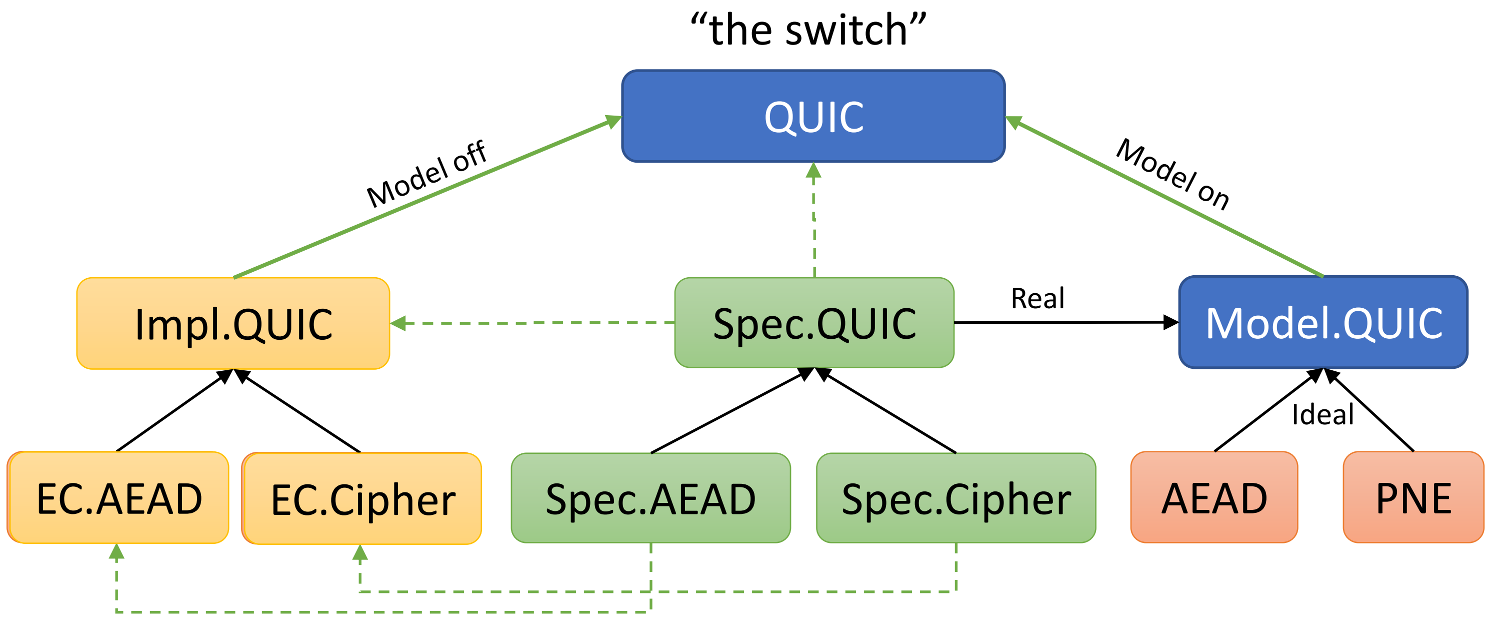
Extracted C interface

```c
typedef struct QUIC_Impl_Base_long_header_specifics_s
{
  QUIC_Impl_Base_long_header_specifics_tags tag;
  union {
    struct {
      uint64_t payload_length;
      uint32_t packet_number_length;
      uint8_t *token;
      uint32_t token_length;
    } case_BInitial;
    struct {
      uint64_t payload_length;
      uint32_t packet_number_length;
    } case_BZeroRTT;
    struct {
      uint64_t payload_length;
      uint32_t packet_number_length;
    } case_BHandshake;
    struct {
      uint8_t unused;
      uint8_t *odcid;
      uint32_t odcil;
    } case_BRetry;
  };
} QUIC_Impl_Base_long_header_specifics;
```

```c
typedef struct QUIC_Impl_Base_header_s
{
  QUIC_Impl_Base_header_tags tag;
  union {
    struct {
      uint32_t version;
      uint8_t *dcid;
      uint32_t dcil;
      uint8_t *scid;
      uint32_t scil;
      QUIC_Impl_Base_long_header_specifics spec;
    } case_BLong;
    struct {
      bool spin;
      bool phase;
      uint8_t *cid;
      uint32_t cid_len;
      uint32_t packet_number_length;
    } case_BShort;
  };
} QUIC_Impl_Base_header;
```

```c
// Opaque state
typedef struct QUIC_Impl_state_s_s QUIC_Impl_state_s;

EverCrypt_Error_error_code
QUIC_Impl_create_in(
  QUIC_Impl_index i1,
  QUIC_Impl_state_s **dst,
  uint64_t initial_pn,
  uint8_t *traffic_secret
);

EverCrypt_Error_error_code
QUIC_Impl_encrypt(
  QUIC_Impl_state_s *s,
  uint8_t *dst,
  uint64_t *dst_pn,
  QUIC_Impl_Base_header h1,
  uint8_t *plain,
  uint32_t plain_len
);

typedef struct QUIC_Impl_result_s
{
  uint64_t pn;
  QUIC_Impl_Base_header header;
  uint32_t header_len;
  uint32_t plain_len;
  uint32_t total_len;
}
QUIC_Impl_result;

EverCrypt_Error_error_code
QUIC_Impl_decrypt(
  QUIC_Impl_state_s *s,
  QUIC_Impl_result *dst,
  uint8_t *packet,
  uint32_t packet_len,
  uint8_t cid_len
);
```

Proving security for the low-level implementation



Properties of the switch

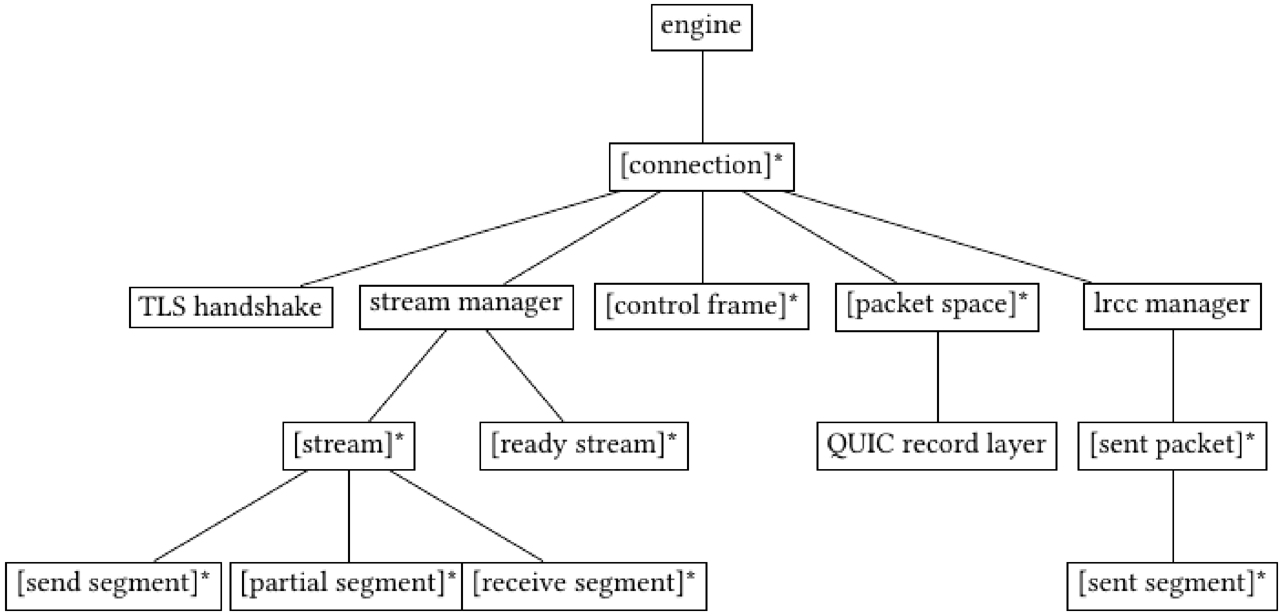
- Preserve perfect confidentiality of ideal plaintexts
- Preserve secret independence of concrete plaintexts

Properties of the concrete spec:

- Conditions for packet formal injectivity
- Conditions (+failure cases) for header encryption correctness
- Conditions for packet number decoding success (+errors)
- Conditions for packet decryption correctness

A safe implementation of the QUIC transport

- We want to integrate our QUIC verified record layer into a full stack implementation for benchmark & interop testing
- The implementation is verified for memory safety in Dafny (with assumed FFI for TLS & QPE)
- This lays the foundation for a fully verified reference implemention

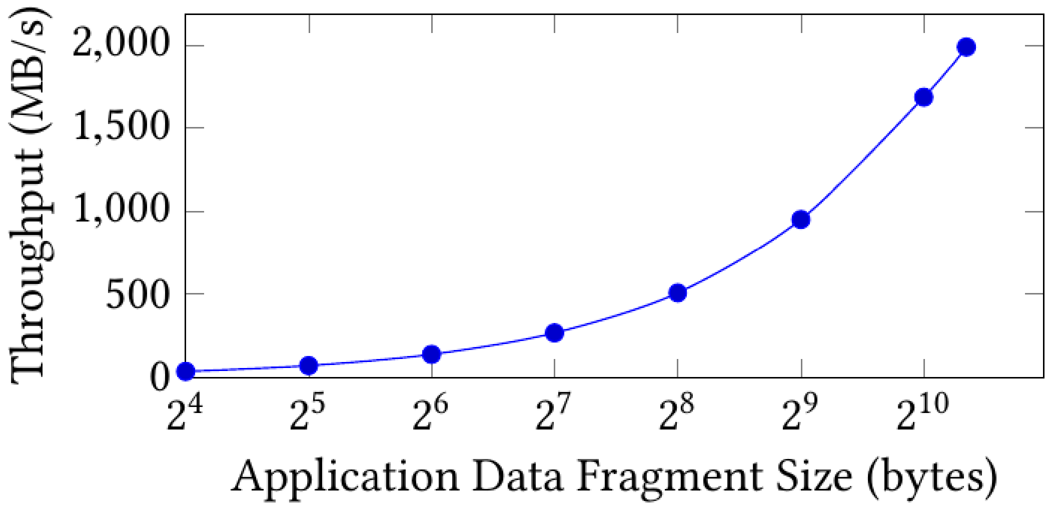


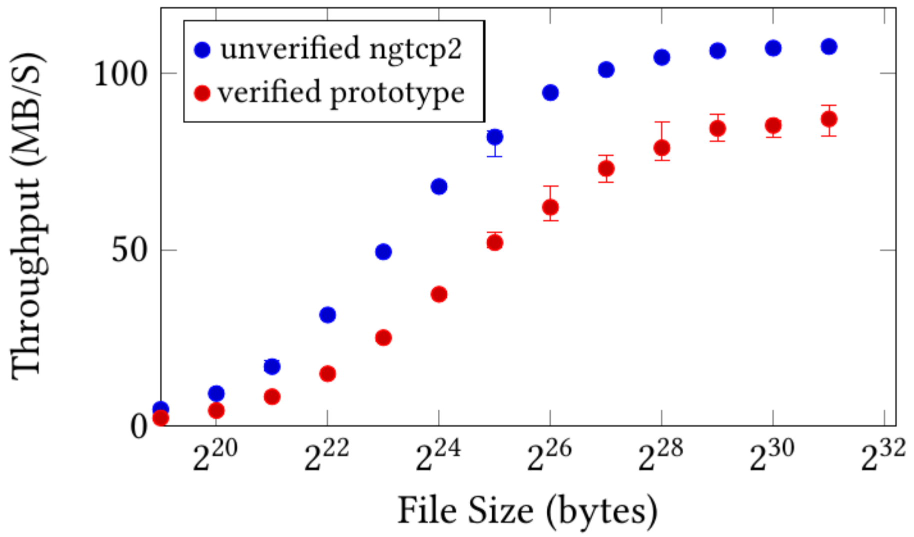
Structure of Dafny modules in our QUIC implementation

Evaluation

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

Performance of verified packet encryption





Performance of integrated QUIC implementation

Conclusions

- We have formally specified QUIC packet encryption, and proved its correctness & cryptographic security (in a nonce-hiding model)
- We have a safe, secure & correct low-level implementation
- Our fully verified packet stream encryption implementation can sustain ~2GB/s of QUIC payload throughput on a single core
- Our memory safe implementation of the QUIC protocol is within 20% of ngtcp2 in representative file download benchmarks

TR https://eprint.iacr.org/2020/114
GitHub https://github.com/project-everest/everquic-crypto