Wireguard: A Modern VPN Protocol

Jinank Jain, Rayhaan Jaufeerally

July 17, 2018

Network Security Group, ETH Zürich

Introduction

Why VPN?

- Necessary for point to point security between campuses (e.g. DC's, corporate offices, . . .)
  - As early as the 1970's governments have tapped undersea cables for intelligence,
  - Operation Ivy Bells in 1971 tapped Russian communicatioons to millitary bases[1],
- Necessary for end users to get a clean connection:
  - ISP's doing DNS hijacking to serve inappropriate content,
  - Open WiFi networks when travelling,
  - Geoblocking,

[1] https://en.wikipedia.org/wiki/Operation_Ivy_Bells

Necessity in real life

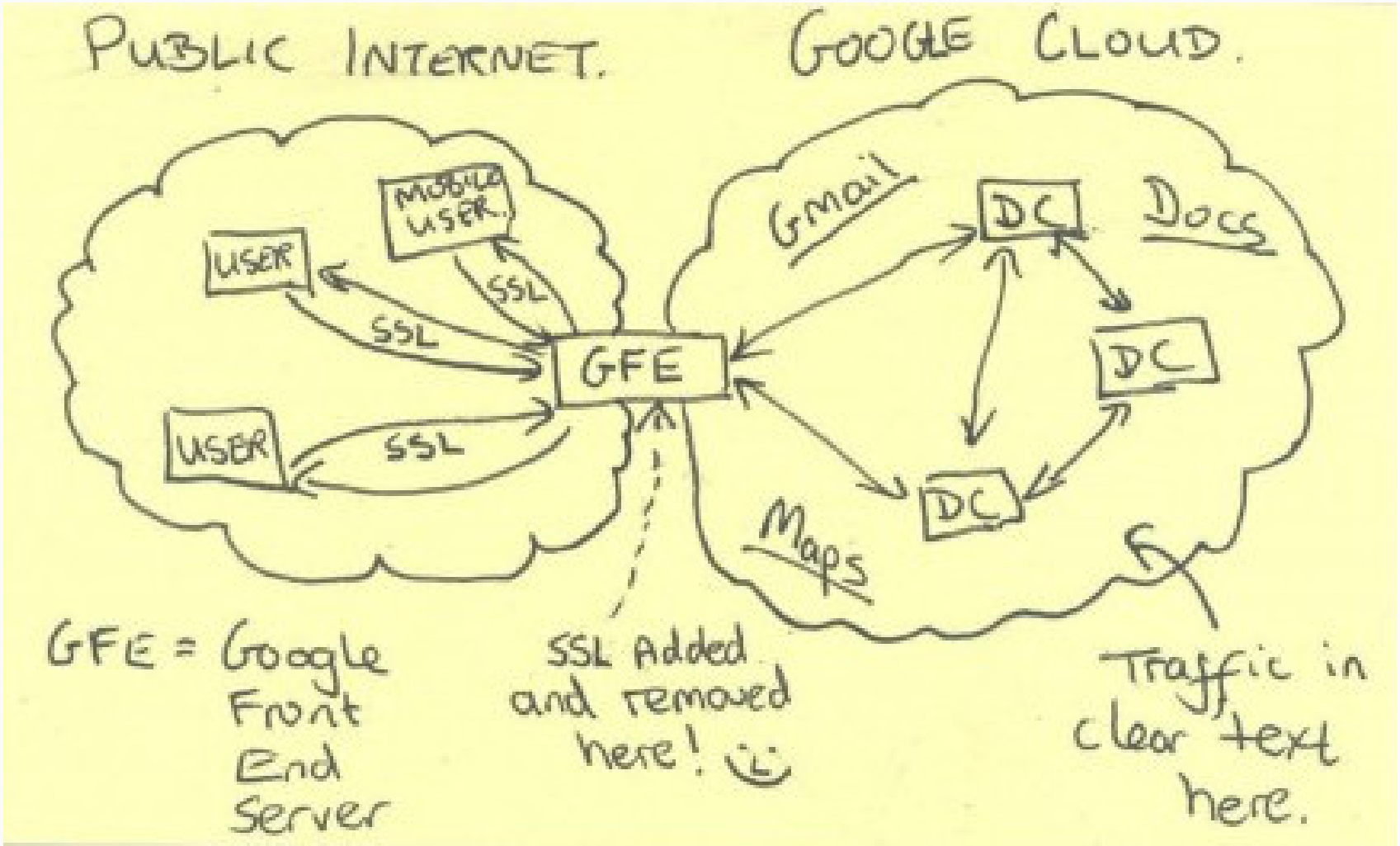


Figure 1: Nation state surveillance of user data including SPII.

History of VPN protocols

- IPSEC
  - Popular for site to site connections with dedicated router hardware,
  - Tedious to set up and high degree of complexity,
  - Large attack surface between IKE (v2), SA mechanisms, XFRM in Linux,
  - Legacy protocol support,
  - IP in IP,
- OpenVPN
  - Implemented in userspace with TUN/TAP (slow),
  - Complex confoguration vulnerable to leaks,
  - Stateful protocol which is brittle in real networks,
  - Large codebase / attack surface,

What is Wireguard?

- Opinionated Layer 3 secure network tunnel for IPv4 and IPv6.
- Lives in the Linux kernel, but cross platform userspace implementations are available.
- UDP based. Punches through firewall.
- Conservative and modern cryptographic principles.
- Emphasis on simplicity and single user auditability.
- Authentication model similar to SSH's `authorized_keys`.

Easily Auditable

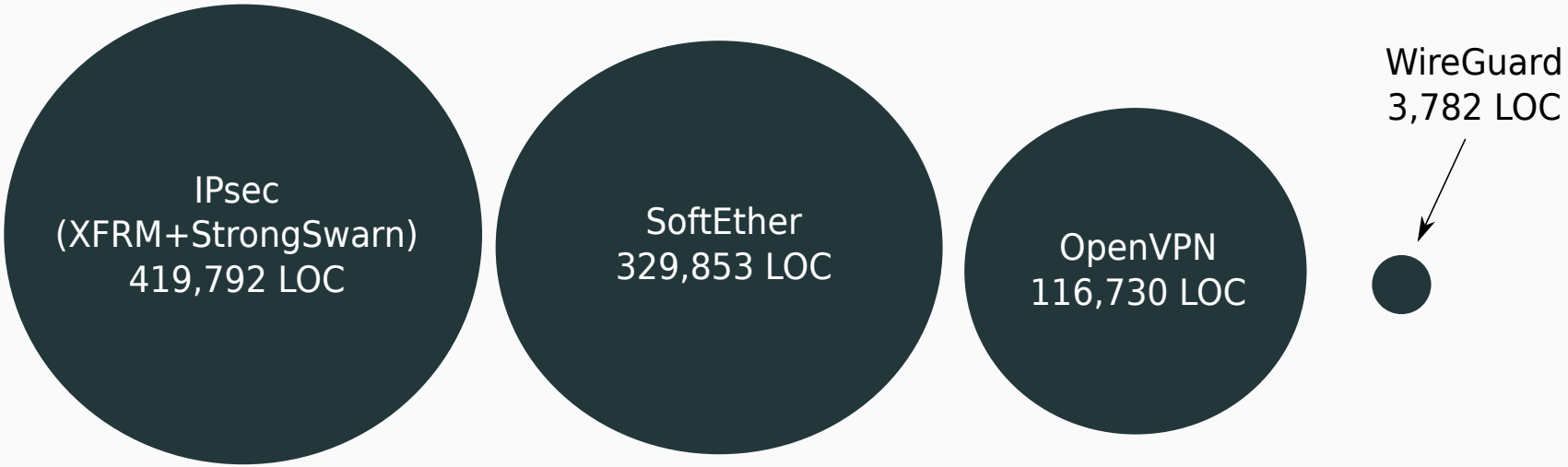


Figure 2: Comparing different VPN protocols in terms of LOC

Minimialistic Interface

"Developers should write programs that can communicate easily with other programs"

— Unix Philosophy

- Wireguard presents a normal network interface

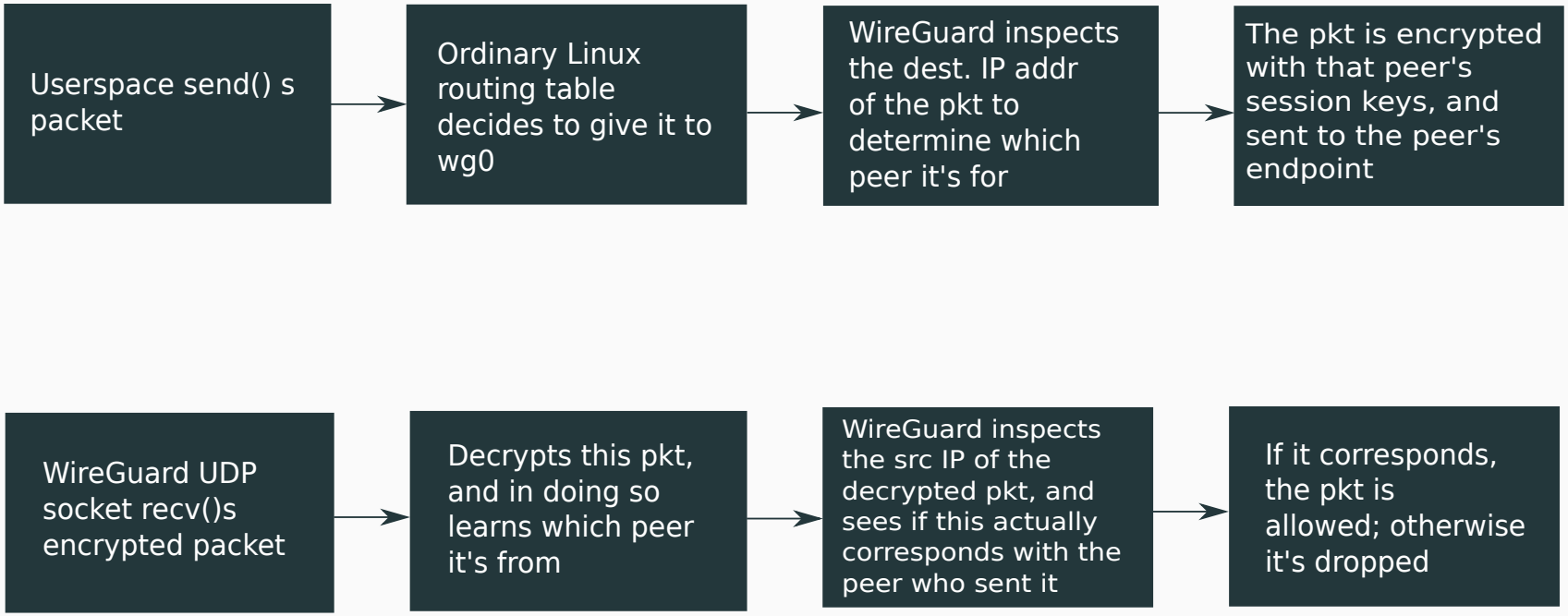
```
ip link add wg0 type wireguard
ip address add 10.0.32.1/24 dev wg0
ip route add default via wg0
```

- By using a standard interface it becomes easier to administer using the existing iproute2 utilities for example

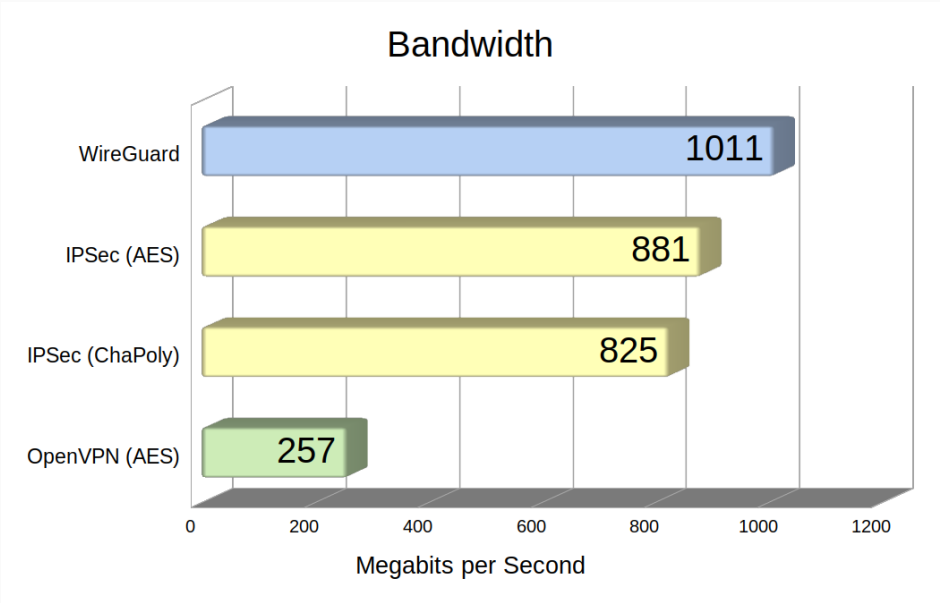
Cryptokey Routing

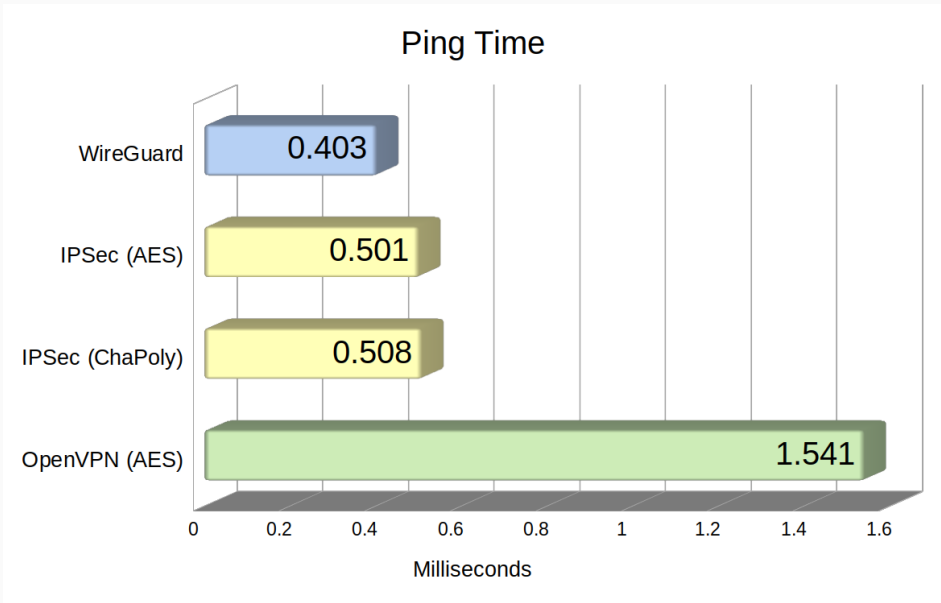
- Fundamental concept of any VPN service
  - Create **mapping** between **public keys of peers** and their **IPs**.
- WireGuard interface has:
  - A private key
  - A listening UDP port
  - A list of peers
- Peer has
  - A public key
  - A list of associated tunnel IPs
  - Optionally has an endpoint IP and port

Cryptokey Routing



Performance





Stealth

- Behaves like a rootkit in some sense.
- Should not respond to any unauthenticated packets.
- Hinder scanners and service discovery.
- Service only responds to packets with correct crypto.
- Not chatty at all.

Protocol design

Handshake DoS

- Verifying authenticity using Curve25519 is expensive,
- Under load a server may send a cookie which needs to be echoed back,
- This proves IP ownership, hence IP rate limiting can be used (e.g. token bucket).
- The cookie is the MAC over the source IP with a secret which changes every 120s.

Flaws to be addressed

- Indescriminate cookie responses violate silence property,
- Cleartext cookies are vulnerable to MiTM replay attacks which cause extra computation,
- Initator itself could be DoS'ed by being sent fake cookies

Issue 1: Silence

- For the responser to remain silent, all messages have a first MAC using the responder's public key,
- This proves that a peer knows to whom it is talking,
- While this public key is not secret, it is acceptable in the threat model to say if the initator knows the public key, then it knows of the existance of the server,
- This MAC is included in all packets as `msg.mac1`

Issue 2: Cleartext cookies

- The cookie is encrypted using `XChaCha20Poly1305 AEAD` with a randomized nonce,
- This uses the responder's public key as as symmetric encryption key

Issue 3: Fraudulent cookies

- The additional data field of AEAD to encrypt the cookie in transit is used to authenticate the first MAC,
- An attacker without an active MiTM cannot send fraudulent invalid cookie responses to prevent them from authenticating,
- This is acceptable in the threat model because a Dolev-Yao attacker could just drop packets to prevent authentication

Crypto background — ChaCha20

- A stream cipher designed by DJB based upon Salsa20,
- Uses simple operations: XOR, 32 bit addition mod 2^{32}, and constant distance rotation operations ($<<<$),
- Limiting to these instructions aims to avoid timing side channels,
- Internal state consists of sixteen 32-bit words in a 4×4 matrix

Crypto background — ChaCha20 State

Cons	Cons	Cons	Cons
Key	Key	Key	Key
Key	Key	Key	Key
Pos	Pos	Nonce	Nonce

Table 1: Initial state of ChaCha20

Constant used is “expand 32-byte k” which is a “nothing up my sleeve number”.

Crypto background — ChaCha20

```
#define ROT(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d)          \
    a += b;  d ^= a;  d = ROT(d,16);  \
    c += d;  b ^= c;  b = ROT(b,12);  \
    a += b;  d ^= a;  d = ROT(d, 8);  \
    c += d;  b ^= c;  b = ROT(b, 7)
```

Crypto background — ChaCha20

```
void chacha_block(uint32 out[16],uint32 in[16]) {
  int i; uint32 x[16];
  for (i = 0;i < 16;++i) x[i] = in[i];
  // 10 loops x 2 rounds/loop = 20 rounds.
  for (int i = 0; i < 10; i++) {
    // Odd round.
    QR(x[0], x[4], x[ 8], x[12]); // column 0
    QR(x[1], x[5], x[ 9], x[13]); // column 1
    QR(x[2], x[6], x[10], x[14]); // column 2
    QR(x[3], x[7], x[11], x[15]); // column 3
    // Even round.
    QR(x[0], x[5], x[10], x[15]); // diagonal 1 (main diagonal)
    QR(x[1], x[6], x[11], x[12]); // diagonal 2
    QR(x[2], x[7], x[ 8], x[13]); // diagonal 3
    QR(x[3], x[4], x[ 9], x[14]); // diagonal 4
  }
  for (i = 0;i < 16;++i) out[i] = x[i] + in[i];
}
```

Primitives

DH(pubkey, privkey) — Curve25519 point multiplication of `privkey` and `pubkey`, 32 bytes output,

DH_Generate() — Generates a random Curve25519 private key and computes its corresponding public key,

AEAD(key, counter, plaintext, authtext) — `ChaCha20Poly1305 AEAD` as specified in RFC 7539 with the nonce being 32 bits of zeros followed by 64bit LE value of `counter`,

XAEAD(key, nonce, plaintext, authtext) — `XChaChaPoly1305 AEAD` with 24 byte random `nonce` instantiated using `HChaCha20` and `ChaChaPoly1305`

Primitives (cont'd)

Hash(input) — `blake2s(input, 32)`

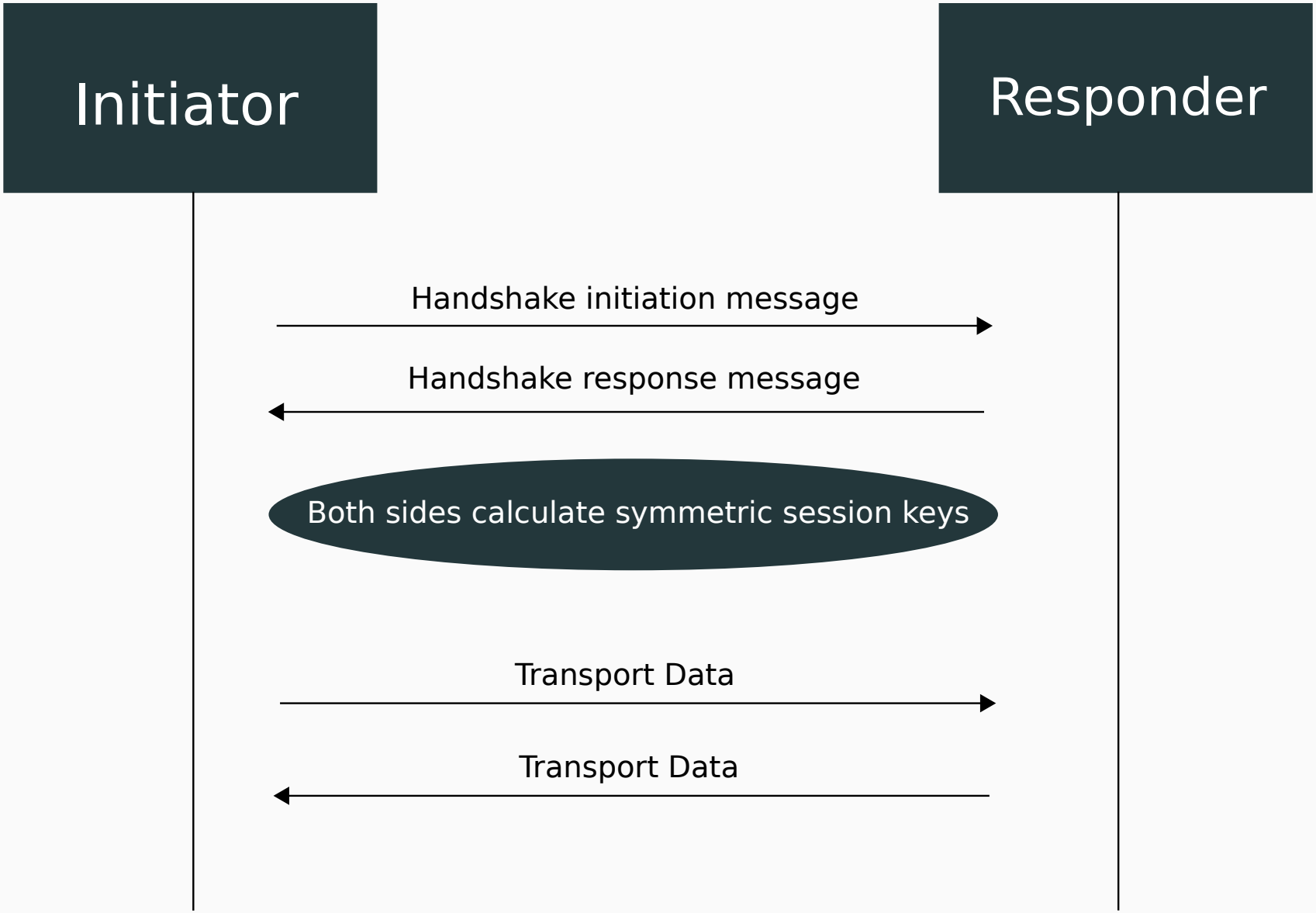
MAC(key, input) — `keyed-blake2s(key, input, 16)`, the keyed MAC variant of the BLAKE2s hash function, 16 byte output,

HMAC(key, input) — `HMAC-blake2s(key, input, 32)`, the ordinary BLAKE2s hash function used in an HMAC construction,

KDF$_n$(key, input) — Sets $\mathcal{T}_0 := HMAC(key, input), \mathcal{T}_1 := HMAC(\mathcal{T}_0, 0x1), \ldots, \mathcal{T}_i := HMAC(\mathcal{T}_0, \mathcal{T}_{i-1}||i)$

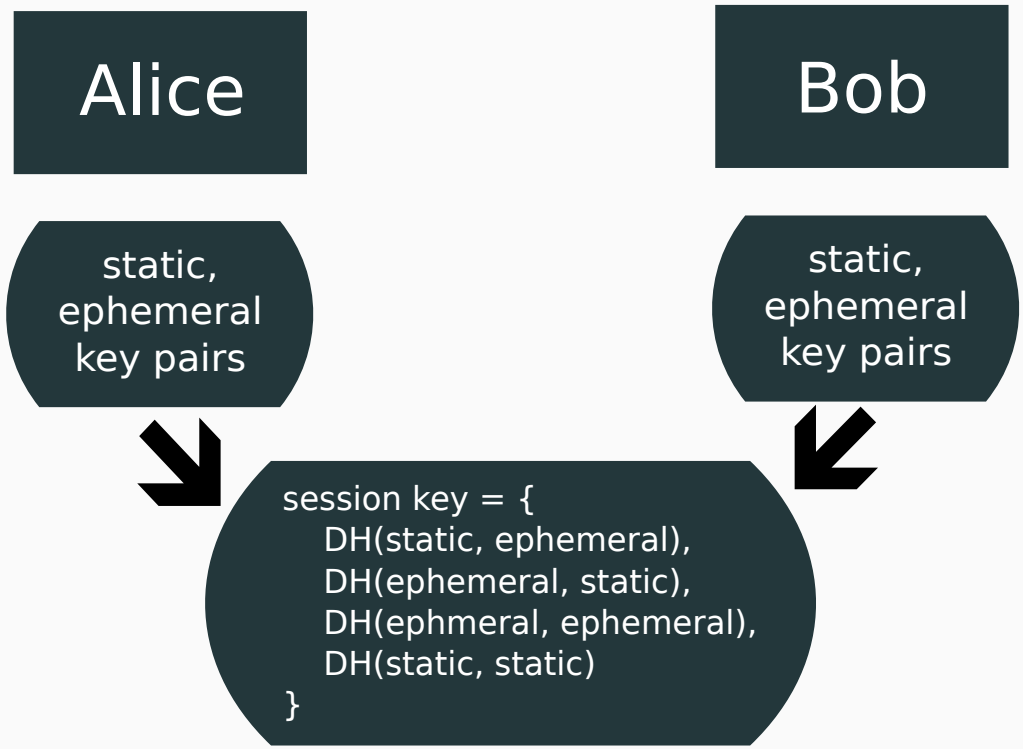
Timestamp() — Returns the TAI64N timestamp of the current time, 12 bytes output, first 8 bytes UNIX timestamp, last 4 bytes number of nanoseconds from the beginning of that second

The Key Exchange



Session key derivation - NoiseIK

- One peer is the initiator; the other is responder
- Each peer has their static identity - their long term *static keypair*
- For each new handshake, each peer generates an *ephemeral keypair*



Message 1: Initator to responder

Type := 1 (1 byte)	Reserved := 0 (3 bytes)
Sender := I_i (4 bytes)	
Ephemeral (32 bytes)	
Static (32 bytes)	
Timestamp (12 bytes)	
mac1 (16 bytes)	mac2 (16 bytes)

Figure 3: Initator to responder packet. NOT TO SCALE.

Message 1: computation

$$C_i := HASH(CONSTRUCTION) \quad (1)$$

$$H_i := HASH(C_i \,||\, IDENTIFIER) \quad (2)$$

$$H_i := HASH(H_i \,||\, S_r^{pub}) \quad (3)$$

$$(E_i^{priv}, E_i^{pub}) := DH\_generate() \quad (4)$$

$$C_i := KDF_1(C_i, E_i^{pub}) \quad (5)$$

$$\texttt{msg.ephemeral} := E_i^{pub} \quad (6)$$

$$H_i := HASH(H_i \,||\, \texttt{msg.ephemeral}) \quad (7)$$

$$(C_i, \kappa) := KDF_2(C_i, DH(E_i^{priv}, S_r^{pub})) \quad (8)$$

$$\texttt{msg.static} := AEAD(\kappa, 0, S_i^{pub}, H_i) \quad (9)$$

$$H_i := HASH(H_i \,||\, \texttt{msg.static}) \quad (10)$$

Message 1: computation (cont'd)

$$(C_i, \kappa) := KDF_2(C_i, DH(S_i^{priv}, S_r^{pub}) \quad (11)$$

$$\texttt{msg.timestamp} := AEAD(\kappa, 0, Timestamp(), H_i) \quad (12)$$

$$H_i := HASH(H_i \mid\mid \texttt{msg.timestamp}) \quad (13)$$

Message 2: Responder to initiator

type := 0x2 (1 byte)	reserved := 0 (3 bytes)
sender := I_r (4 bytes)	receiver := I_i (4 bytes)
Ephemeral (32 bytes)	
Empty (0 bytes)	
mac1 (16 bytes)	mac2 (16 bytes)

Figure 4: Responder to initiator packet. NOT TO SCALE

Message 2: computation

$$(E_r^{priv}, E_R^{pub}) := DH\_Generate() \quad (1)$$

$$C_R := KDF_1(C_r, E_r^{pub}) \quad (2)$$

$$\texttt{msg.ephemeral} := E_r^{pub} \quad (3)$$

$$H_r := HASH(C_r \mid\mid \texttt{msg.ephemeral} \quad (4)$$

$$C_r := KDF_1(C_r, DH(E_r^{priv}, E_i^{pub})) \quad (5)$$

$$C_r := KDF_1(C_r, DH(E_r^{priv}, S_i^{pub}) \quad (6)$$

$$(C_r, \mathcal{T}, \kappa) := KDF_3(C_r, Q) \quad (7)$$

$$H_r := HASH(H_r \mid\mid \mathcal{T}) \quad (8)$$

$$\texttt{msg.empty} := AEAD(\kappa, 0, \epsilon, H_r) \quad (9)$$

$$H_r := HASH(H_r, \texttt{msg.empty}) \quad (10)$$

Transport data messages

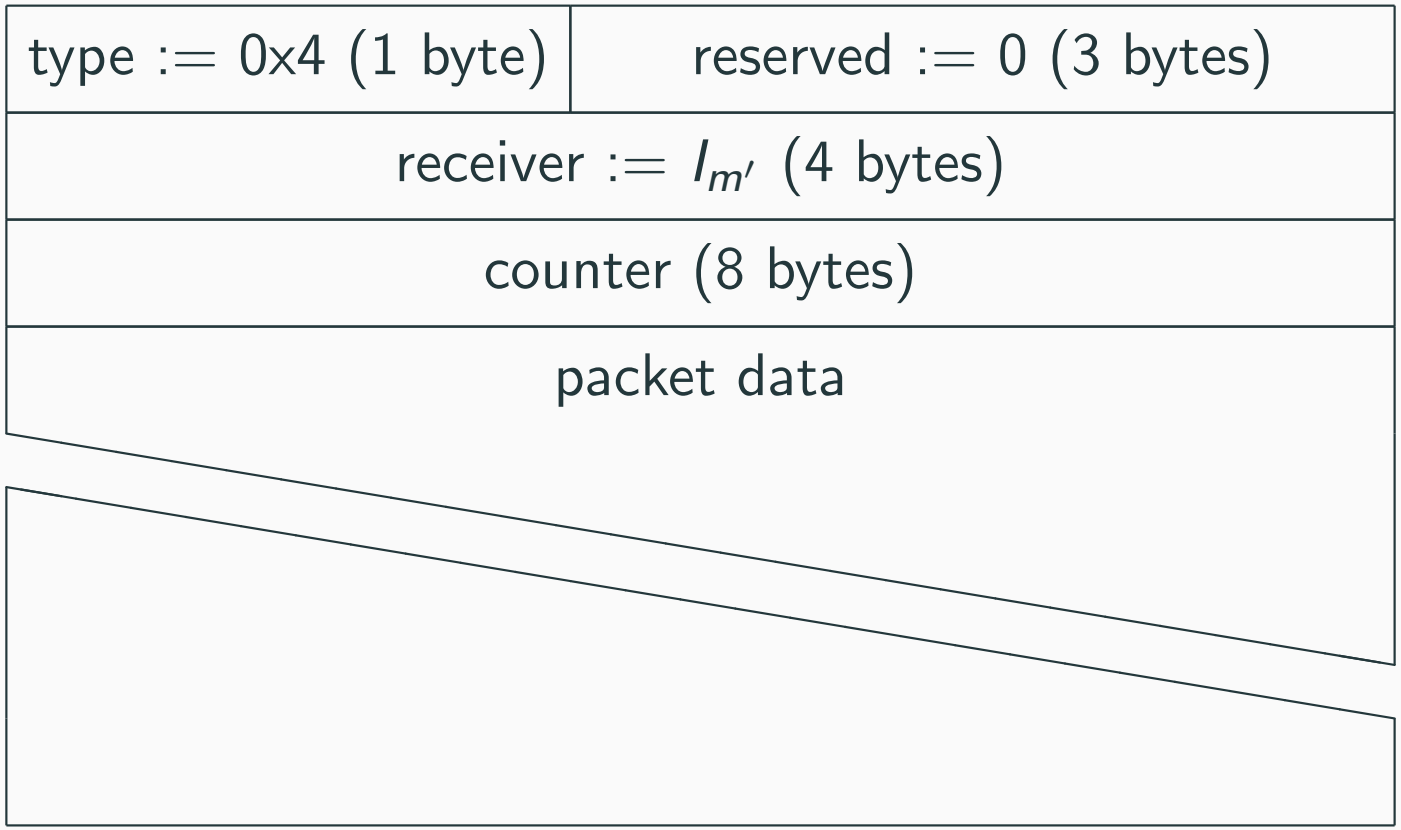


Figure 5: Payload packet. NOT TO SCALE