CSCI-UA.9480
Introduction to Computer Security

Session 2.3

Designing Secure Network Systems

Prof. Nadim Kobeissi

Goals of today's class.

A look into some secure network systems:

- WireGuard: a modern VPN.
- A critical look at ProtonMail, a secure email service.

WireGuard is an example of a *well-designed* secure network application.

ProtonMail is an example of a *badly designed* network application.

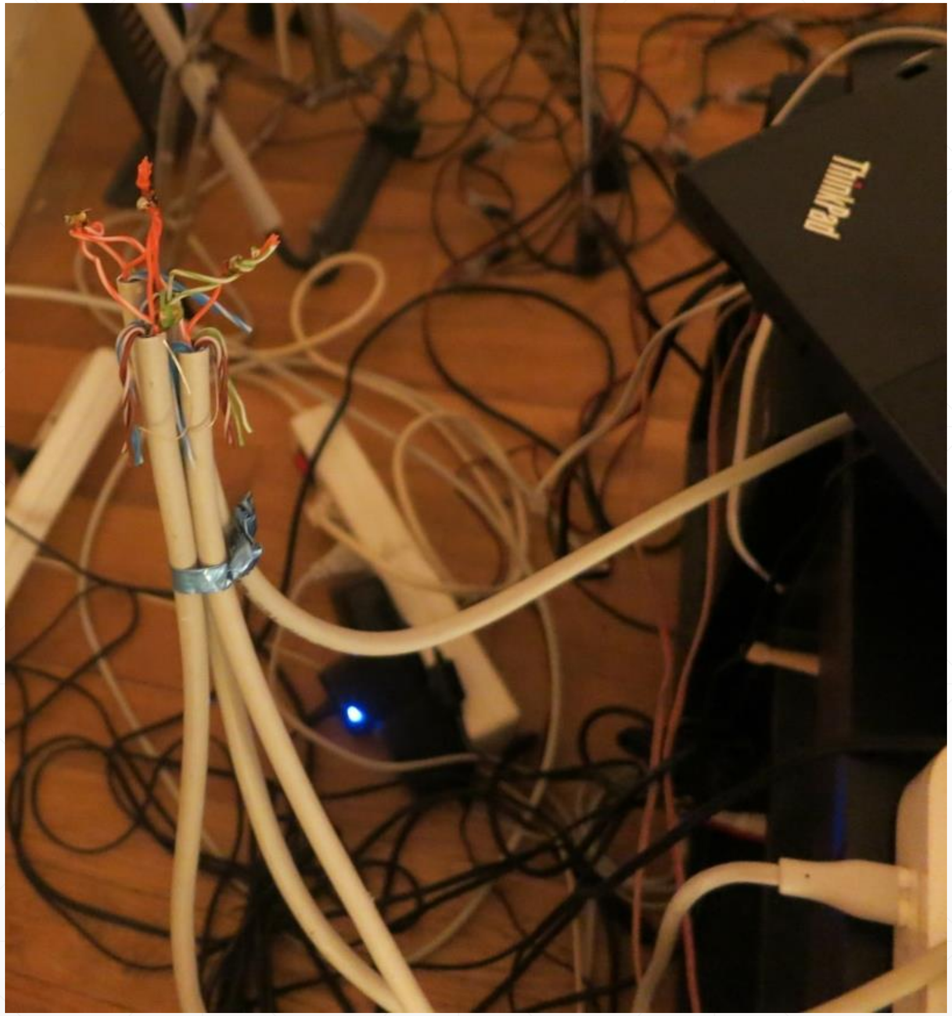
WireGuard

Following slides are by Jason A. Donenfeld, author of WireGuard.

2.3a

What is WireGuard?

- Layer 3 secure network tunnel for IPv4 and IPv6.
  - Opinionated. Only layer 3!
- *Designed* for the Linux kernel
  - Slower cross platform implementations also.
- UDP-based. Punches through firewalls.
- Modern conservative cryptographic principles.
- Emphasis on simplicity and auditability.
- Authentication model similar to SSH's `authenticated_keys`.
- Replacement for OpenVPN and IPsec.
- Grew out of a stealth rootkit project.
  - Techniques desired for stealth are equally as useful for tunnel defensive measures.

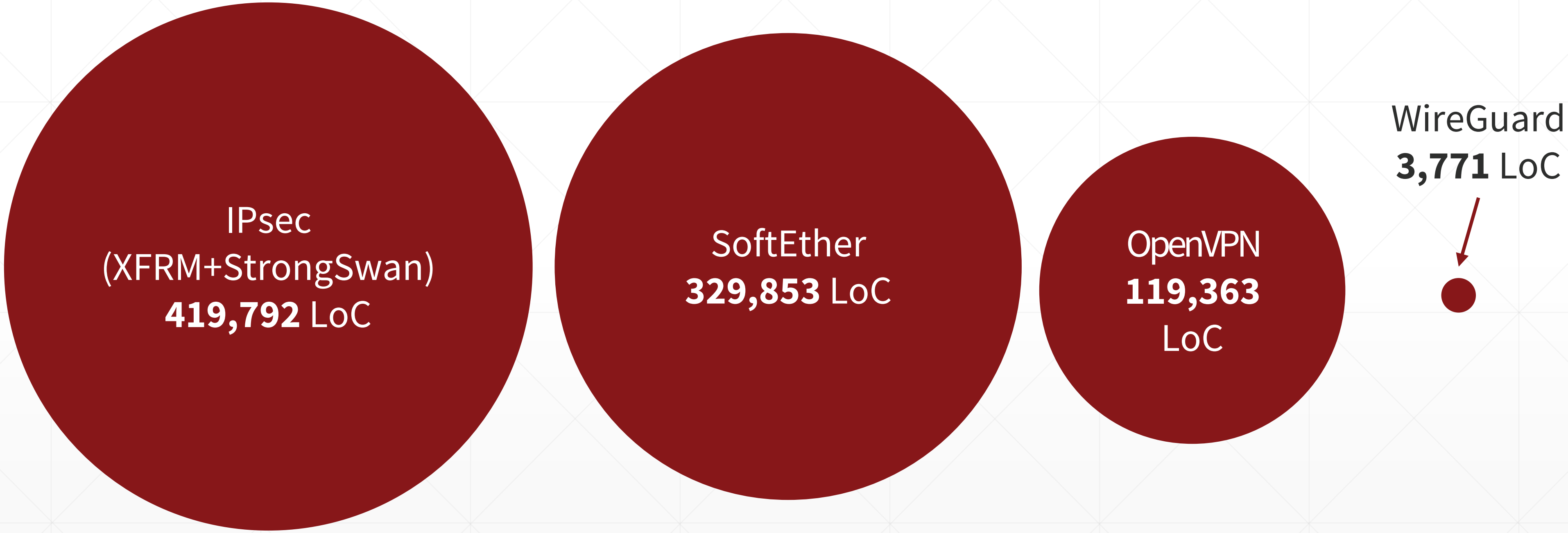


Security Design Principle 1: Easily Auditable

OpenVPN	Linux XFRM	StrongSwan	SoftEther	WireGuard
116,730 LoC Plus OpenSSL!	119,363 LoC Plus StrongSwan!	405,894 LoC Plus XFRM!	329,853 LoC	**3,771 LoC**

Less is more.

Security Design Principle 1: Easily Auditable



IPsec
(XFRM+StrongSwan)
419,792 LoC

SoftEther
329,853 LoC

OpenVPN
119,363
LoC

WireGuard
3,771 LoC

Security Design Principle 2: Simplicity of Interface

- WireGuard presents a normal network interface:

```
# ip link add wg0 type wireguard
# ip address add 192.168.3.2/24 dev wg0
# ip route add default via wg0
# ifconfig wg0 …
# iptables -A INPUT -i wg0 …

/etc/hosts.{allow,deny}, bind(), …
```

- Everything that ordinarily builds on top of network interfaces – like `eth0` or `wlan0` – can build on top of `wg0`.

Blasphemy!

- WireGuard is blasphemous!
- We break several layering assumptions of 90s networking technologies like IPsec (opinioned).
  - IPsec involves a “transform table” for outgoing packets, which is managed by a user space daemon, which does key exchange and updates the transform table.
- With WireGuard, we start from a very basic building block – the network interface – and build up from there.
- Lacks the academically pristine layering, but through clever organization we arrive at something more coherent.

Cryptokey Routing

- **The fundamental concept of any VPN is an association between public keys of peers and the IP addresses that those peers are allowed to use.**
- A WireGuard interface has:
  - A private key
  - A listening UDP port
  - A list of peers
- A peer:
  - Is identified by its public key
  - Has a list of associated tunnel IPs
  - Optionally has an endpoint IP and port

Cryptokey Routing

PUBLIC KEY :: IP ADDRESS

Cryptokey Routing

Server Config

```
[Interface]
PrivateKey =
yAnz5TF+lXXJte14tji3zlMNq+hd2rYUIgJBgB3fBmk=
ListenPort = 41414

[Peer]
PublicKey =
xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.192.122.3/32,10.192.124.1/24

[Peer]
PublicKey =
TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.192.122.4/32,192.168.0.0/16
```

Client Config

```
[Interface]
PrivateKey =
gI6EdUSYvn8ugXOt8QQD6Yc+JyiZxIhp3GInSWRfWGE=
ListenPort = 21841

[Peer]
PublicKey =
HIgo9xNzJMWLKASShiTqIybxZ0U3wGLiUeJ1PKf8ykw=
Endpoint = 192.95.5.69:41414
AllowedIPs = 0.0.0.0/0
```

Cryptokey Routing

Userspace:
`send(packet)`

Linux kernel:
Ordinary routing table
→ wg0

WireGuard:
Destination IP address
→ which *peer*

WireGuard:
`encrypt(packet)`
`send(encrypted)`
→ *peer's* endpoint

WireGuard:
`recv(encrypted)`

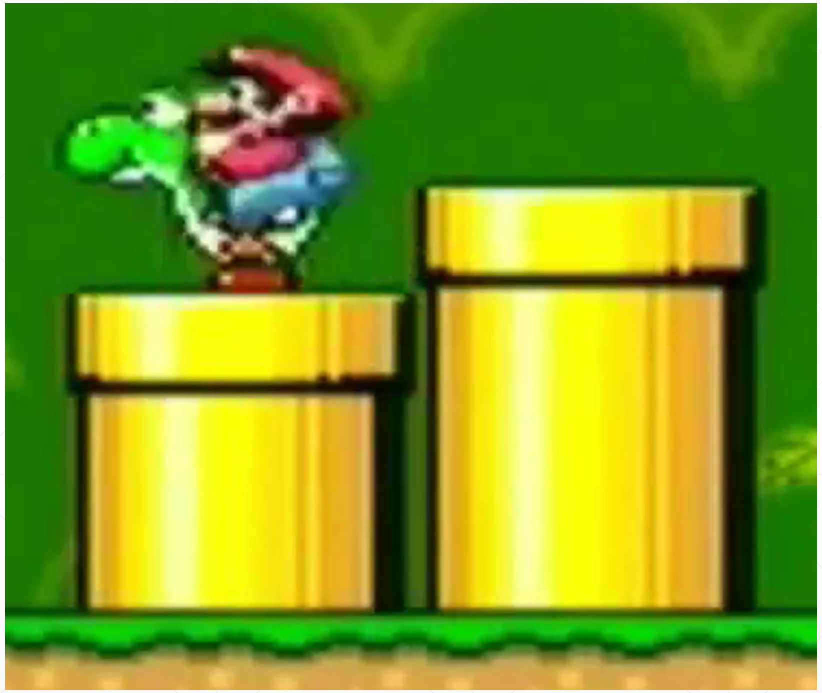
WireGuard:
`decrypt(packet)`
→ which *peer*

WireGuard:
Source IP address
↔ *peer's* allowed IPs

Linux:
Hand packet to networking stack

Cryptokey Routing

- Makes system administration very simple.
- If it comes from interface `wg0` and is from Yoshi’s tunnel IP address of `192.168.5.17`, then the packet *definitely came from Yoshi*.
- The iptables rules are plain and clear.



Timers: A Stateless Interface for a Stateful Protocol

- As mentioned prior, WireGuard appears "stateless" to user space; you set up your peers, and then it *just works*.
- A series of timers manages session state internally, invisible to the user.
- Every transition of the state machine has been accounted for, so there are no undefined states or transitions.
- Event based.

Timers

User space sends packet.	• If no session has been established for 120 seconds, send handshake initiation.
No handshake response after 5 seconds.	• Resend handshake initiation.
Successful authentication of incoming packet.	• Send an encrypted empty packet after 10 seconds, if we don’t have anything else to send during that time.
No successfully authenticated incoming packets after 15 seconds.	• Send handshake initiation.

Security Design Principle 2: Simplicity of Interface

- The interface *appears* stateless to the system administrator.
- Add an interface – `wg0`, `wg1`, `wg2`, `...` – configure its peers, and immediately packets can be sent.
- If it’s not set up correctly, most of the time it will just refuse to work, rather than running insecurely: **fails safe, rather than fails open.**
- Endpoints roam, like in mosh.
- Identities are just the static public keys, just like SSH.
- Everything else, like session state, connections, and so forth, is invisible to admin.

Demo

Simple Composable Tools

- Since wg`(8)` is a very simple tool, that works with `ip(8)`, other more complicated tools can be built on top.
- Integration into various network managers:
  - OpenWRT
  - OpenRC netifrc
  - NixOS
  - systemd-networkd
  - LinuxKit
  - Ubiquiti’s EdgeOS
  - NetworkManager
  - …

Simple Composable Tools: `wg-quick`

- Simple shell script

-
  ```
  # wg-quick up vpn0
  # wg-quick down vpn0
  ```
- /etc/wireguard/vpn0.conf:

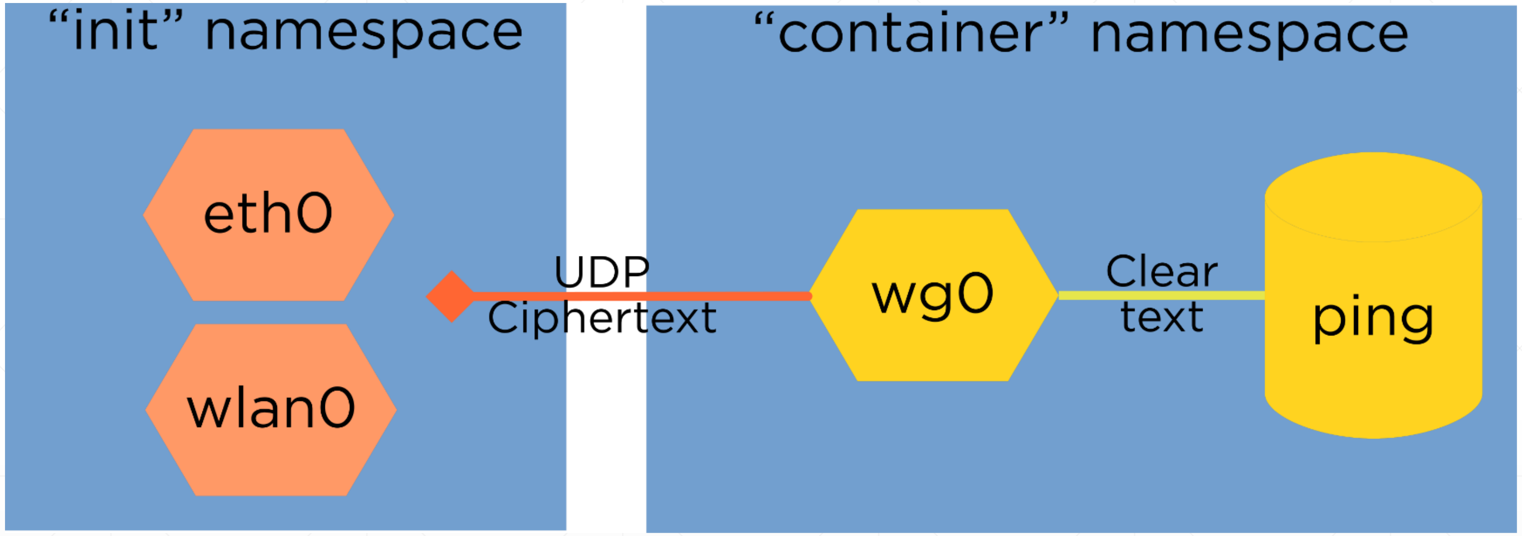
  ```
  [Interface]
  Address = 10.200.100.2
  DNS = 10.200.100.1
  PostDown = resolvconf -d %i
  PrivateKey = uDmW0qECQZWPv4K83yg26b3L4r93HvLRcal997IGlEE=

  [Peer]
  PublicKey = +LRS63OXvyCoVDs1zmWRO/6gVkfQ/pTKEZvZ+CehO1E=
  AllowedIPs = 0.0.0.0/0
  Endpoint = demo.wireguard.io:51820
  ```

Network Namespace Tricks

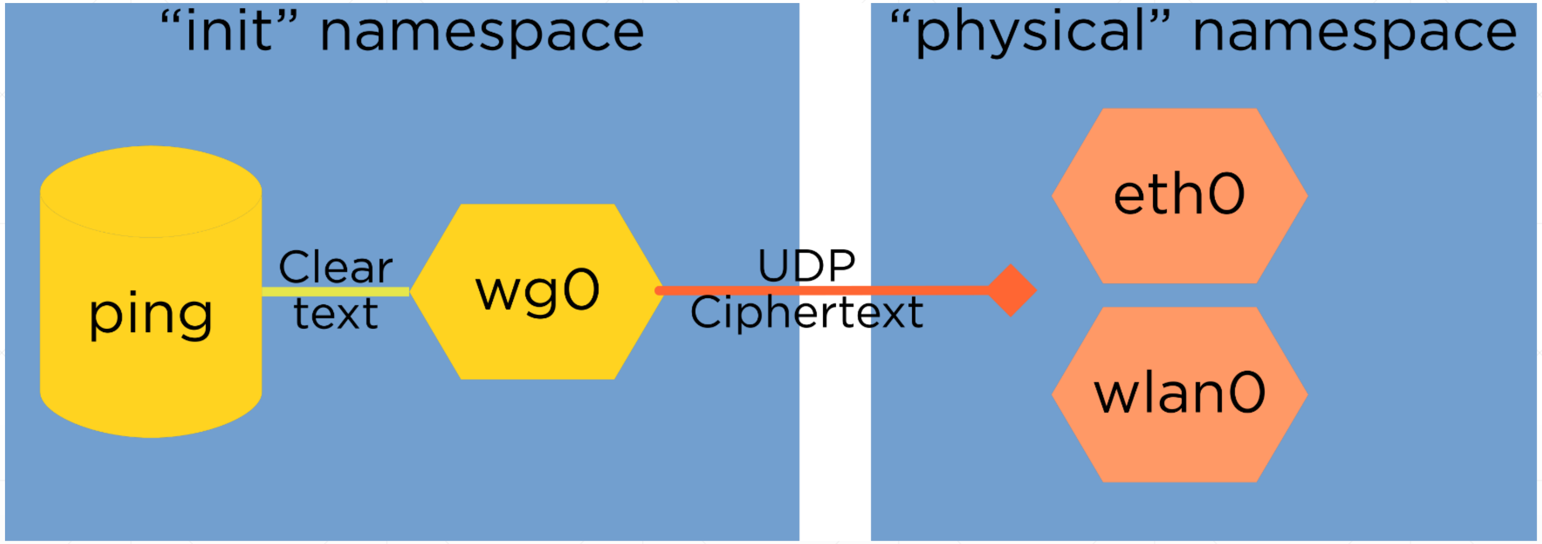
- The WireGuard interface can live in one namespace, and the physical interface can live in another.
- Only let a Docker container connect via WireGuard.
- Only let your DHCP client touch physical interfaces, and only let your web browser see WireGuard interfaces.
- Nice alternative to routing table hacks.

Namespaces: Containers



```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP>
    inet 127.0.0.1/8 scope host lo
17: wg0: <NOARP,UP,LOWER_UP>
    inet 192.168.4.33/32 scope global wg0
```

Namespaces: Personal VPN



```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP>
    inet 127.0.0.1/8 scope host lo
17: wg0: <NOARP,UP,LOWER_UP>
    inet 192.168.4.33/32 scope global wg0
```

Security Design Principle 3: Static Fixed Length Headers

- All packet headers have fixed width fields, so no parsing is necessary.
  - Eliminates an entire class of vulnerabilities.
  - No parsers → no parser vulnerabilities.
- Quite a different approach to formats like ASN.1/X.509 or even variable length IP and TCP packet headers.

Security Design Principle 4: Static Allocations and Guarded State

- All state required for WireGuard to work is allocated during config.
- No memory is dynamically allocated in response to received packets.
  - Eliminates *another* entire classes of vulnerabilities.
  - Places an unusual constraint on the crypto, since we are operating over a finite amount of preallocated memory.
- No state is modified in response to unauthenticated packets.
  - Eliminates *yet another* entire class of vulnerabilities.
  - Also places unusual constraints on the crypto.

Security Design Principle 5: Stealth

- Some aspects of WireGuard grew out of a kernel rootkit project.
- Should not respond to any unauthenticated packets.
- Hinder scanners and service discovery.
- Service only responds to packets with correct crypto.
- Not chatty at all.
    - When there's no data to be exchanged, both peers become silent.

Security Design Principle 6: Solid Crypto

- We make use of Noise Protocol Framework – noiseprotocol.org
  - WireGuard was involved early on with the design of Noise, ensuring it could do what we needed.
  - Custom written very specific implementation of Noise_IKpsk2 for the kernel.
  - Related in spirit to the Signal Protocol.
- The usual list of modern desirable properties you'd want from an authenticated key exchange
- Modern primitives: Curve25519, Blake2s, ChaCha20, Poly1305
- Lack of cipher agility! (Opinionated.)

Security Design Principle 6: Solid Crypto

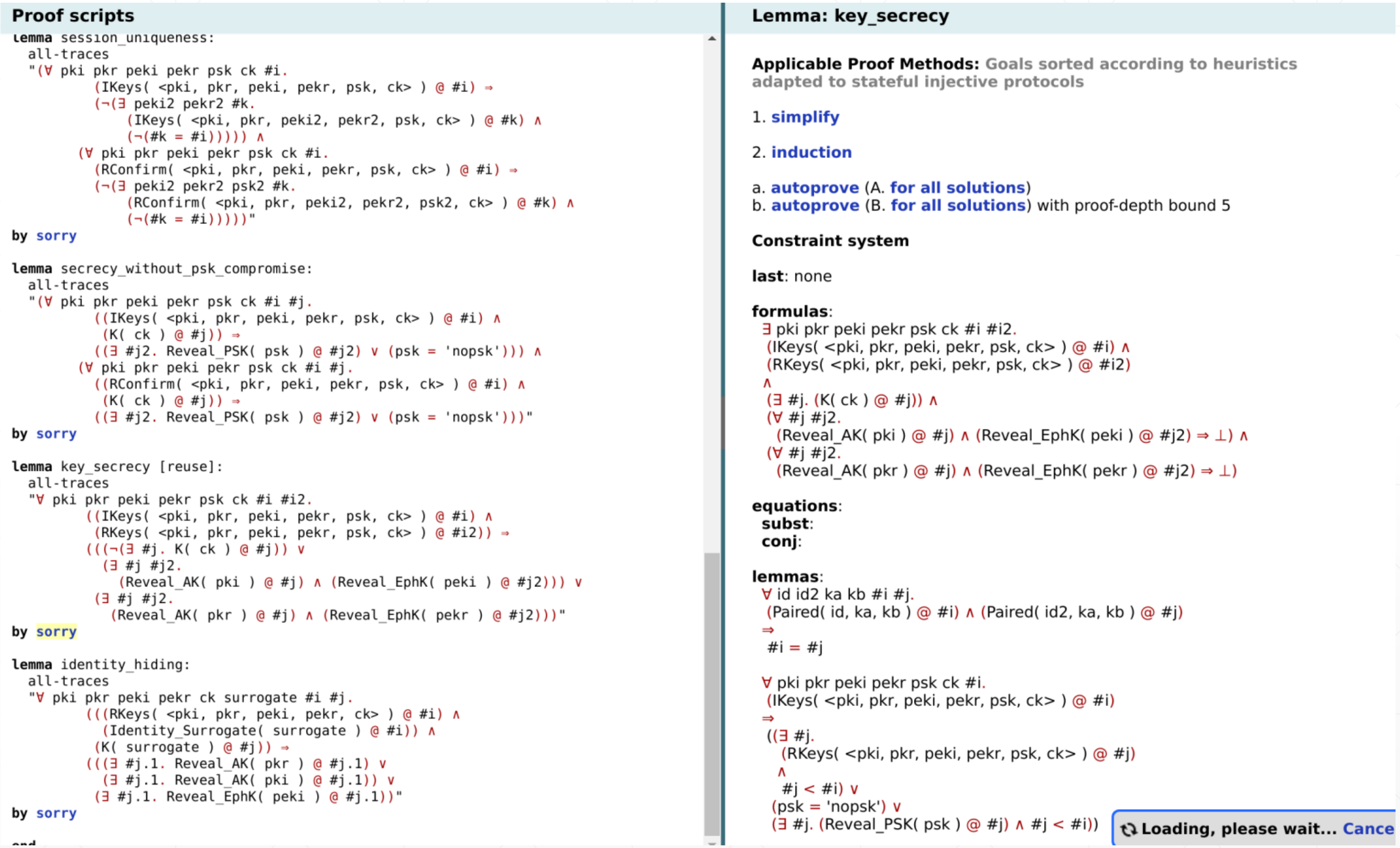
- Strong key agreement & authenticity
- Key-compromise impersonation resistance
- Unknown key-share attack resistance
- Key secrecy
- Forward secrecy
- Session uniqueness
- Identity hiding
- Replay-attack prevention, while allowing for network packet reordering

Crypto Designed for Kernel

- Design goals of guarded memory safety, few allocations, etc have direct effect on cryptography used.
  - Ideally be 1-RTT.
- Fast crypto primitives.
- Clear division between slowpath for ECDH and fastpath for symmetric crypto.
- Handshake in kernel space, instead of punted to userspace daemon like IKE/IPsec.
  - Allows for more efficient and less complex protocols.
  - Exploit interactions between handshake state and packet encryption state.

Formal Symbolic Verification

- The cryptographic protocol has been formally verified using Tamarin.



Proof scripts

```
lemma session_uniqueness:
  all-traces
  "(∀ pki pkr peki pekr psk ck #i.
          (IKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i) ⇒
          (¬(∃ peki2 pekr2 #k.
              (IKeys( <pki, pkr, peki2, pekr2, psk, ck> ) @ #k) ∧
              (¬(#k = #i))))) ∧
        (∀ pki pkr peki pekr psk ck #i.
          (RConfirm( <pki, pkr, peki, pekr, psk, ck> ) @ #i) ⇒
          (¬(∃ peki2 pekr2 psk2 #k.
              (RConfirm( <pki, pkr, peki2, pekr2, psk2, ck> ) @ #k) ∧
              (¬(#k = #i)))))"
by sorry

lemma secrecy_without_psk_compromise:
  all-traces
  "(∀ pki pkr peki pekr psk ck #i #j.
          ((IKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i) ∧
           (K( ck ) @ #j)) ⇒
          ((∃ #j2. Reveal_PSK( psk ) @ #j2) ∨ (psk = 'nopsk'))) ∧
        (∀ pki pkr peki pekr psk ck #i #j.
          ((RConfirm( <pki, pkr, peki, pekr, psk, ck> ) @ #i) ∧
           (K( ck ) @ #j)) ⇒
          ((∃ #j2. Reveal_PSK( psk ) @ #j2) ∨ (psk = 'nopsk')))"
by sorry

lemma key_secrecy [reuse]:
  all-traces
  "∀ pki pkr peki pekr psk ck #i #i2.
         ((IKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i) ∧
          (RKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i2)) ⇒
         (((¬(∃ #j. K( ck ) @ #j)) ∨
           (∃ #j #j2.
             (Reveal_AK( pki ) @ #j) ∧ (Reveal_EphK( peki ) @ #j2))) ∨
          (∃ #j #j2.
            (Reveal_AK( pkr ) @ #j) ∧ (Reveal_EphK( pekr ) @ #j2)))"
by sorry

lemma identity_hiding:
  all-traces
  "∀ pki pkr peki pekr ck surrogate #i #j.
         (((RKeys( <pki, pkr, peki, pekr, ck> ) @ #i) ∧
           (Identity_Surrogate( surrogate ) @ #i)) ∧
          (K( surrogate ) @ #j)) ⇒
         (((∃ #j.1. Reveal_AK( pkr ) @ #j.1) ∨
           (∃ #j.1. Reveal_AK( pki ) @ #j.1)) ∨
          (∃ #j.1. Reveal_EphK( peki ) @ #j.1))"
by sorry
```

Lemma: key_secrecy

Applicable Proof Methods: Goals sorted according to heuristics adapted to stateful injective protocols

1. simplify

2. induction

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5

Constraint system

last: none

```
formulas:
 ∃ pki pkr peki pekr psk ck #i #i2.
  (IKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i) ∧
  (RKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i2)
 ∧
  (∃ #j. (K( ck ) @ #j)) ∧
  (∀ #j #j2.
    (Reveal_AK( pki ) @ #j) ∧ (Reveal_EphK( peki ) @ #j2) ⇒ ⊥) ∧
  (∀ #j #j2.
    (Reveal_AK( pkr ) @ #j) ∧ (Reveal_EphK( pekr ) @ #j2) ⇒ ⊥)

equations:
 subst:
 conj:

lemmas:
 ∀ id id2 ka kb #i #j.
  (Paired( id, ka, kb ) @ #i) ∧ (Paired( id2, ka, kb ) @ #j)
 ⇒
  #i = #j

 ∀ pki pkr peki pekr psk ck #i.
  (IKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #i)
 ⇒
  ((∃ #j.
     (RKeys( <pki, pkr, peki, pekr, psk, ck> ) @ #j)
    ∧
     #j < #i) ∨
   (psk = 'nopsk') ∨
   (∃ #j. (Reveal_PSK( psk ) @ #j) ∧ #j < #i))
```

Multicore Cryptography

- Encryption and decryption of packets can be spread out to all cores in parallel.
- Nonce/sequence number checking, `netif_rx`, and transmission must be done in serial order.
- Requirement: fast for single flow traffic in addition to multiflow traffic.
  - Different from usual assumptions.

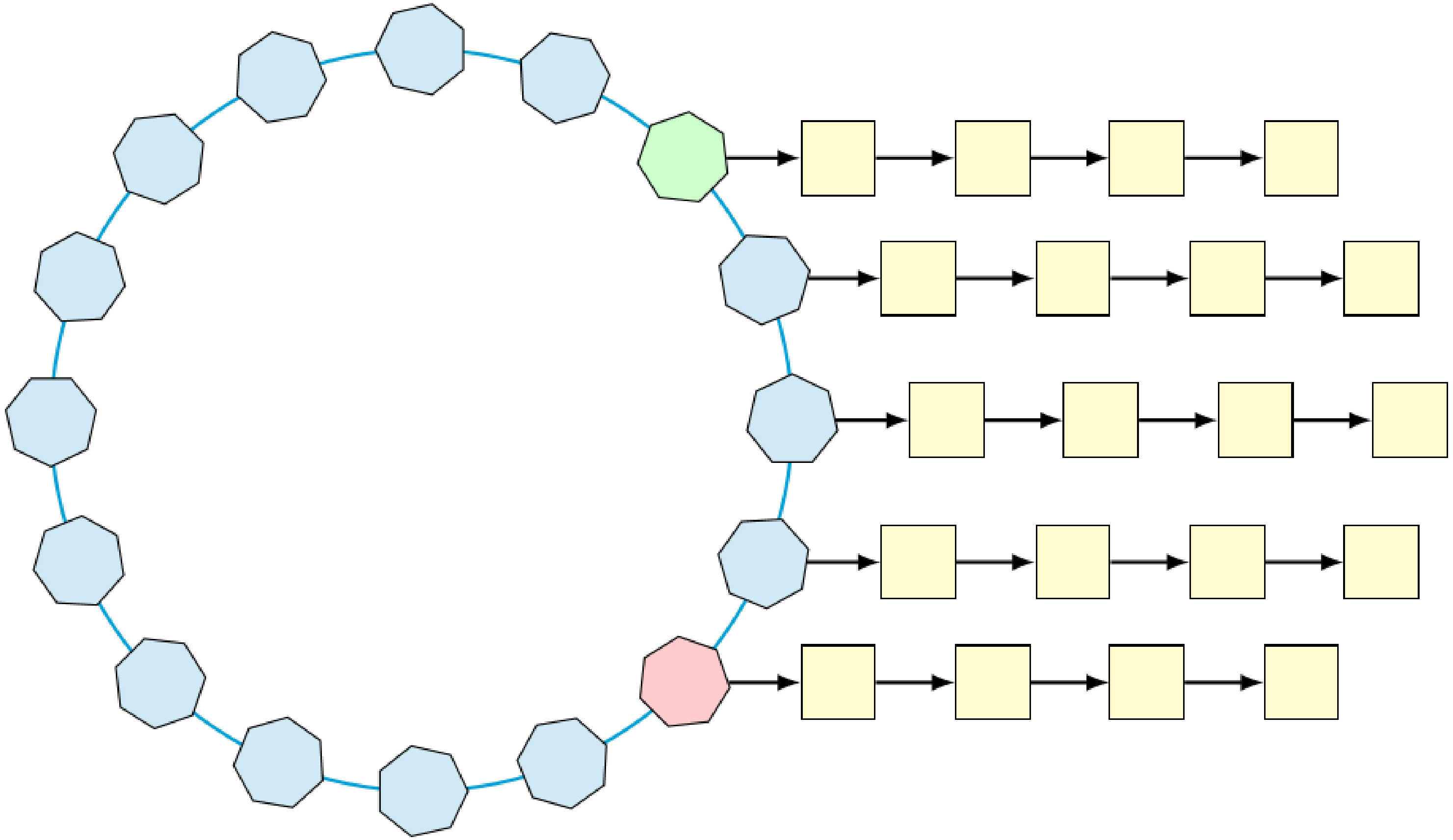
Multicore Cryptography

- Single queue, shared by all CPUs, rather than queue per CPU
    - No reliance on process scheduler, which tends to add latency when waiting for packets to complete
    - Serial transmission queue waits on ordered completion of parallel queue items
    - Using `netif_receive_skb` instead of `netif_rx` to push back on encryption queue
- Bunching bundles of packets together to be encrypted on one CPU results in high performance gains
    - How to choose the size of the bundle?

Generic Segmentation Offload

- By advertising that the `net_device` suppports GSO, WireGuard receives massive “super-packets” all at the same time.
- WireGuard can then split the super-packets by itself, and bundle these to be encrypted on a single CPU all at once.
- Each bundle is a linked list of skbs, which is added to the ring buffer queue.

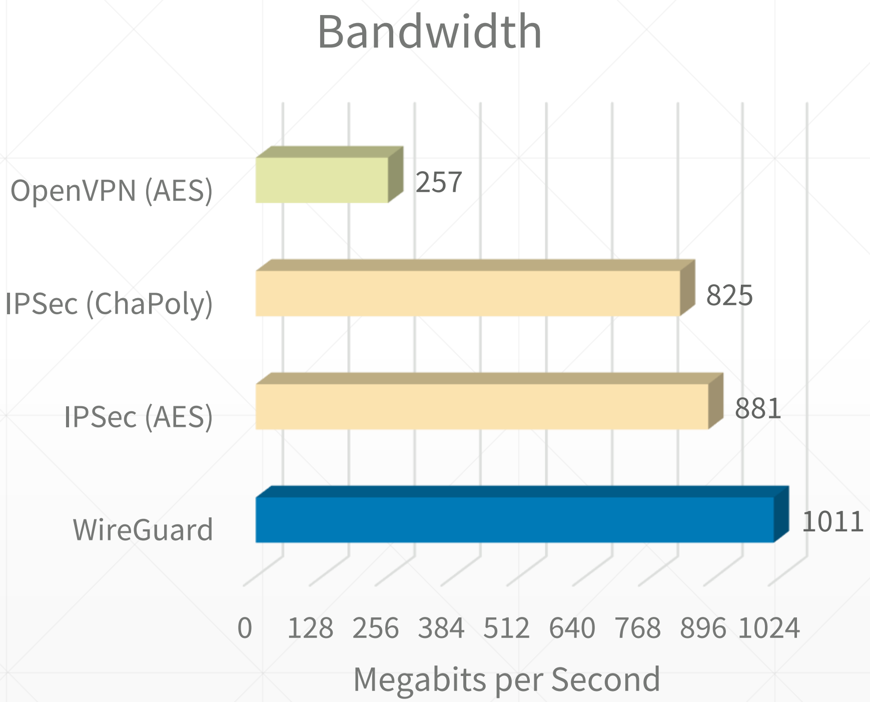
Multicore Cryptography

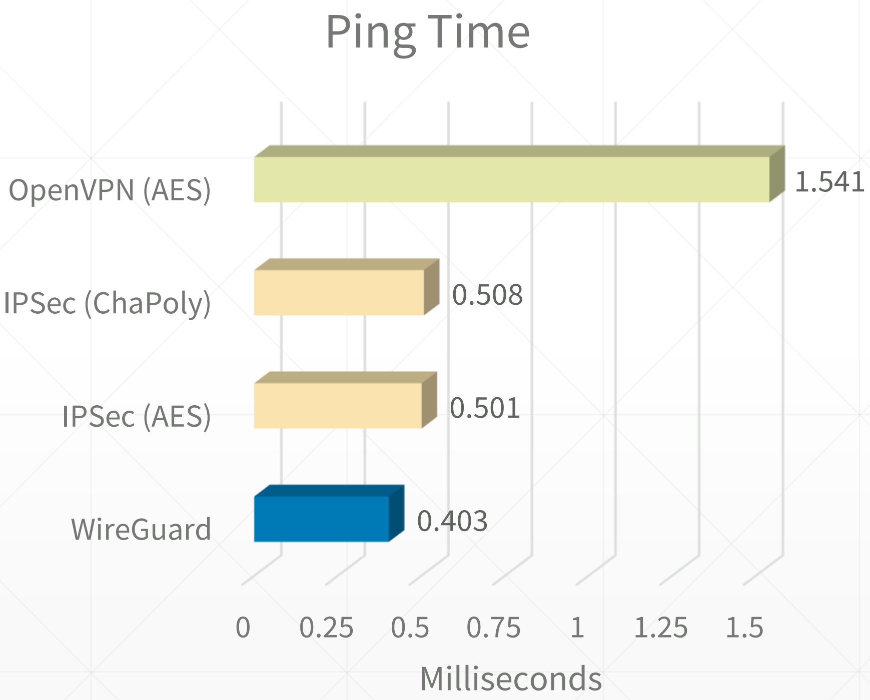


Performance

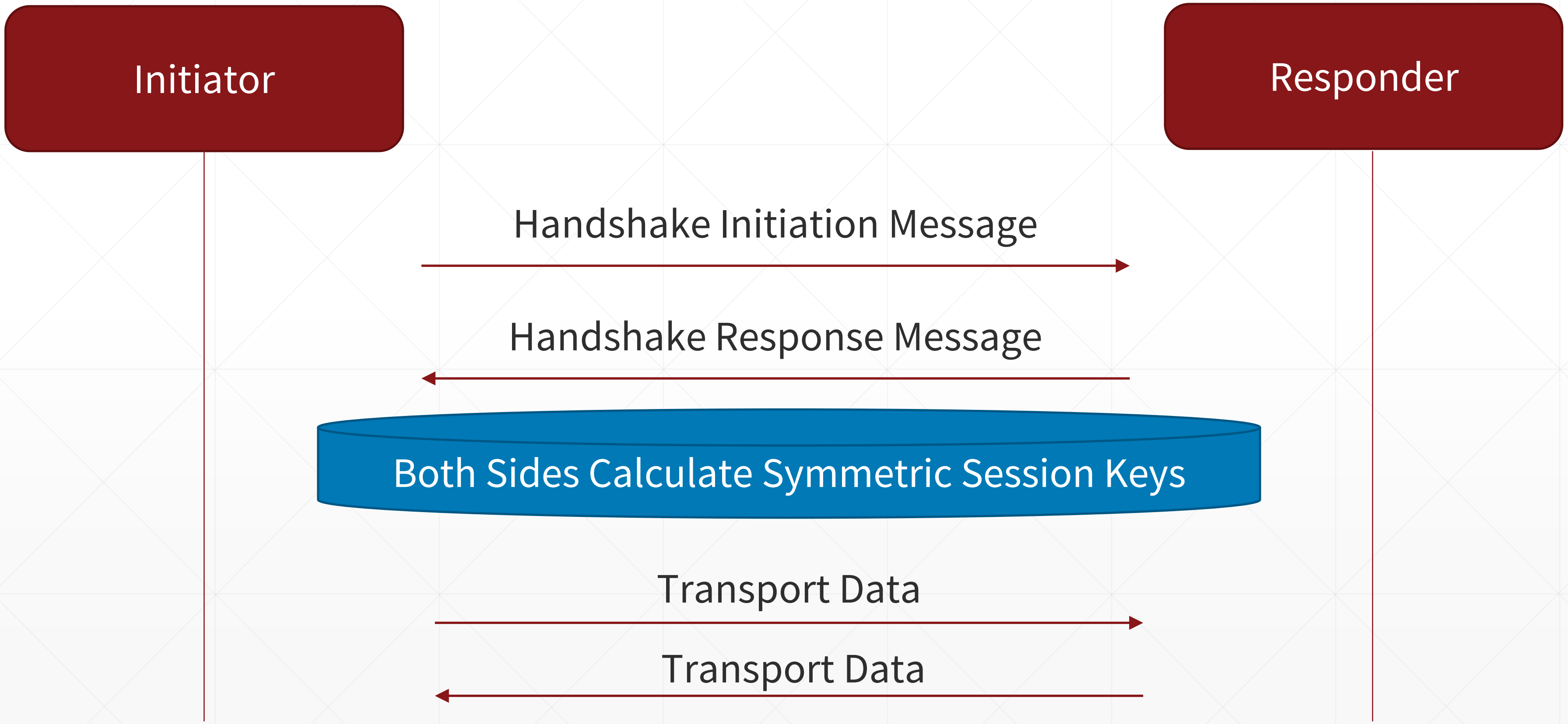
- Being in kernel space means that it is *fast* and low latency.
  - No need to copy packets twice between user space and kernel space.
- ChaCha20Poly1305 is extremely fast on nearly all hardware, and safe.
  - AES-NI is fast too, obviously, but as Intel and ARM vector instructions become wider and wider, ChaCha is handedly able to compete with AES-NI, and even perform better in some cases.
  - AES is exceedingly difficult to implement performantly and safely (no cache-timing attacks) without specialized hardware.
  - ChaCha20 can be implemented efficiently on nearly all general purpose processors.
- Simple design of WireGuard means less overhead, and thus better performance.
  - Less code → Faster program? Not always, but in this case, certainly.

Performance: Measurements





Confluence of Principles → The Key Exchange



Initiator

Responder

Handshake Initiation Message

Handshake Response Message

Both Sides Calculate Symmetric Session Keys

Transport Data

Transport Data

The Key Exchange

- The key exchange designed to keep our principles static allocations, guarded state, fixed length headers, and stealthiness.
- In order for two peers to exchange data, they must first derive ephemeral symmetric crypto session keys from their static public keys.
- Either side can reinitiate the handshake to derive new session keys.
  - So initiator and responder can “swap” roles.
- Invalid handshake messages are ignored, maintaining stealth.

The Key Exchange: (Elliptic Curve) Diffie-Hellman Review

```
private A = random()
public A = derive_public(private A)
```

```
private B = random()
public B = derive_public(private B)
```

`ECDH(private A, public B) == ECDH(private B, public A)`

The Key Exchange: NoiseIK

- One peer is the initiator; the other is the responder.
- Each peer has their static identity – their long term *static keypair*.
- For each new handshake, each peer generates an *ephemeral keypair*.
- The security properties we want are achieved by computing `ECDH()` on the combinations of two ephemeral keypairs and two static keypairs.

The Key Exchange: NoiseIK

Alice	**Bob**
Static Private	Static Public
Ephemeral Private	Ephemeral Public

The Key Exchange: NoiseIK

Bob	**Alice**
Static Private	Static Public
Ephemeral Private	Ephemeral Public

The Key Exchange: NoiseIK

- One peer is the initiator; the other is the responder.
- Each side has a static identity keypair and an ephemeral session keypair.
- Session keys = `Noise(`

```
ECDH(ephemeral, static),
ECDH(static, ephemeral),
ECDH(ephemeral, ephemeral),
ECDH(static, static)
)
```

- The first three `ECDH()` make up the “triple DH”, like in Signal, and the last one allows for authentication in the first message, for 1-RTT.

The Key Exchange: NoiseIK – Initiator → Responder

- The initiator begins by knowing the long term static public key of the responder.
- The initiator sends to the responder:
  - A cleartext ephemeral public key.
  - The initiator's public key, authenticated-encrypted using a key that is an (indirect) result of:

    `ECDH(Ei, Sr) == ECDH(Sr, Ei)`

    - After decrypting this, the responder knows the initiator's public key.
    - Only the responder can decrypt this, because it requires control of the responder's static private key.
    - No forward secrecy for identity hiding.
  - A monotonically increasing counter (usually just a timestamp in TAI64N) that is authenticated-encrypted using a key that is an (indirect) result of the above calculation as well as:

    `ECDH(Si, Sr) == ECDH(Sr, Si)`

    - This counter prevents against replay DoS.
    - Authenticating it verifies the initiator controls its private key.
    - Authentication in the first message – static-static `ECDH()`.

The Key Exchange: NoiseIK – Responder → Initiator

- The responder at this point has learned the initiator’s static public key from the prior first message, as well as the initiator’s ephemeral public key.
- The responder sends to the initiator:
  - A cleartext ephemeral public key.
  - An empty buffer, authenticated-encrypted using a key that is an (indirect) result of the calculations in the prior message as well as:

    `ECDH(Er, Ei) == ECDH(Ei, Er)`

    and

    `ECDH(Er, Si) == ECDH(Si, Er)`

    - Authenticating it verifies the responder controls its private key.

The Key Exchange: Session Derivation

- After the previous two messages (initiator → responder and responder → initiator), both initiator and responder have something bound to these `ECDH()` calculations:
  - `ECDH(Ei, Sr) == ECDH(Sr, Ei)`
  - `ECDH(Si, Sr) == ECDH(Sr, Si)`
  - `ECDH(Ei, Er) == ECDH(Er, Ei)`
  - `ECDH(Si, Er) == ECDH(Er, Si)`
- From this they can derive symmetric authenticated-encryption session keys – one for sending and one for receiving.
- When the initiator sends its first data message using these session keys, the responder receives *confirmation* that the initiator has understood its response message, and can then send data to the initiator.

The Key Exchange

- Just 1-RTT.
- *Extremely* simple to implement in practice, and doesn't lead to the type of complicated messes we see in OpenSSL and StrongSwan.
- No certificates, X.509, or ASN.1: both sides exchange very short (32 bytes) base64-encoded public keys, just as with SSH.

```
zx2c4@thinkpad WireGuard/src $ cloc noise.c
-----------------------------------------------
Language     blank          comment         code
-----------------------------------------------
C               87               39          441
-----------------------------------------------
```

Poor-man’s PQ Resistance

- Optionally, two peers can have a pre-shared key, which gets “mixed” into the handshake.
- Grover’s algorithm – 256-bit symmetric key, brute forced with 2^{128} complexity.
  - This speed-up is *optimal.*
- Pre-shared keys are easy to steal, especially when shared amongst lots of parties.
  - But simply augments the ordinary handshake, not replaces it.
- By the time adversary can decrypt past traffic, hopefully all those PSKs have been forgotten by various hard drives anyway.

Hybrid PQ Resistance

- Alternatively, do a post-quantum key exchange, *through*, the tunnel.
- PQ primitives not directly built-in because they are slow and new and likely to change.
- PSK design allows us to easily swap them in and out for experiments as we learn more.

Security Design Principle 7: Abuse Resistance

- Hashing and symmetric crypto is fast, but pubkey crypto is slow.
- We use Curve25519 for elliptic curve Diffie-Hellman (ECDH), which is one of the fastest curves, but still is slower than the network.
- Overwhelm a machine asking it to compute ECDH( ).
    - Vulnerability in OpenVPN!
- UDP makes this difficult.
- WireGuard uses “cookies” to solve this.

Cookies: TCP-like

- Dialog:
  - Initiator: Compute this ECDH( ).
  - Responder: Your magic word is “baby penguin”. Ask me again with the magic word.
  - Initiator: My magic word is “baby penguin”. Compute this ECDH( ).
- Proves IP ownership, but cannot rate limit IP address without storing state.
  - Violates security design principle, no dynamic allocations!
- Always responds to message.
  - Violates security design principle, stealth!
- Magic word can be intercepted.

Cookies: DTLS-like and IKEv2-like

- Dialog:
  - Initiator: Compute this `ECDH().`
  - Responder: Your magic word is "`cbdd7c…bb71d9c0`". Ask me again with the magic word.
  - Initiator: My magic word is "`cbdd7c…bb71d9c0`". Compute this `ECDH().`
- `"cbdd7c…bb71d9c0" == MAC(responder_secret, initator_ip_address)`

  Where `responder_secret` changes every few minutes.
- Proves IP ownership without storing state.
- Always responds to message.
  - Violates security design principle, stealth!
- Magic word can be intercepted.
- Initiator can be DoS'd by flooding it with fake magic words.

Cookies: HIPv2-like and Bitcoin-like

- Dialog:
  - Initiator: Compute this `ECDH()`.
  - Responder: Mine a Bitcoin first, then ask me!
  - Initiator: I toiled away and found a Bitcoin. Compute this `ECDH()`.
- Proof of work.
- Robust for combating DoS if the puzzle is harder than `ECDH()`.
- However, it means that a responder can DoS an initiator, and that initiator and responder cannot symmetrically change roles without incurring CPU overhead.
  - Imagine a server having to do proofs of work for each of its clients.

Cookies: The WireGuard Variant

- Each handshake message (initiation and response) has two macs: `mac1` and `mac2`.
- `mac1` is calculated as:
  `HASH(responder_public_key || handshake_message)`
  - If this mac is invalid or missing, the message will be ignored.
  - Ensures that initiator must know the identity key of the responder in order to elicit a response.
    - Ensures stealthiness – security design principle.
- If the responder is not under load (not under DoS attack), it proceeds normally.
- If the responder is under load (experiencing a DoS attack), …

Cookies: The WireGuard Variant

- If the responder is under load (experiencing a DoS attack), it replies with a cookie computed as:

```
XAEAD(
  key=HASH(responder_public_key),
  additional_data=handshake_message,
  MAC(key: responder_secret, initiator_ip_address)
)
```

- `mac2` is then calculated as:
  `MAC(key: cookie, handshake_message)`
  - If it's valid, the message is processed even under load.

Cookies: The WireGuard Variant

- Once IP address is attributed, ordinary token bucket rate limiting can be applied.
- Maintains stealthiness.
- Cookies cannot be intercepted by somebody who couldn’t already initiate the same exchange.
- Initiator cannot be DoS’d, since the encrypted cookie uses the original handshake message as the “additional data” parameter.
  - An attacker would have to already have a MITM position, which would make DoS achievable by other means, anyway.

Fast, Modern, Secure

- **Less than 4,000 lines of code.**
- Easily implemented with basic data structures.
- Design of WireGuard lends itself to coding patterns that are secure in practice.
- Minimal state kept, no dynamic allocations.
- Stealthy and minimal attack surface.
- Handshake based on NoiseIK
- Fundamental property of a secure tunnel: association between a peer and a peer’s IPs.
- Extremely performant – best in class.
- Simple standard interface via an ordinary network device.
- Opinionated.

Fast, Modern, Secure

- Available now for all major Linux distros, FreeBSD, OpenBSD, macOS, iOS, and Android, Windows on its way: wireguard.com/install
- Paper published in NDSS 2017, available at: wireguard.com/papers/wireguard.pdf
-
  ```
  $ git clone https://git.zx2c4.com/WireGuard
  ```
- wireguard@lists.zx2c4.com
  lists.zx2c4.com/mailman/listinfo/wireguard
- `#wireguard` on Freenode
- **STICKERS FOR EVERYBODY**
- Plenty of work to be done: looking for interested devs.

Jason Donenfeld

- Personal website: www.zx2c4.com
- Email: Jason@zx2c4.com

ProtonMail

A critical analysis.

2.3b

What is ProtonMail?

Webmail provider, similar user experience to Gmail…

- *“All emails are secured automatically with end-to-end encryption. This means even we cannot decrypt and read your emails.”*
- Browser as well as mobile applications (iOS, Android.)
- Over 5 million users.

Secure Email Based in Switzerland

Secure Your Communications with ProtonMail

What are ProtonMail's security claims?

- *"All emails are secured automatically with end-to-end encryption. This means even we cannot decrypt and read your emails."*
- *"ProtonMail conservatively assumes that all mail servers may eventually be compromised. Thus, ProtonMail uses end-to-end encryption to ensure that plaintext email data is never sent to the server. If a server only contains encrypted messages, then the risks of a central server breach are mitigated."*

Secure Email Based in Switzerland

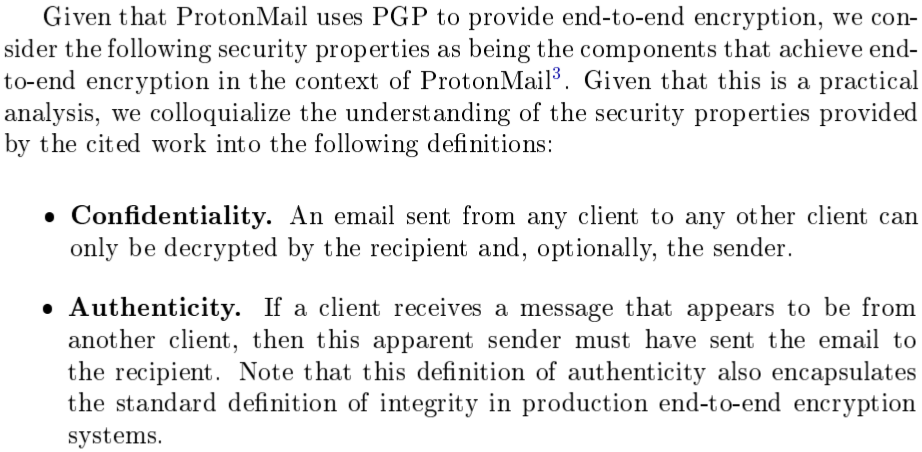
Secure Your Communications with ProtonMail

Central claim: end-to-end encryption.

But what *is* end-to-end encryption?

- End-to-end encryption has been defined as confidentiality, integrity and authentication between two parties by PGP (which ProtonMail uses for encryption), then by OTR, then by Signal and others.

- I decided to study whether ProtonMail achieves confidentiality and authenticity based on their own claims and definitions.



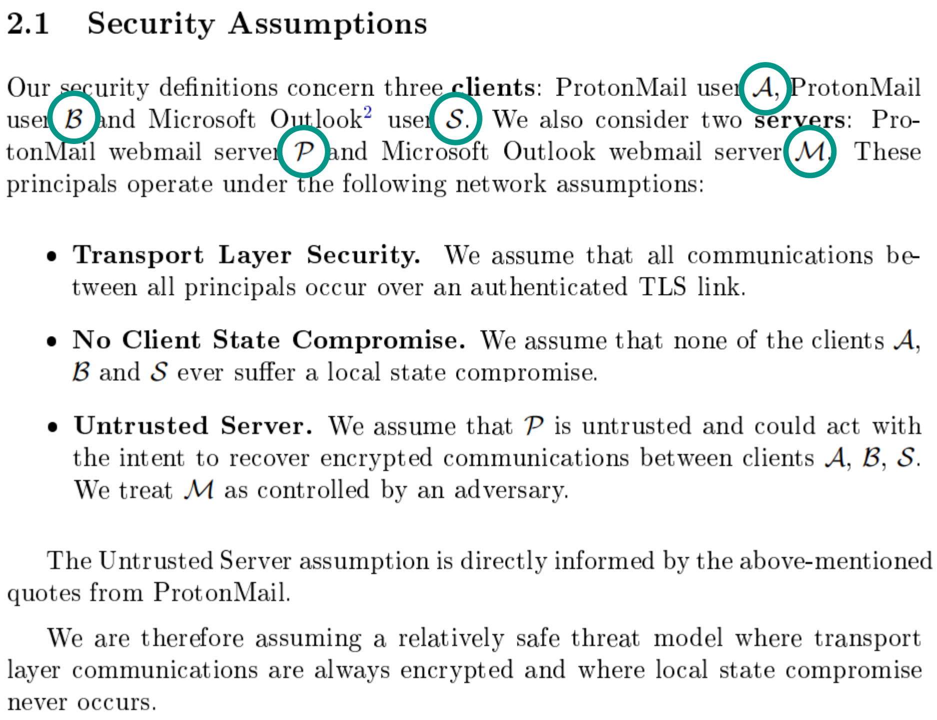
Given that ProtonMail uses PGP to provide end-to-end encryption, we consider the following security properties as being the components that achieve end-to-end encryption in the context of ProtonMail[3]. Given that this is a practical analysis, we colloquialize the understanding of the security properties provided by the cited work into the following definitions:

- **Confidentiality.** An email sent from any client to any other client can only be decrypted by the recipient and, optionally, the sender.
- **Authenticity.** If a client receives a message that appears to be from another client, then this apparent sender must have sent the email to the recipient. Note that this definition of authenticity also encapsulates the standard definition of integrity in production end-to-end encryption systems.

Examining ProtonMail protocol flows.

Two types of flows:

- *ProtonMail-to-ProtonMail:* A sends a PGP-encrypted message m to B through server P.
- *“Encrypt-to-Outside”*: A sends an encrypted email to S through P which relays the email to S through M.
    - S sends a PGP-encrypted reply r to A using the web interface J and A’s public key, both provided by P.



2.1 Security Assumptions

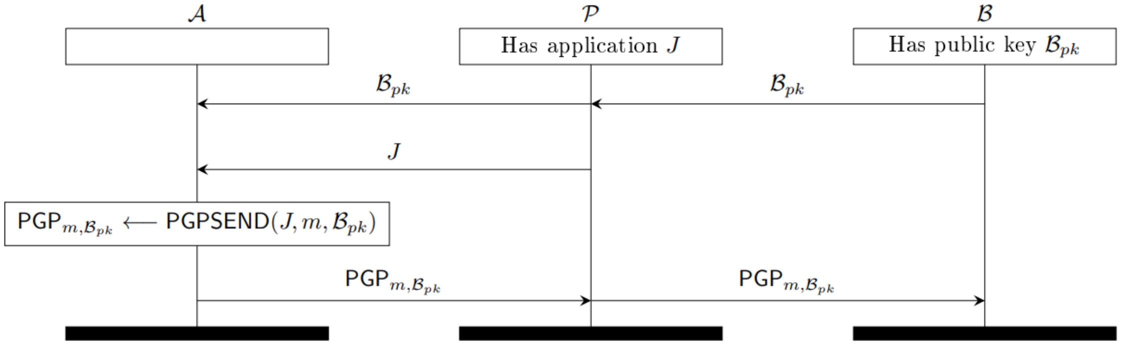
Our security definitions concern three **clients**: ProtonMail user $\mathcal{A}$, ProtonMail user $\mathcal{B}$ and Microsoft Outlook[2] user $\mathcal{S}$. We also consider two **servers**: ProtonMail webmail server $\mathcal{P}$ and Microsoft Outlook webmail server $\mathcal{M}$. These principals operate under the following network assumptions:

- **Transport Layer Security.** We assume that all communications between all principals occur over an authenticated TLS link.
- **No Client State Compromise.** We assume that none of the clients $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{S}$ ever suffer a local state compromise.
- **Untrusted Server.** We assume that $\mathcal{P}$ is untrusted and could act with the intent to recover encrypted communications between clients $\mathcal{A}$, $\mathcal{B}$, $\mathcal{S}$. We treat $\mathcal{M}$ as controlled by an adversary.

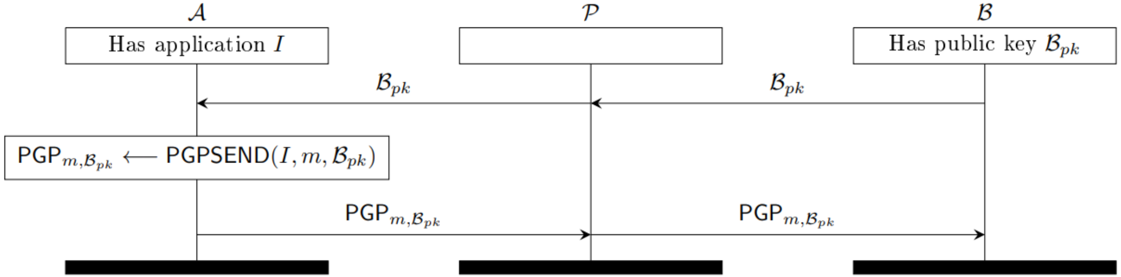
The Untrusted Server assumption is directly informed by the above-mentioned quotes from ProtonMail.

We are therefore assuming a relatively safe threat model where transport layer communications are always encrypted and where local state compromise never occurs.

Web application vs. smartphone.

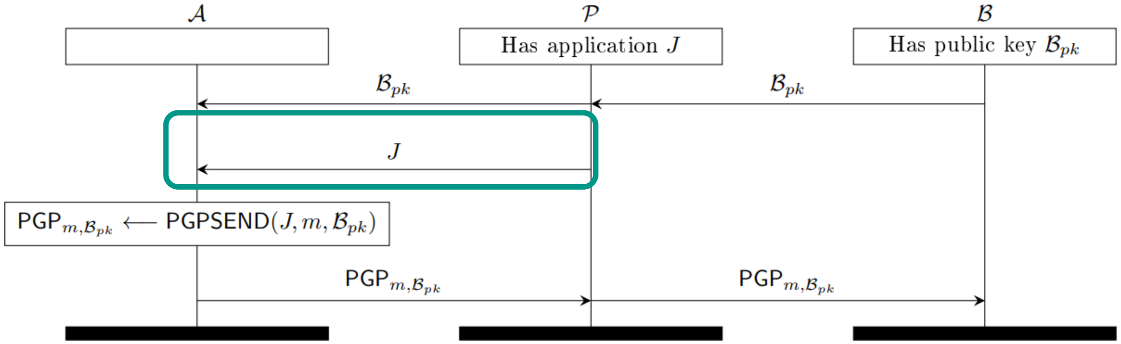


(a) $\mathcal{A}$ sends an email to $\mathcal{B}$ using the ProtonMail webmail application. We assume that $\mathcal{A}$ authenticates the fingerprint for PGP public key $\mathcal{B}_{pk}$ out of band.

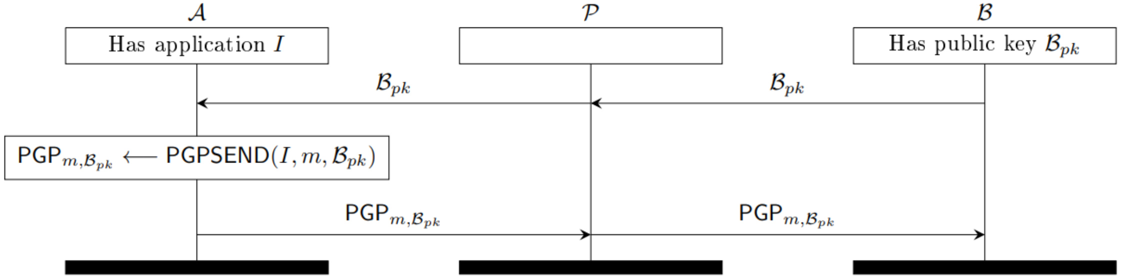


(b) $\mathcal{A}$ sends an email to $\mathcal{B}$ using the ProtonMail smartphone application. We assume that $\mathcal{A}$ authenticates the fingerprint for PGP public key $\mathcal{B}_{pk}$ out of band.

Web application vs. smartphone.



(a) $\mathcal{A}$ sends an email to $\mathcal{B}$ using the ProtonMail webmail application. We assume that $\mathcal{A}$ authenticates the fingerprint for PGP public key $\mathcal{B}_{pk}$ out of band.

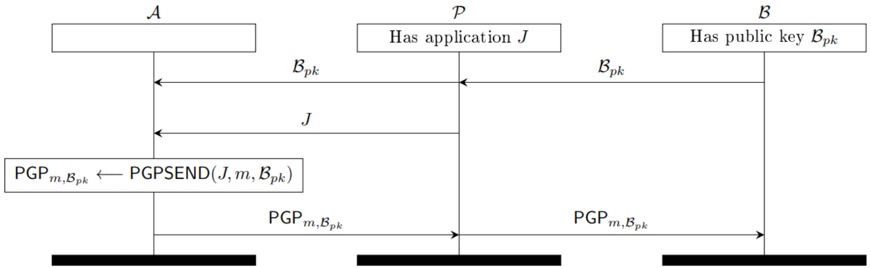


(b) $\mathcal{A}$ sends an email to $\mathcal{B}$ using the ProtonMail smartphone application. We assume that $\mathcal{A}$ authenticates the fingerprint for PGP public key $\mathcal{B}_{pk}$ out of band.

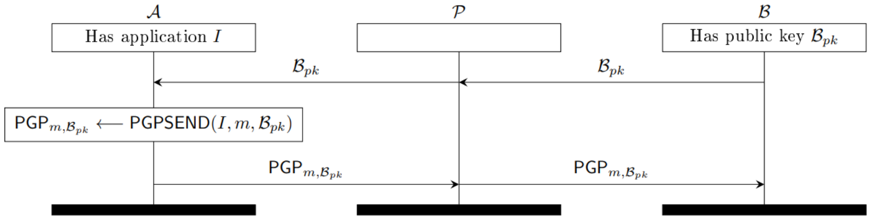
Code delivery: what's the difference?

Web-based versus code-signed downloads.

- *Web-based:* Code server unauthenticated, from scratch, at every request. If P swaps J out for malicious code, it's almost impossible to detect. P can also do this selectively.
- *Code-signed binaries:* Code signature from both the publisher and the app store. **Version numbers!** Traceable, runs locally and only updates with user consent.



(a) $\mathcal{A}$ sends an email to $\mathcal{B}$ using the ProtonMail webmail application. We assume that $\mathcal{A}$ authenticates the fingerprint for PGP public key $\mathcal{B}_{pk}$ out of band.



(b) $\mathcal{A}$ sends an email to $\mathcal{B}$ using the ProtonMail smartphone application. We assume that $\mathcal{A}$ authenticates the fingerprint for PGP public key $\mathcal{B}_{pk}$ out of band.

“Encrypt-to-Outside” feature.

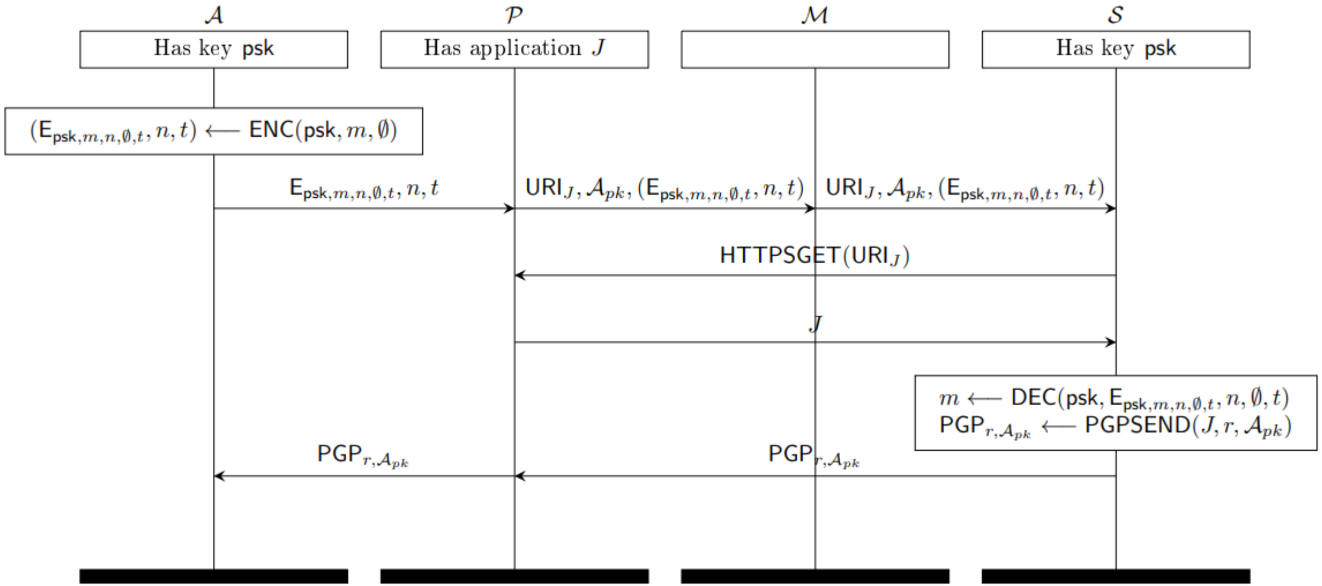


Figure 3: $\mathcal{A}$ sends an email containing message m to Microsoft Outlook user $\mathcal{S}$ symmetrically encrypted using a pre-shared key psk. $\mathcal{S}$ responds through the webmail interface provided by $\mathcal{P}$, encrypting his reply r using PGP to $\mathcal{A}_{pk}$.

“Encrypt-to-Outside” feature.

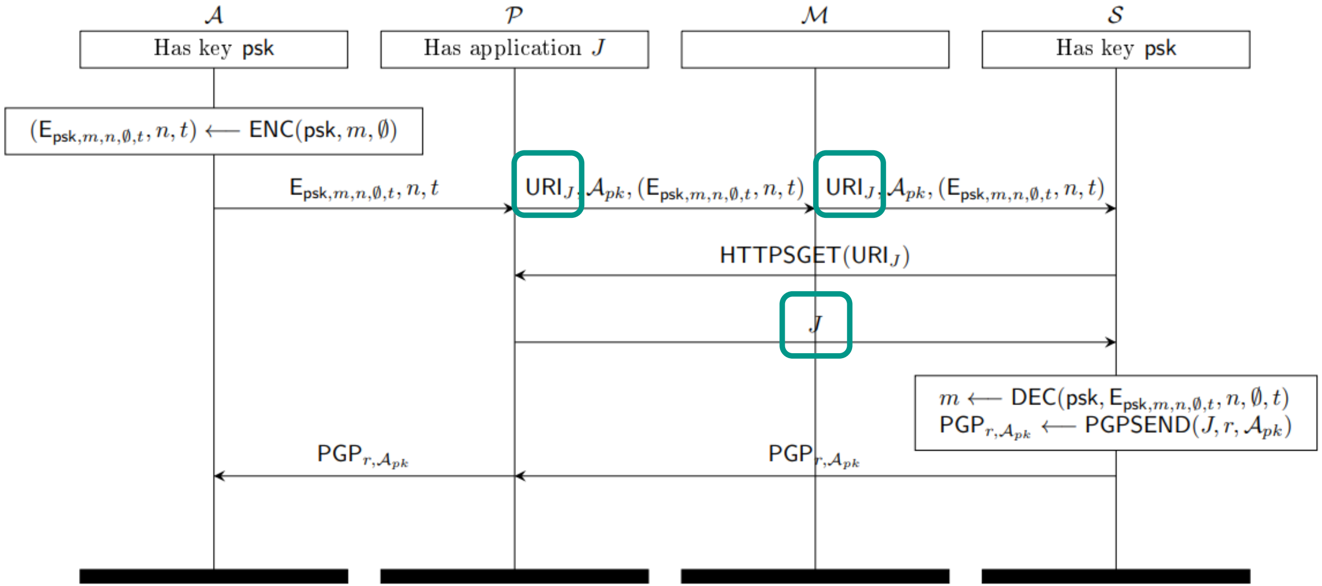


Figure 3: $\mathcal{A}$ sends an email containing message m to Microsoft Outlook user $\mathcal{S}$ symmetrically encrypted using a pre-shared key psk. $\mathcal{S}$ responds through the webmail interface provided by $\mathcal{P}$, encrypting his reply r using PGP to $\mathcal{A}_{pk}$.

“Encrypt-to-Outside” feature.

P can obtain psk and m by replacing J.

M can obtain psk, r, m and even encrypt a fake r to A.

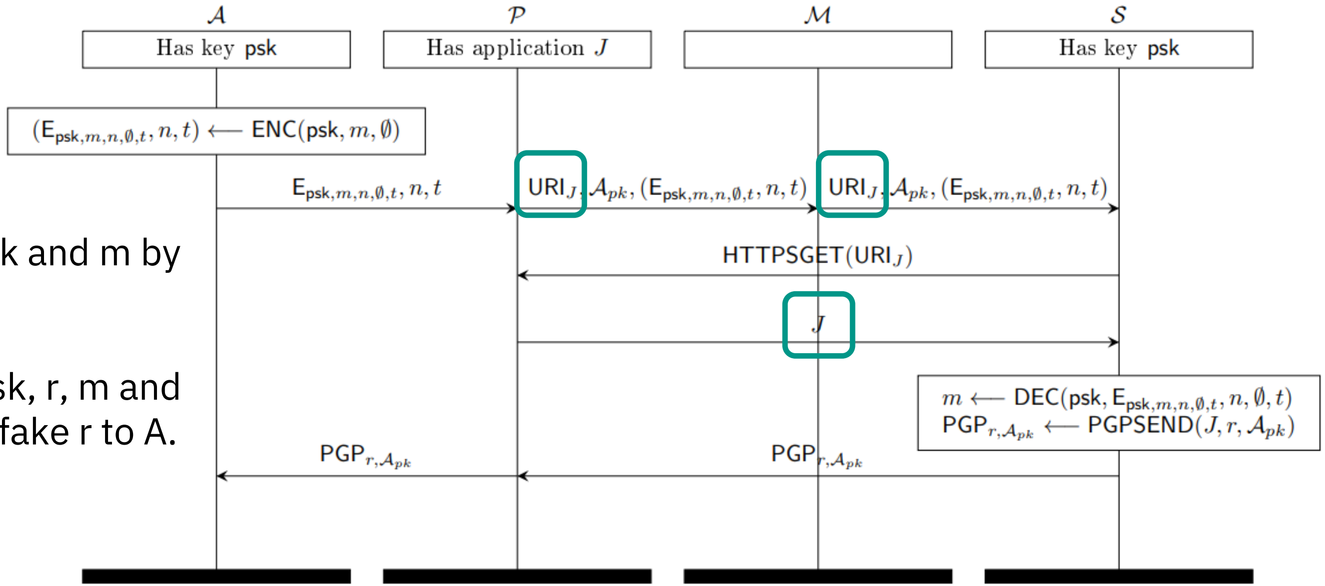
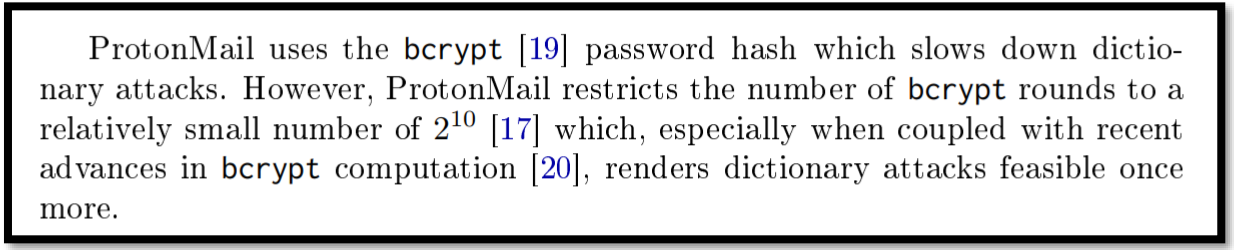
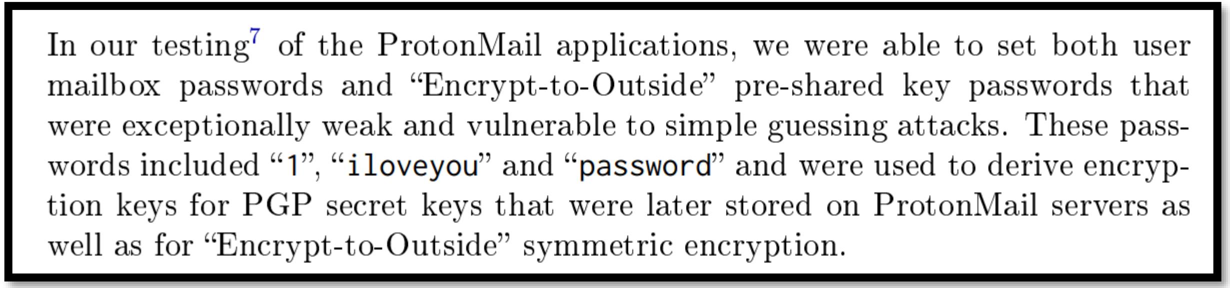


Figure 3: $\mathcal{A}$ sends an email containing message m to Microsoft Outlook user $\mathcal{S}$ symmetrically encrypted using a pre-shared key psk. $\mathcal{S}$ responds through the webmail interface provided by $\mathcal{P}$, encrypting his reply r using PGP to $\mathcal{A}_{pk}$.

Other issues found in ProtonMail.



ProtonMail uses the `bcrypt` [19] password hash which slows down dictionary attacks. However, ProtonMail restricts the number of `bcrypt` rounds to a relatively small number of 2^{10} [17] which, especially when coupled with recent advances in `bcrypt` computation [20], renders dictionary attacks feasible once more.



In our testing[7] of the ProtonMail applications, we were able to set both user mailbox passwords and "Encrypt-to-Outside" pre-shared key passwords that were exceptionally weak and vulnerable to simple guessing attacks. These passwords included "`1`", "`iloveyou`" and "`password`" and were used to derive encryption keys for PGP secret keys that were later stored on ProtonMail servers as well as for "Encrypt-to-Outside" symmetric encryption.

Fixed

- Fixed a bug where the SRP modulus signature was not verified by the web client. Reported by N. Kobeissi and S. Zanella.

ProtonMail's response?

- *Ad-hominem:* Personal attacks, smears, claiming I did my analysis for ulterior motives…
- *Misrepresentation of the findings:* "weak passwords are weak".
- *Side-stepping the issue:* Claim the security goals don't match their claims, despite being based on their own definitions.
- But the most damning quote of all:

"If the recipient's email service is the probable attacker in your threat model, then you probably shouldn't email them at that address" – i.e. we don't offer end-to-end encryption after all.

Response to analysis of ProtonMail's cryptographic architecture

Posted on January 20, 2019 by Admin

Next time: New Secure Protocols

2.4