WireGuard

FAST, MODERN, SECURE VPN TUNNEL

Presented by Jason A. Donenfeld

July 19, 2017

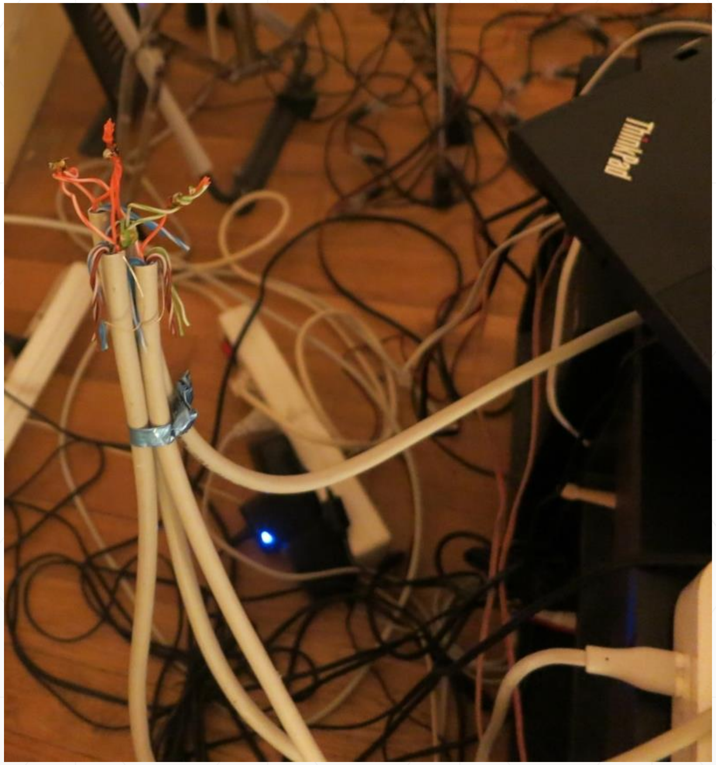
Security Interest Group

Who Am I?

- Jason Donenfeld, also known as **zx2c4**.
- Background in exploitation, kernel vulnerabilities, crypto vulnerabilities, though quite a bit of development experience too.
- Motivated to make a VPN that avoids the problems in both crypto and implementation that I've found in numerous other projects.

What is WireGuard?

- Layer 3 secure network tunnel for IPv4 and IPv6.
  - Opinionated.
- Lives in the Linux kernel, but cross platform implementations are in the works.
- 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.

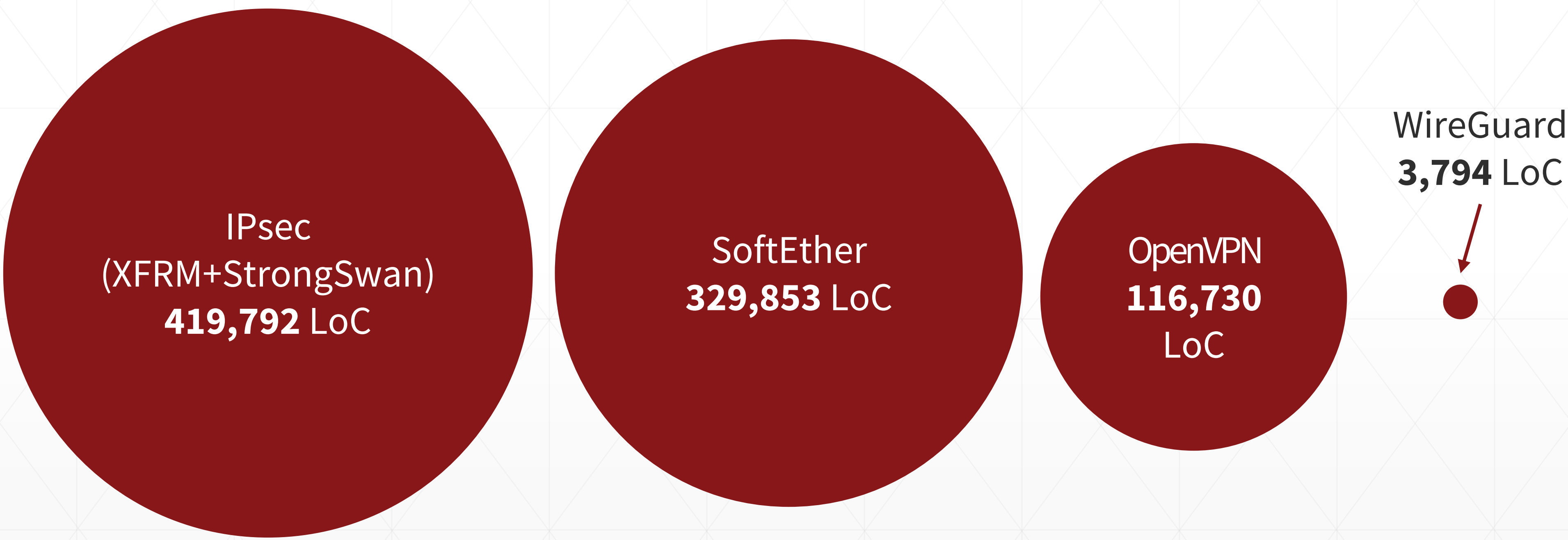


Easily Auditable

OpenVPN	Linux XFRM	StrongSwan	SoftEther	WireGuard
116,730 LoC Plus OpenSSL!	13,898 LoC Plus StrongSwan!	405,894 LoC Plus XFRM!	329,853 LoC	**3,794 LoC**

Less is more.

Easily Auditable



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.
  - 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.

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.
- 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.

Cryptokey Routing

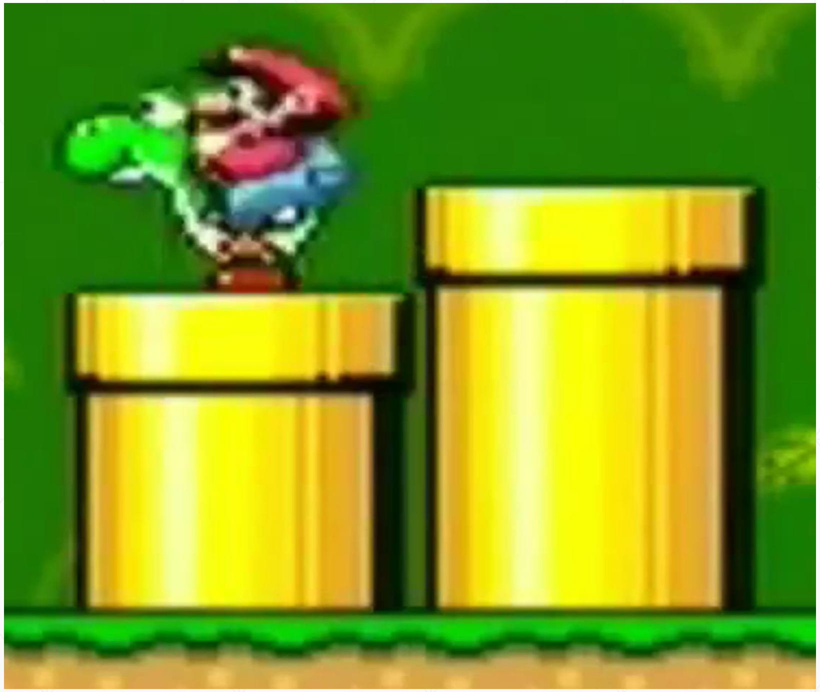
- **The fundamental concept of any VPN is an association between public keys of peers and the IP addresses that those peers are allowed to use.**
- A WireGuard interface has:
  - A private key
  - A listening UDP port
  - A list of peers
- A peer:
  - Is identified by its public key
  - Has a list of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

Cryptokey Routing

PUBLIC KEY :: IP ADDRESS

Cryptokey Routing

- 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.



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:
  - ifupdown
  - OpenWRT/LEDE
  - OpenRC netifrc
  - NixOS
  - systemd-networkd (WIP)
  - NetworkManager (WIP)

Simple Composable Tools: `wg-quick`

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

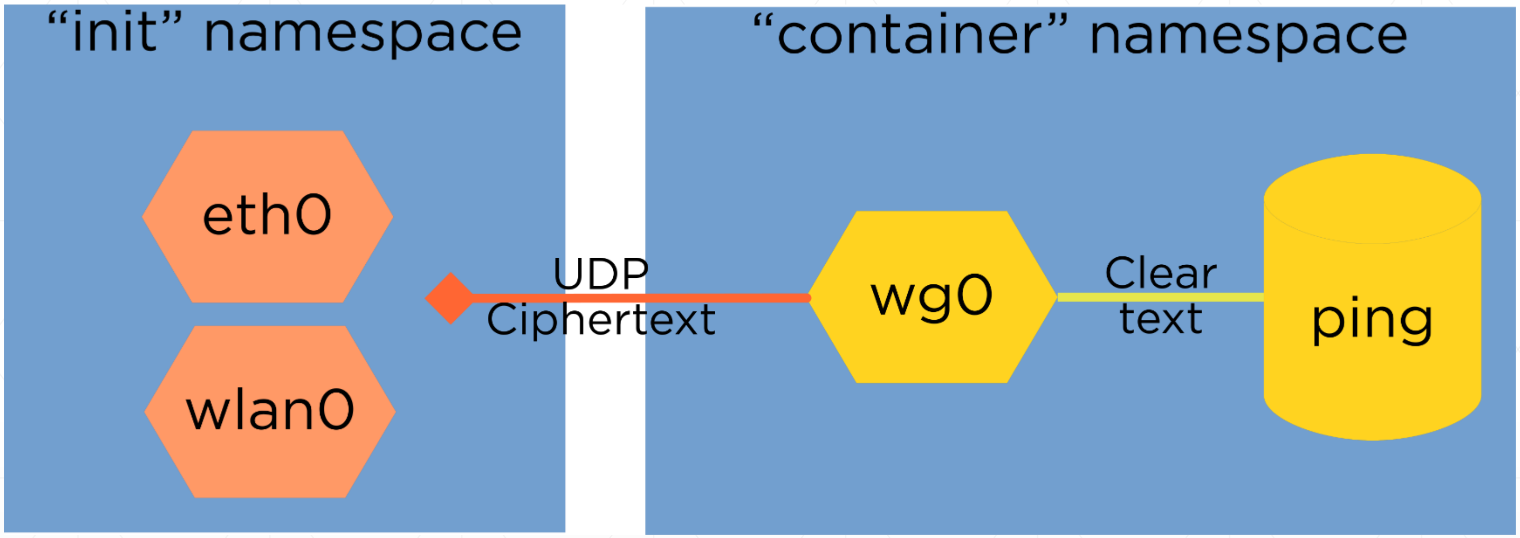
```
[Interface]
Address = 10.200.100.2
PostUp = echo nameserver 10.200.100.1 | resolvconf -a %i -m 0 -x
PostDown = resolvconf -d %i
PrivateKey = uDmW0qECQZWPv4K83yg26b3L4r93HvLRcal997IGlEE=

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

Network Namespace Tricks

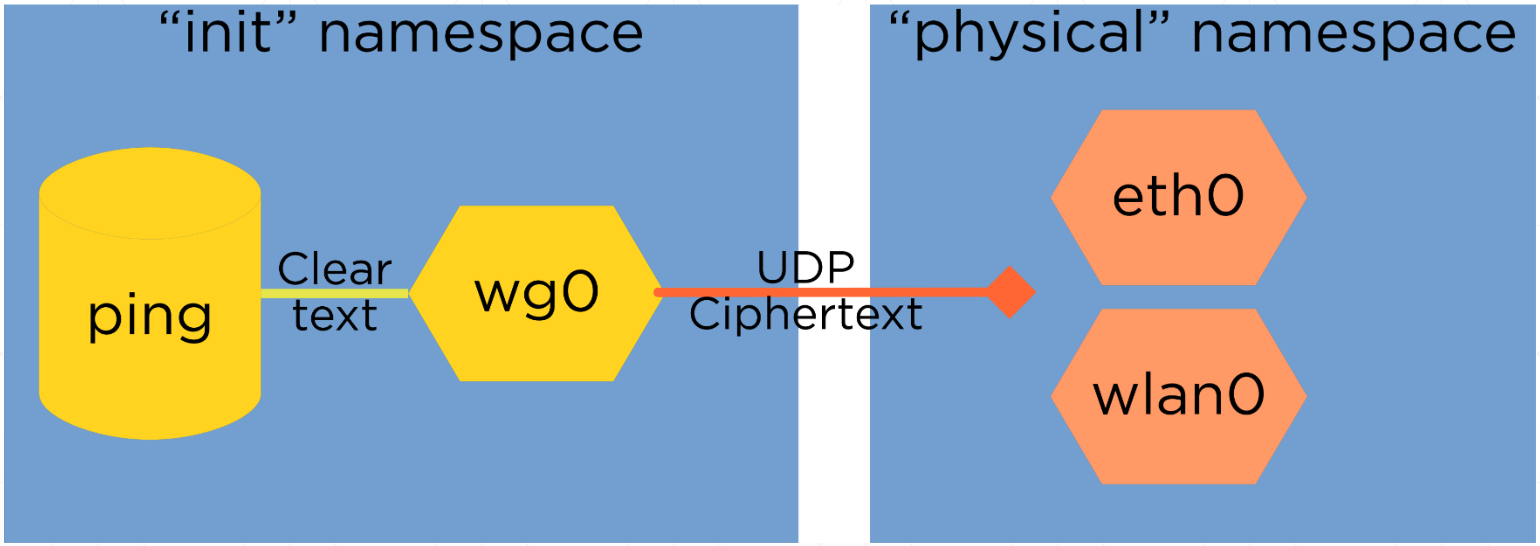
- The WireGuard interface can live in one namespace, and the physical interface can live in another.
- Only let a Docker container connect via WireGuard.
- Only let your DHCP client touch physical interfaces, and only let your web browser see WireGuard interfaces.
- Nice alternative to routing table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

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.

Static Allocations, Guarded State, and Fixed Length Headers

- All state required for WireGuard to work is allocated during config.
- No memory is dynamically allocated in response to received packets.
  - Eliminates entire classes of vulnerabilities.
- All packet headers have fixed width fields, so no parsing is necessary.
  - Eliminates *another* entire class of vulnerabilities.
- No state is modified in response to unauthenticated packets.
  - Eliminates *yet another* entire class of vulnerabilities.

Stealth

- Some aspects of WireGuard grew out of an earlier 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.

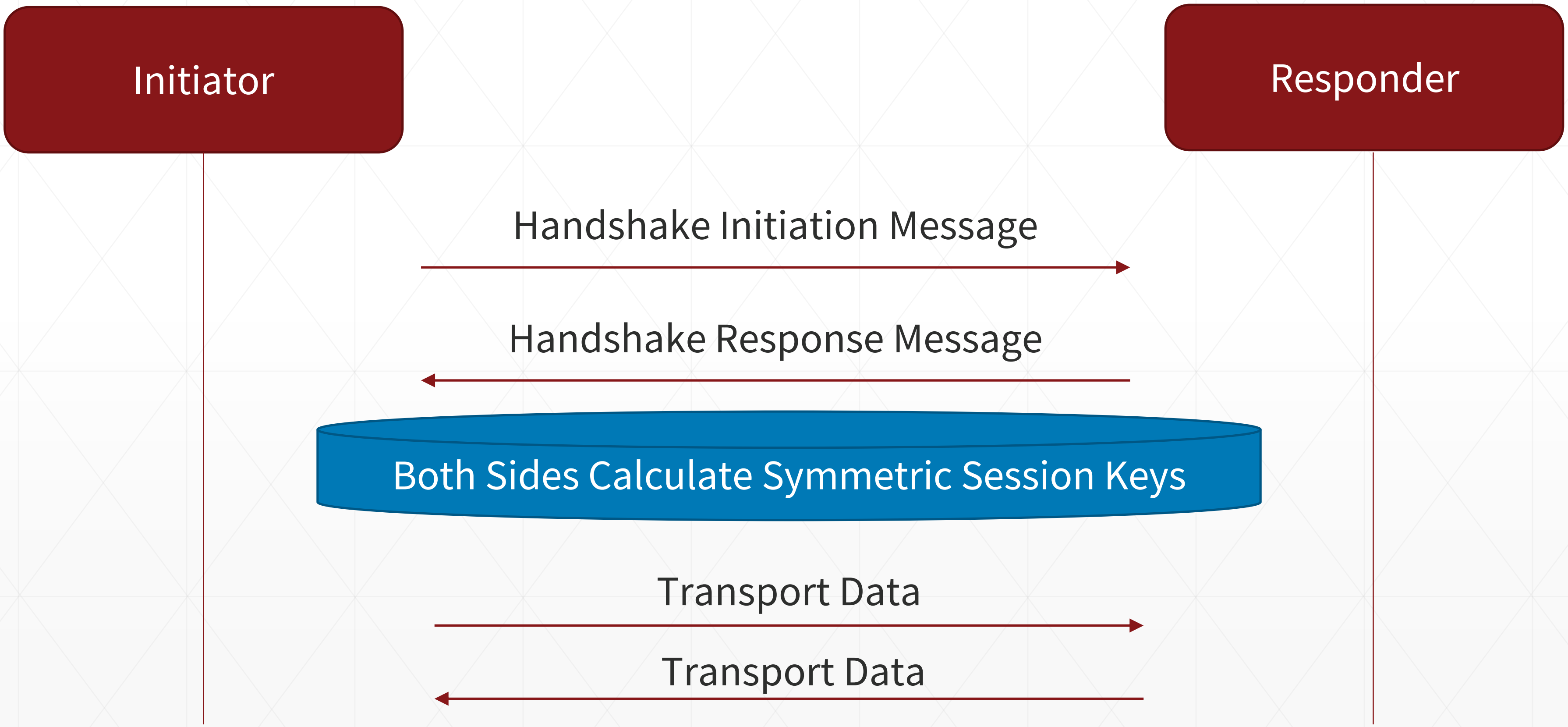
Crypto

- We make use of Trevor Perrin's Noise Protocol Framework – noiseprotocol.org
  - Developed with much feedback from the WireGuard development.
  - Custom written very specific implementation of NoiseIK for the kernel.
- The usual list of modern desirable properties you'd want from an authenticated key exchange
- Modern primitives: Curve25519, Blake2s, ChaCha20, Poly1305, SipHash2-4
- Lack of cipher agility!

Crypto

- Key secrecy
  - Perfect forward secrecy – new key every 2 minutes
- Key agreement
  - Authenticity
  - KCI-resistance
- Identity hiding
- Replay-attack prevention, while allowing for network packet reordering
- Working on formal verification via Tamarin with Kevin Milner

The Key Exchange



WireGuard

The Key Exchange

- In order for two peers to exchange data, they must first derive ephemeral symmetric crypto session keys from their static public keys.
- The key exchange designed to keep our principles static allocations, guarded state, fixed length headers, and stealthiness.
- 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: 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.
- Session keys = `Noise(`

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

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

Key Agreement and Correctness

- Key agreement is achieved even in multiple compromise situations:
  - Both ephemeral keys compromised
  - Initiator static compromised → Initiator still has key agreement with responder
    - (KCI resistance)
  - Responder static compromised → Responder still has key agreement with initiator
    - (KCI resistance)
  - Combinations of a static key and an ephemeral key compromised

Key Secrecy

- Dependent on key agreement.
- Key secrecy is achieved even in these compromise situations:
  - Both ephemeral keys compromised
  - Both static keys compromised
    - Implies forward secrecy
  - One static key and one ephemeral key

Session Uniqueness

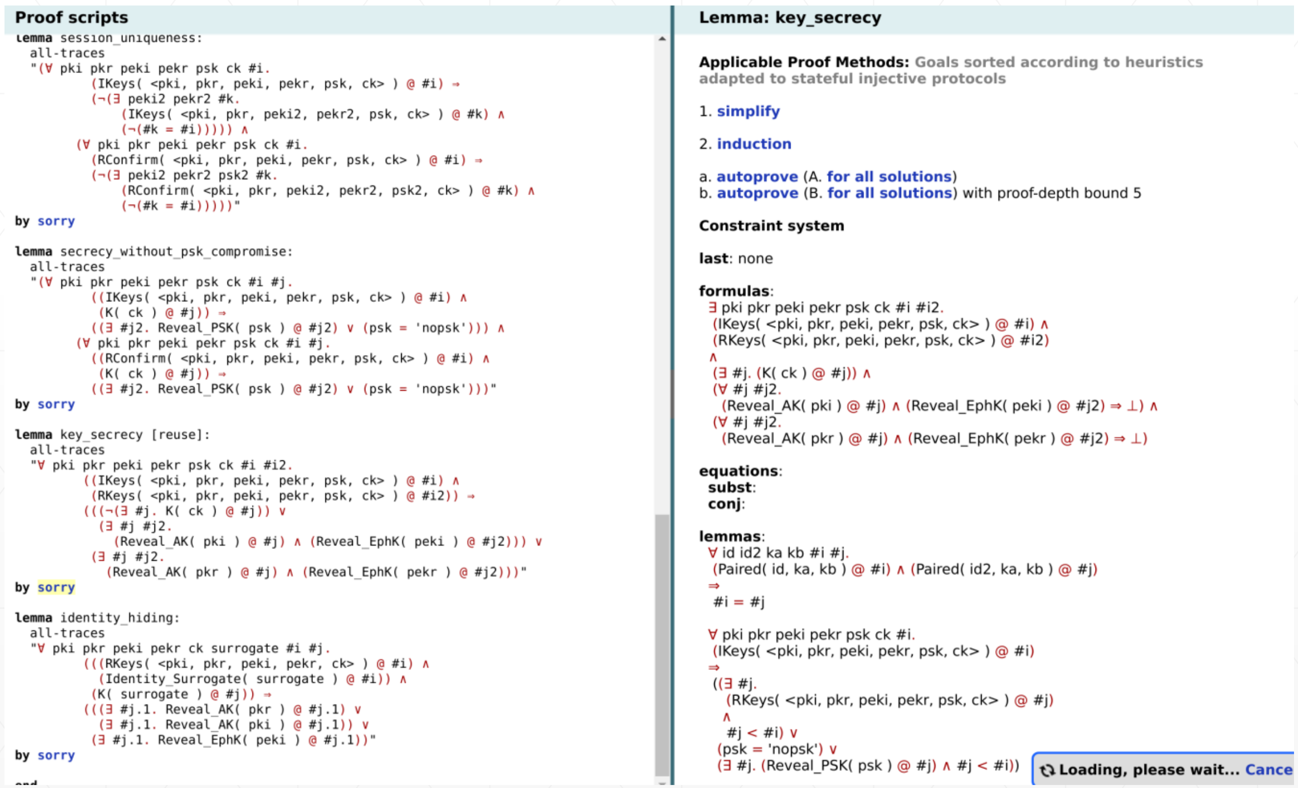
- Different sessions should always have different unique keys
- When both ephemerals are fresh, this is achieved
- Also, when only one ephemeral is fresh, it is achieved

Identity Hiding

- Initiator achieves identity hiding when no keys are compromised.
- Initiator also achieves identity hiding when the responder's ephemeral key is compromised.
- Initiator does *not* achieve identity hiding when the responder's static key is compromised.
  - Lack of forward secrecy for identity hiding
  - A necessity of a 1-RTT handshake

Formal Symbolic Verification

- Formally verified using Tamarin.



WireGuard

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.
  - 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} iterations.
  - 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.

Denial of Service 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 “latke”. Ask me again with the magic word.
  - Initiator: My magic word is “latke”. 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(key=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.

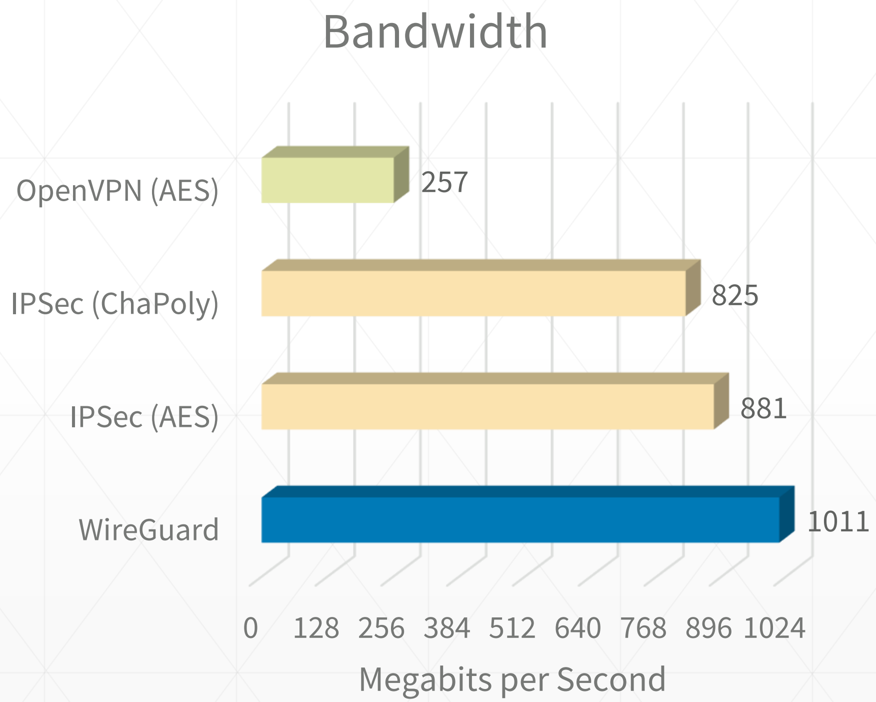
Cookies: The WireGuard Variant

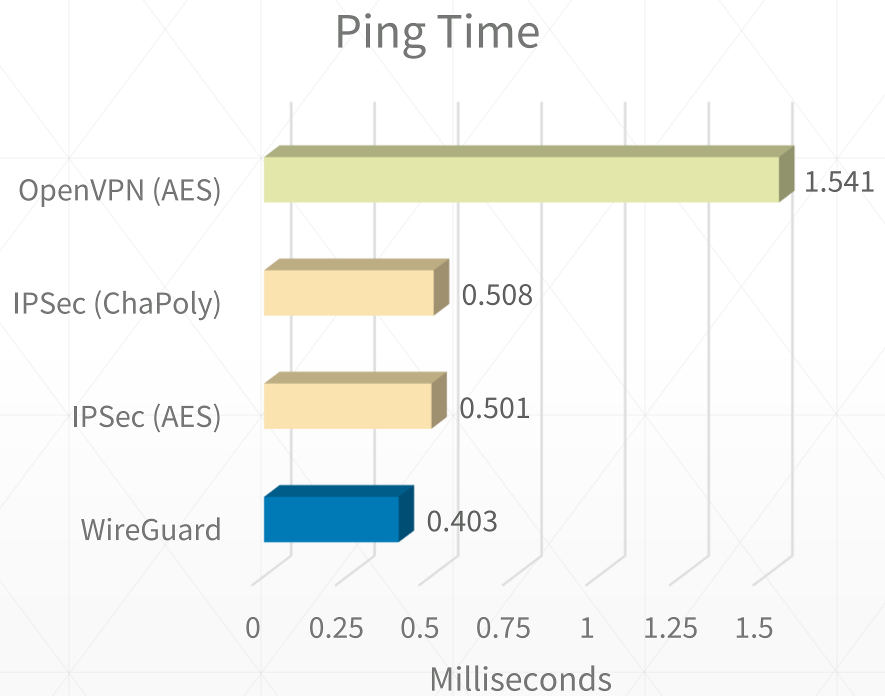
- Once IP address is attributed, ordinary token bucket rate limiting can be applied.
- Maintains stealthiness.
- Cookies cannot be intercepted by somebody who couldn't already initiate the same exchange.
- Initiator cannot be DoS'd, since the encrypted cookie uses the original handshake message as the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

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





Simple, Fast, and 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.

www.wireguard.com

WireGuard

- Paper published in NDSS 2017, available at: wireguard.com/papers/wireguard.pdf
- Real production code, not just an “academic” proof of concept
- Open source
- $ git clone https://git.zx2c4.com/WireGuard
- Mailing list: lists.zx2c4.com/mailman/listinfo/wireguard wireguard@lists.zx2c4.com

Jason Donenfeld

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