WireGuard: Next Generation Kernel Network Tunnel

www.wireguard.com

Jason A. Donenfeld
jason@zx2c4.com

Draft Revision

Abstract

WireGuard is a secure network tunnel, operating at layer 3, implemented as a kernel virtual network interface for Linux, which aims to replace both IPsec for most use cases, as well as popular user space and/or TLS-based solutions like OpenVPN, while being more secure, more performant, and easier to use. The virtual tunnel interface is based on a proposed fundamental principle of secure tunnels: an association between a peer public key and a tunnel source IP address. It uses a single round trip key exchange, based on NoiseIK, and handles all session creation transparently to the user using a novel timer state machine mechanism. Short pre-shared static keys—Curve25519 points—are used for mutual authentication in the style of OpenSSH. The protocol provides strong perfect forward secrecy in addition to a high degree of identity hiding. Transport speed is accomplished using ChaCha20Poly1305 authenticated-encryption for encapsulation of packets in UDP. An improved take on IP-binding cookies is used for mitigating denial of service attacks, improving greatly on IKEv2 and DTLS's cookie mechanisms to add encryption and authentication. The overall design allows for allocating no resources in response to received packets, and from a systems perspective, there are multiple interesting Linux implementation techniques for queues and parallelism. Finally, WireGuard can be simply implemented for Linux in less than 4,000 lines of code, making it easily audited and verified.

Permanent ID of this document: 4846ada1492f5d92198df154f48c3d54205657bc. Static link: wireguard.com/papers/wireguard.pdf. Date: June 30, 2018. This is draft revision 416d63b. A version of this paper appears in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2017*.

Contents

1 Introduction & Motivation

In Linux, the standard solution for encrypted tunnels is IPsec, which uses the Linux transform ("xfrm") layer. Users fill in a kernel structure determining which ciphersuite and key, or other transforms such as compression, to use for which selector of packets traversing the subsystem. Generally a user space daemon is responsible for updating these data structures based on the results of a key exchange, generally done with IKEv2 [13], itself a complicated protocol with much choice and malleability. The complexity, as well as the sheer amount of code, of this solution is considerable. Administrators have a completely separate set of firewalling semantics and secure labeling for IPsec packets. While separating the key exchange layer from the transport encryption—or transformation—layer is a wise separation from a semantic viewpoint, and similarly while separating the transformation layer from the interface layer is correct from a networking viewpoint, this strictly correct layering approach increases complexity and makes correct implementation and deployment prohibitive.

WireGuard does away with these layering separations. Instead of the complexity of IPsec and the xfrm layers, WireGuard simply gives a virtual interface—wg0 for example—which can then be administered using the standard ip(8) and ifconfig(8) utilities. After configuring the interface with a private key (and optionally a pre-shared symmetric key as explained in section 5.2) and the various public keys of peers with whom it will communicate securely, the tunnel simply works. Key exchanges, connections, disconnections, reconnections, discovery, and so forth happen behind the scenes transparently and reliably, and the administrator does not need to worry about these details. In other words, from the perspective of administration, the WireGuard interface appears to be *stateless*. Firewall rules can then be configured using the ordinary infrastructure for firewalling interfaces, with the guarantee that packets coming from a WireGuard interface will be authenticated and encrypted. Simple and straightforward, WireGuard is much less prone to catastrophic failure and misconfiguration than IPsec. It is important to stress, however, that *the layering of IPsec is correct and sound*; everything is in the right place with IPsec, to academic perfection. But, as often happens with correctness of abstraction, there is a profound lack of usability, and a verifiably safe implementation is very difficult to achieve. WireGuard, in contrast, starts from the basis of flawed layering violations and then attempts to rectify the issues arising from this conflation using practical engineering solutions and cryptographic techniques that solve real world problems.

On the other end of the spectrum is OpenVPN, a user space TUN/TAP based solution that uses TLS. By virtue of it being in user space, it has very poor performance—since packets must be copied multiple times between kernel space and user space—and a long-lived daemon is required; OpenVPN appears far from stateless to an administrator. While TUN/TAP interfaces (say, tun0) have similar wg0-like benefits as described above, OpenVPN is also enormously complex, supporting the entire plethora of TLS functionality, which exposes quite a bit of code to potential vulnerabilities. OpenVPN is right to be implemented in user space, since ASN.1 and x509 parsers in the kernel have historically been quite problematic (CVE-2008-1673, CVE-2016-2053), and adding a TLS stack would only make that issue worse. TLS also brings with it an enormous state machine, as well as a less clear association between source IP addresses and public keys.

For key distribution, WireGuard draws inspiration from OpenSSH, for which common uses include a very simple approach toward key management. Through a diverse set of out-of-band mechanisms, two peers generally exchange their static public keys. Sometimes it is simple as PGP-signed email, and other times it is a complicated key distribution mechanism using LDAP and certificate authorities. Importantly, for the most part OpenSSH key distribution is entirely agnostic. WireGuard follows suit. Two WireGuard peers exchange their public keys through some unspecified mechanism, and afterward they are able to communicate. In other words, WireGuard's attitude toward key distribution is that this is the wrong layer to address that particular problem, and so the interface is simple enough that any key distribution solution can be used with it. As an additional advantage, public keys are only 32 bytes long and can be easily represented in Base64 encoding in 44 characters, which is useful for transferring keys through a variety of different mediums.

Finally, WireGuard is cryptographically opinionated. It intentionally lacks cipher and protocol agility. If holes are found in the underlying primitives, all endpoints will be required to update. As shown by the continuing torrent of SSL/TLS vulnerabilities, cipher agility increases complexity monumentally. WireGuard uses a variant of Trevor Perrin's Noise [23]—which during its development received quite a bit of input from the authors of this paper for the purposes of being used in WireGuard—for a 1-RTT key exchange, with Curve25519 [5] for ECDH, HKDF [15] for expansion of ECDH results, RFC7539 [17]'s construction of ChaCha20 [3] and Poly1305 [8] for authenticated encryption, and BLAKE2s [2] for hashing. It has built-in protection against denial of service attacks, using a new crypto-cookie mechanism for IP address attributability.

Similarly opinionated, WireGuard is layer 3-only; as explained below in section 2, this is the cleanest approach for ensuring authenticity and attributability of the packets. The authors believe that layer 3 is the correct way for bridging multiple IP networks, and the imposition of this onto WireGuard allows for many simplifications, resulting in a cleaner and more easily implemented protocol. It supports layer 3 for both IPv4 and IPv6, and can encapsulate v4-in-v6 as well as v6-in-v4.

WireGuard puts together these principles, focusing on simplicity and an auditable codebase, while still being extremely high-speed and suitable for a modicum of environments. By combining the key exchange and the layer 3 transport encryption into one mechanism and using a virtual interface rather than a transform layer, WireGuard indeed breaks traditional layering principles, in pursuit of a solid *engineering* solution that is both more practical and more secure. Along the way, it employs several novel cryptographic and systems solutions to achieve its goals.

2 Cryptokey Routing

The fundamental principle of a secure VPN is an association between peers and the IP addresses each is allowed to use as source IPs. In WireGuard, peers are identified strictly by their public key, a 32-byte Curve25519 point. This means that there is a simple association mapping between public keys and a set of allowed IP addresses. Examine the following *cryptokey routing table*:

Configuration 1a

Interface Public Key	**Interface Private Key**	**Listening UDP Port**
HIgo...8ykw	yAnz...fBmk	41414
Peer Public Key	**Allowed Source IPs**	
xTIB...p8Dg	10.192.122.3/32, 10.192.124.0/24	
TrMv...WXX0	10.192.122.4/32, 192.168.0.0/16	
gN65...z6EA	10.10.10.230/32	

The interface itself has a private key and a UDP port on which it listens (more on that later), followed by a list of peers. Each peer is identified by its public key. Each then has a list of allowed source IPs.

When an outgoing packet is being transmitted on a WireGuard interface, wg0, this table is consulted to determine which public key to use for encryption. For example, a packet with a destination IP of 10.192.122.4 will be encrypted using the secure session derived from the public key TrMv...WXX0. Conversely, when wg0 receives an encrypted packet, after decrypting and authenticating it, it will only accept it if its source IP resolves in the table to the public key used in the secure session for decrypting it. For example, if a packet is decrypted from xTIB...qp8D, it will only be allowed if the decrypted packet has a source IP of 10.192.122.3 or in the range of 10.192.124.0 to 10.192.124.255; otherwise it is dropped.

With this very simple principle, administrators can rely on simple firewall rules. For example, an incoming packet on interface wg0 with a source IP of 10.10.10.230 may be considered as authentically from the peer with a public key of gN65...Bz6E. More generally, any packets arriving on a WireGuard interface will have a reliably authentic source IP (in addition, of course, to guaranteed perfect forward secrecy of the transport). Do note that this is only possible because WireGuard is strictly layer 3 based. Unlike some common VPN protocols, like L2TP/IPsec, using authenticated identification of peers at a layer 3 level enforces a much cleaner network design.

In the case of a WireGuard peer who wishes to route *all* traffic through another WireGuard peer, the cryptokey routing table could be configured more simply as:

Configuration 2a

Interface Public Key	**Interface Private Key**	**Listening UDP Port**
gN65...z6EA	gI6E...fWGE	21841
Peer Public Key	**Allowed Source IPs**	
HIgo...8ykw	0.0.0.0/0	

Here, the peer authorizes HIgo...f8yk to put packets onto wg0 with any source IP, and all packets that are outgoing on wg0 will be encrypted using the secure session associated with that public key and sent to that peer's endpoint.

2.1 Endpoints & Roaming

Of course, it is important that peers are able to send encrypted WireGuard UDP packets to each other at particular Internet endpoints. Each peer in the cryptokey routing table may *optionally* pre-specify a known external IP address and UDP port of that peer's endpoint. The reason it is optional is that if it is not specified and WireGuard receives a correctly authenticated packet from a peer, it will use the outer external source IP address for determining the endpoint.

Since a public key uniquely identifies a peer, the outer external source IP of an encrypted WireGuard packet is used to identify the remote endpoint of a peer, enabling peers to roam freely between different external IPs, between mobile networks for example, similar to what is allowed by Mosh [25]. For example, the prior cryptokey routing table could be augmented to have the initial endpoint of a peer:

Configuration 2b

Interface Public Key	**Interface Private Key**	**Listening UDP Port**
gN65...z6EA	gI6E...fWGE	21841
Peer Public Key	**Allowed Source IPs**	**Internet Endpoint**
HIgo...8ykw	0.0.0.0/0	192.95.5.69:41414

Then, this host, gN65...z6EA, sends an encrypted packet to HIgo...f8yk at 192.95.5.69:41414. After HIgo...f8yk receives a packet, it updates its table to learn that the endpoint for sending reply packets is, for example, 192.95.5.64:21841:

Configuration 1b

Interface Public Key	**Interface Private Key**	**Listening UDP Port**
HIgo...8ykw	yAnz...fBmk	41414
Peer Public Key	**Allowed Source IPs**	**Internet Endpoint**
xTIB...p8Dg	10.192.122.3/32, 10.192.124.0/24	
TrMv...WXX0	10.192.122.4/32, 192.168.0.0/16	
gN65...z6EA	10.10.10.230/32	192.95.5.64:21841

Note that the listen port of peers and the source port of packets sent are always the same, adding much simplicity, while also ensuring reliable traversal behind NAT. And since this roaming property ensures that peers will have the very latest external source IP and UDP port, there is no requirement for NAT to keep sessions open for long. (For use cases in which it is imperative to keep open a NAT session or stateful firewall indefinitely, the interface can be optionally configured to periodically send persistent authenticated keepalives.)

This design allows for great convenience and minimal configuration. While an attacker with an active man-in-the-middle could, of course, modify these unauthenticated external source IPs, the attacker would not be able to decrypt or modify any payload, which merely amounts to a denial-of-service attack, which would already be trivially possible by just dropping the original packets from this presumed man-in-the-middle position. And, as explained in section 6.5, hosts that cannot decrypt and subsequently reply to packets will quickly be forgotten.

3 Send/Receive Flow

The roaming design of section 2.1, put together with the cryptokey routing table of section 2, amounts to the following flows when receiving and sending a packet on interface wg0 using "Configuration 1" from above.

A packet is locally generated (or forwarded) and is ready to be transmitted on the outgoing interface wg0:

1. The plaintext packet reaches the WireGuard interface, wg0.
2. The destination IP address of the packet, 192.168.87.21, is inspected, which matches the peer TrMv...WXX0. (If it matches no peer, it is dropped, and the sender is informed by a standard ICMP "no route to host" packet, as well as returning -ENOKEY to user space.)
3. The symmetric sending encryption key and nonce counter of the secure session associated with peer TrMv...WXX0 are used to encrypt the plaintext packet using ChaCha20Poly1305.
4. A header containing various fields, explained in section 5.4, is prepended to the now encrypted packet.
5. This header and encrypted packet, together, are sent as a UDP packet to the Internet UDP/IP endpoint associated with peer TrMv...WXX0, resulting in an outer UDP/IP packet containing as its payload a header and encrypted inner-packet. The peer's endpoint is either pre-configured, or it is learned from the outer external source IP header field of the most recent correctly-authenticated packet received. (Otherwise, if no endpoint can be determined, the packet is dropped, an ICMP message is sent, and -EHOSTUNREACH is returned to user space.)

A UDP/IP packet reaches UDP port 41414 of the host, which is the listening UDP port of interface wg0:

1. A UDP/IP packet containing a particular header and an encrypted payload is received on the correct port (in this particular case, port 41414).
2. Using the header (described below in section 5.4), WireGuard determines that it is associated with peer TrMv...WXX0's secure session, checks the validity of the message counter, and attempts to authenticate and decrypt it using the secure session's receiving symmetric key. If it cannot determine a peer or if authentication fails, the packet is dropped.
3. Since the packet has authenticated correctly, the source IP of the outer UDP/IP packet is used to update the endpoint for peer TrMv...WXX0.
4. Once the packet payload is decrypted, the interface has a plaintext packet. If this is not an IP packet, it is dropped. Otherwise, WireGuard checks to see if the source IP address of the plaintext inner-packet routes correspondingly in the cryptokey routing table. For example, if the source IP of the decrypted plaintext packet is 192.168.31.28, the packet correspondingly routes. But if the source IP is 10.192.122.3, the packet does not route correspondingly for *this* peer, and is dropped.
5. If the plaintext packet has not been dropped, it is inserted into the receive queue of the wg0 interface.

It would be possible to separate the list of allowed IPs into two lists—one for checking the source address of incoming packets and one for choosing peer based on the destination address. But, by keeping these as part of the same list, it allows for something similar to reverse-path filtering. When sending a packet, the list is consulted based on the destination IP; when receiving a packet, that same list is consulted for determining if the source IP is allowed. However, rather than asking whether the received packet's sending peer has that source IP as part of its allowed IPs list, it instead is able to ask a more global question—which peer would be chosen in the table for that source IP, and does that peer match that of the received packet. This enforces a one-to-one mapping of sending and receiving IP addresses, so that if a packet is received from a particular peer, replies to that IP will be guaranteed to go to that same peer.

4 Basic Usage

Before going deep into the cryptography and implementation details, it may be useful to see a simple command line interface for using WireGuard, to bring concreteness to the concepts thus far presented.

Consider a Linux environment with a single physical network interface, eth0, connecting it to the Internet with a public IP of 192.95.5.69. A WireGuard interface, wg0, can be added and configured to have a tunnel IP address of 10.192.122.3 in a /24 subnet with the standard ip(8) utilities, shown on the left. The cryptokey routing table can then be configured using the wg(8) tool in a variety of fashions, including reading from configuration files, shown on the right:

Adding the wg0 interface

```
$ ip link add dev wg0 type wireguard
$ ip address add dev wg0 10.192.122.3/24
$ ip route add 10.0.0.0/8 dev wg0
$ ip address show
1: lo: <LOOPBACK> mtu 65536
    inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST> mtu 1500
    inet 192.95.5.69/24 scope global eth0
3: wg0: <POINTOPOINT,NOARP> mtu 1420
    inet 10.192.122.3/24 scope global wg0
```

Configuring the cryptokey routing table of wg0

```
$ wg setconf wg0 configuration-1.conf
$ wg show wg0
interface: wg0
  public key: HIgo...8ykw
  private key: yAnz...fBmk
  listening port: 41414
peer: xTIB...p8Dg
  allowed ips: 10.192.124.0/24, 10.192.122.3/32
peer: TrMv...WXX0
  allowed ips: 192.168.0.0/16, 10.192.122.4/32
peer: gN65...z6EA
  allowed ips: 10.10.10.230/32
  endpoint: 192.95.5.70:54421
$ ip link set wg0 up
$ ping 10.10.10.230
PING 10.10.10.230 56(84) bytes of data.
64 bytes: icmp_seq=1 ttl=49 time=0.01 ms
```

At this point, sending a packet to 10.10.10.230 on that system will send the data through the wg0 interface, which will encrypt the packet using a secure session associated with the public key gN65...z6EA and send that encrypted and encapsulated packet to 192.95.5.70:54421 over UDP. When receiving a packet from 10.10.10.230 on wg0, the administrator can be assured that it is authentically from gN65...z6EA.

5 Protocol & Cryptography

As mentioned prior, in order to begin sending encrypted encapsulated packets, a 1-RTT key exchange handshake must first take place. The initiator sends a message to the responder, and the responder sends a message back to the initiator. After this handshake, the initiator may send encrypted messages using a shared pair of symmetric keys, one for sending and one for receiving, to the responder, and following the first encrypted message from initator to responder, the responder may begin to send encrypted messages to the initiator. This ordering restriction is to require confirmation as described for KEA+C [18], as well as allowing handshake message to be processed asynchronously to transport data messages. These messages use the "IK" pattern from Noise [23], in addition to a novel cookie construction to mitigate denial of service attacks. The net result of the protocol is a very robust security system, which achieves the requirements of authenticated key exchange (AKE) security [18], avoids key-compromise impersonation, avoids replay attacks, provides perfect forward secrecy, provides identity hiding of static public keys similar to SIGMA [16], and has resistance to denial of service attacks.

5.1 Silence is a Virtue

One design goal of WireGuard is to avoid storing any state prior to authentication and to not send any responses to unauthenticated packets. With no state stored for unauthenticated packets, and with no response generated, WireGuard is invisible to illegitimate peers and network scanners. Several classes of attacks are avoided by not allowing unauthenticated packets to influence any state. And more generally, it is possible to implement WireGuard in a way that requires no dynamic memory allocation at all, even for authenticated packets, as explained in section 7. However, this property requires the very first message received by the responder to authenticate the initiator. Having authentication in the first packet like this potentially opens up the responder to a replay attack. An attacker could replay initial handshake messages to trick the responder into regenerating its ephemeral key, thereby invalidating the session of the legitimate initiator (though not affecting the secrecy or authenticity of any messages). To prevent this, a 12-byte TAI64N [7] timestamp is included, encrypted and authenticated, in the first message. The responder keeps track of the greatest timestamp received *per peer* and discards packets containing timestamps less than or equal to it. (In fact, it does not even have to be an accurate timestamp; it simply must be a per-peer monotonically increasing 96-bit number.) If the responder restarts and loses this state, that is not a problem: even though an initial packet from earlier can be replayed, it could not possibly disrupt any ongoing secure sessions, because the responder has just restarted and therefore has no active secure sessions to disrupt. Once the initiator reestablishes a secure session with the responder after its restart, the initiator will be using a greater timestamp, invalidating the previous one. This timestamp ensures that an attacker may not disrupt a current session between initiator and responder via replay attack. (This also means that two distinct peers should *not* share private keys, since in that situation a packet sent to one could be replayed to another, and the ensuing response would then cause the initiator to involuntarily roam from one peer to another. But one should not be sharing private keys in the first place, anyway.) From an implementation point of view, TAI64N [7] is very convenient because it is big-endian, allowing comparisons between two 12-byte timestamps to be done using standard `memcmp()`. Since WireGuard does not use signatures, in order to gain a degree of deniability, the first message relies only on a Diffie-Hellman result of both peers' static keys for authentication. This means that if either one of their static keys is compromised, an attacker would be able to forge an initiation message—though it would not be able to complete the full handshake—containing a maximum timestamp value, thereby preventing all future connections from succeeding. While this may seem similar to traditional key-compromise impersonation vulnerabilities—to which WireGuard is *not* vulnerable—it is in fact very different. For, if a key compromise enables an attacker to prevent peers from ever using their compromised keys again, the attacker has actually aided a proper response to such a compromise.

5.2 Optional Pre-shared Symmetric Key Mode

WireGuard rests upon peers exchanging static public keys with each other *a priori*, as their static identities. The secrecy of all data sent relies on the security of the Curve25519 ECDH function. In order to mitigate any future advances in quantum computing, WireGuard also supports a mode in which any pair of peers might *additionally* pre-share a single 256-bit symmetric encryption key between themselves, in order to add an additional layer of symmetric encryption. The attack model here is that adversaries may be recording encrypted traffic on a long term basis, in hopes of someday being able to break Curve25519 and decrypt past traffic. While pre-sharing symmetric encryption keys is usually troublesome from a key management perspective and might be more likely stolen, the idea is that by the time quantum computing advances to break Curve25519, this pre-shared symmetric key has been long forgotten. And, more importantly, in the shorter term, if the pre-shared symmetric key is compromised, the Curve25519 keys still provide more than sufficient protection. In lieu of using a completely post-quantum crypto system, which as of writing are not practical for use here, this optional hybrid approach of a pre-shared symmetric key to complement the elliptic curve cryptography provides a sound and acceptable trade-off for the extremely paranoid. Furthermore, it allows for building on top of WireGuard sophisticated key-rotation schemes, in order to achieve varying types of post-compromise security.

5.3 Denial of Service Mitigation & Cookies

Computing Curve25519 point multiplication is CPU intensive, even if Curve25519 is an extremely fast curve on most processors. In order to determine the authenticity of a handshake message, a Curve25519 multiplication must be computed, which means there is a potential avenue for a denial-of-service attack. In order to fend off a CPU-exhaustion attack, if the responder—the recipient of a message—is under load, it may choose to not process a handshake message (either an initiation or a response handshake message), but instead to respond with a cookie reply message, containing a cookie. The initiator then uses this cookie in order to resend the message and have it be accepted the following time by the responder.

The responder maintains a secret random value that changes every two minutes. A cookie is simply the result of computing a MAC of the initiator's source IP address using this changing secret as the MAC key. The initiator, when resending its message, sends a MAC of its message using this cookie as the MAC key. When the responder receives the message, if it is under load, it may choose whether or not to accept and process the message based on whether or not there is a correct MAC that uses the cookie as the key. This mechanism ties messages sent from an initiator to its IP address, giving proof of IP ownership, allowing for rate limiting using classical IP rate limiting algorithms (token bucket, etc—see section 7.4 for implementation details).

This is more or less the scheme used by DTLS [24] and IKEv2 [13]. However it suffers from three major flaws. First, as mentioned in section 5.1, we prefer to stay silent by not sending any reply to unauthenticated messages; indiscriminately sending a cookie reply message when under load would break this property. Second, the cookie should not be sent in clear text, because a man-in-the-middle could use this to then send fraudulent messages that are processed. And third, the initiator himself could be denial-of-service attacked by being sent fraudulent cookies, which it would then use with no success in computing a MAC of its message. The cookie mechanism of WireGuard, which uses two MACs (msg.mac1 and msg.mac2), fixes these problems, the computations for which will be shown in section 5.4.4 below.

For the first problem, in order for the responder to remain silent, even while under load, all messages have a first MAC (msg.mac1) that uses the responder's public key. This means that at the very least, a peer sending a message must know to whom it is talking (by virtue of knowing its public key), in order to elicit any kind of response. Under load or not under load, this first MAC (msg.mac1) always is required to be present and valid. While the public key of the responder itself is not secret, it is sufficiently secret within this attack model, in which the goal is to ensure stealthiness of services, and so knowing the responder's public key is sufficient proof for already knowing of its existence. (It is worth noting that this first MAC allows a passive attacker to make guesses about for which public key the packet is intended, slightly weakening identity hiding properties, though a correct guess would not constitute cryptographic *proof* since no private material was used in generating the MAC.)

Likewise, to solve the second problem—that of sending MACs in clear text—we apply an AEAD with an extended randomized nonce to the cookie in transit, again using as a symmetric encryption key the responder's public key. Again, the mostly public values here are sufficient for our purposes within the denial-of-service attack threat model.

Finally, to solve the third problem, we use the "additional data" field of the AEAD to encrypt the cookie in transit to additionally authenticate the first MAC (msg.mac1) of the initiating message that provoked a cookie reply message. This ensures that an attacker without a man-in-the-middle position cannot send torrents of invalid cookie replies to initiators to prevent them from authenticating with a correct cookie. (An attacker *with* an man-in-the-middle position could simply drop cookie reply messages anyway to prevent a connection, so that case is not relevant, though an attacker with a merely *passive* man-in-the-middle position could indeed forge these packets, which is not considerably different from a denial-of-service attack against TCP.) In other words, we use the AD field to bind cookie replies to initiation messages.

With these problems solved, we can then add the aforementioned second MAC (msg.mac2) using the securely transmitted cookie as the MAC key. When the responder is under load, it will only accept messages that additionally have this second MAC.

In sum, the responder, after computing these MACs as well and comparing them to the ones received in the message, must always reject messages with an invalid msg.mac1, and when under load *may* reject messages with an invalid msg.mac2. If the responder receives a message with a valid msg.mac1 yet with an invalid msg.mac2, *and is under load*, it may respond with a cookie reply message, detailed in section 5.4.7. This considerably improves on the cookie scheme used by DTLS and IKEv2.

In contrast to HIPv2 [20], which solves this problem by using a 2-RTT key exchange and complexity puzzles, WireGuard eschews puzzle-solving constructs, because the former requires storing state while the latter makes the relationship between initiator and responder asymmetric. In WireGuard, either peer at any point might be motivated to begin a handshake. This means that it is not feasible to require a complexity puzzle from the initiator, because the initatior and responder may soon change roles, turning this mitigation mechanism into a denial of service vulnerability itself. Our above cookie solution, in contrast, enables denial of service attack mitigation on a 1-RTT protocol, while keeping the initiator and responder roles symmetric.

5.4 Messages

There are four types of messages, each prefixed by a single-byte message type identifier, notated as msg.type below:

- Section 5.4.2: The handshake initiation message that begins the handshake process for establishing a secure session.
- Section 5.4.3: The handshake response to the initiation message that concludes the handshake, after which a secure session can be established.
- Section 5.4.7: A reply to either a handshake initiation message or a handshake response message, explained in section 5.3, that communicates an encrypted cookie value for use in resending either the rejected handshake initiation message or handshake response message.
- Section 5.4.6: An encapsulated and encrypted IP packet that uses the secure session negotiated by the handshake.

The initator of the handshake is denoted as subscript i, and the responder of the handshake is denoted as subscript r, and either one is denoted as subscript $*$. For messages that can be created by either an initiator or responder, if the peer creating the message is the initator, let $(m, m') = (i, r)$, and if the peer creating the message is the responder, let $(m, m') = (r, i)$. The two peers have several variables they maintain locally:

I_* A 32-bit index that locally represents the other peer, analogous to IPsec's "SPI".
S_*^{priv}, S_*^{pub} The static private and public key values.
E_*^{priv}, E_*^{pub} The ephemeral private and public key values.
Q The optional pre-shared symmetric key value from section 5.2 When pre-shared key mode is not in use, this is set to 0^{32}.
H_*, C_* A hash result value and a chaining key value.
T_*^{send}, T_*^{recv} Transport data symmetric key values for sending and receiving.
N_*^{send}, N_*^{recv} Transport data message nonce counters for sending and receiving.

In the constructions that follow, several symbols, functions, and operators are used. The binary operator $\|$ represents concatenation of its operands, and the binary operator $:=$ represents assignment of its right operand to its left operand. The annotation $\widehat{n}$ returns the value $(n + 16)$, which is the Poly1305 authentication tag length added to n. ϵ represents an empty zero-length bitstring, 0^n represents the all zero (0x0) bitstring of length n bytes, and ρ^n represents a random bitstring of length n bytes. Let τ be considered a temporary variable and let κ be considered a temporary encryption key. All integer assignments are little-endian, unless otherwise noted. The following functions and constants are utilized:

DH(private key, public key) Curve25519 point multiplication of private key and public key, returning 32 bytes of output.

DH-Generate() Generates a random Curve25519 private key and derives its corresponding public key, returning a pair of 32 bytes values, (private, public).

Aead(key, counter, plain text, auth text) ChaCha20Poly1305 AEAD, as specified in RFC7539 [17], with its nonce being composed of 32 bits of zeros followed by the 64-bit little-endian value of counter.

Xaead(key, nonce, plain text, auth text) XChaCha20Poly1305 AEAD, with a 24-byte random nonce, instantiated using HChaCha20 [6] and ChaCha20Poly1305.

Hash(input) Blake2s(input, 32), returning 32 bytes of output.

Mac(key, input) Keyed-Blake2s(key, input, 16), the keyed MAC variant of the BLAKE2s hash function, returning 16 bytes of output.

Hmac(key, input) Hmac-Blake2s(key, input, 32), the ordinary BLAKE2s hash function used in an HMAC construction, returning 32 bytes of output.

Kdf$_n$(key, input) Sets $\tau_0 := \textsc{Hmac}(\textsc{key}, \textsc{input})$, $\tau_1 := \textsc{Hmac}(\tau_0, \texttt{0x1})$, $\tau_i := \textsc{Hmac}(\tau_0, \tau_{i-1} \parallel \texttt{i})$, and returns an n-tuple of 32 byte values, $(\tau_1, \ldots, \tau_n)$. This is the HKDF [15] function.

Timestamp() Returns the TAI64N timestamp [7] of the current time, which is 12 bytes of output, the first 8 bytes being a big-endian integer of the number of seconds since 1970 TAI and the last 4 bytes being a big-endian integer of the number of nanoseconds from the beginning of that second.

Construction The UTF-8 string literal "Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s", 37 bytes of output.

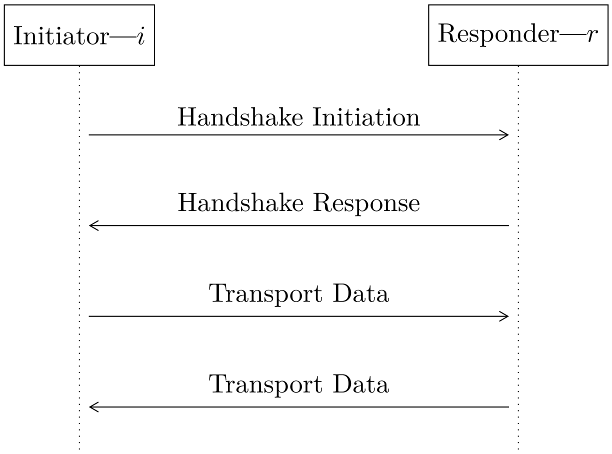
Identifier The UTF-8 string literal "WireGuard v1 zx2c4 Jason@zx2c4.com", 34 bytes of output.

Label-Mac1 The UTF-8 string literal "mac1----", 8 bytes of output.

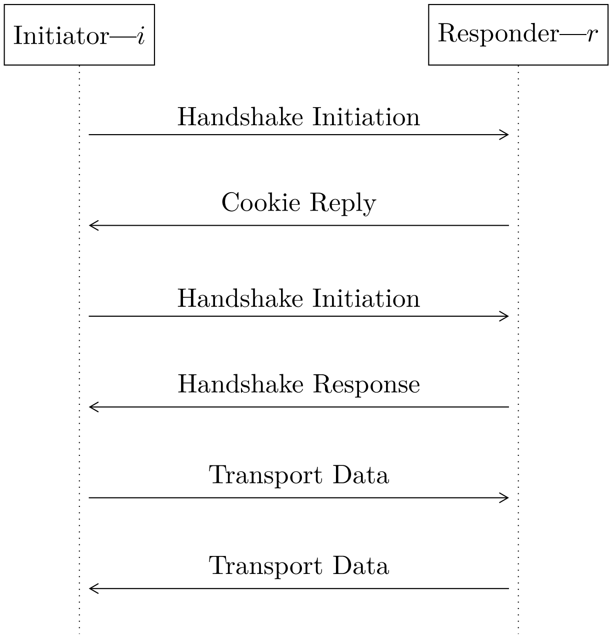
Label-Cookie The UTF-8 string literal "cookie--", 8 bytes of output.

5.4.1 Protocol Overview

In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:



If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:



5.4.2 First Message: Initiator to Responder

The initiator sends this message, msg:

type := 0x1 (1 byte)	reserved := 0^3 (3 bytes)
sender := I_i (4 bytes)	
ephemeral (32 bytes)	
static ($\widehat{32}$ bytes)	
timestamp ($\widehat{12}$ bytes)	
mac1 (16 bytes)	mac2 (16 bytes)

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. I_i is generated randomly (ρ^4) when this message is sent, and is used to tie subsequent replies to the session begun by this message. The above remaining fields are calculated [23] as follows:

$$
\begin{aligned}
&C_i := \textsc{Hash}(\textsc{Construction})\\
&H_i := \textsc{Hash}(C_i \parallel \textsc{Identifier})\\
&H_i := \textsc{Hash}(H_i \parallel S_r^{pub})\\
&(E_i^{priv}, E_i^{pub}) := \text{DH-}\textsc{Generate}()\\
&C_i := \textsc{Kdf}_1(C_i, E_i^{pub})\\
&\text{msg.ephemeral} := E_i^{pub}\\
&H_i := \textsc{Hash}(H_i \parallel \text{msg.ephemeral})\\
&(C_i, \kappa) := \textsc{Kdf}_2(C_i, \text{DH}(E_i^{priv}, S_r^{pub}))\\
&\text{msg.static} := \textsc{Aead}(\kappa, 0, S_i^{pub}, H_i)\\
&H_i := \textsc{Hash}(H_i \parallel \text{msg.static})\\
&(C_i, \kappa) := \textsc{Kdf}_2(C_i, \text{DH}(S_i^{priv}, S_r^{pub}))\\
&\text{msg.timestamp} := \textsc{Aead}(\kappa, 0, \textsc{Timestamp}(), H_i)\\
&H_i := \textsc{Hash}(H_i \parallel \text{msg.timestamp})
\end{aligned}
$$

When the responder receives this message, it does the same operations so that its final state variables are identical, replacing the operands of the DH function to produce equivalent values.

5.4.3 Second Message: Responder to Initiator

The responder sends this message, after processing the first message above from the initiator and applying the same operations to arrive at an identical state. I_r is generated randomly (ρ^4) when this message is sent, and is used to tie subsequent replies to the session begun by this message, just as above. The responder sends this message, msg:

type := 0x2 (1 byte)	reserved := 0^3 (3 bytes)
sender := I_r (4 bytes)	receiver := I_i (4 bytes)
ephemeral (32 bytes)	
empty ($\widehat{0}$ bytes)	
mac1 (16 bytes)	mac2 (16 bytes)

The fields mac1 and mac2 are explained further in section 5.4.4. The above remaining fields are calculated [23] as follows:

$$
\begin{aligned}
&(E_r^{priv}, E_r^{pub}) := \text{DH-}\textsc{Generate}()\\
&C_r := \textsc{Kdf}_1(C_r, E_r^{pub})\\
&\text{msg.ephemeral} := E_r^{pub}\\
&H_r := \textsc{Hash}(H_r \parallel \text{msg.ephemeral})\\
&C_r := \textsc{Kdf}_1(C_r, \text{DH}(E_r^{priv}, E_i^{pub}))\\
&C_r := \textsc{Kdf}_1(C_r, \text{DH}(E_r^{priv}, S_i^{pub}))\\
&(C_r, \tau, \kappa) := \textsc{Kdf}_3(C_r, Q)\\
&H_r := \textsc{Hash}(H_r \parallel \tau)\\
&\text{msg.empty} := \textsc{Aead}(\kappa, 0, \epsilon, H_r)\\
&H_r := \textsc{Hash}(H_r \parallel \text{msg.empty})
\end{aligned}
$$

When the initiator receives this message, it does the same operations so that its final state variables are identical, replacing the operands of the DH function to produce equivalent values. Note that this handshake response message is smaller than the handshake initiation message, preventing amplification attacks.

5.4.4 Cookie MACs

In sections 5.4.2 and 5.4.3, the two handshake messages have the msg.mac1 and msg.mac2 parameters. For a given handshake message, msg_α represents all bytes of msg prior to msg.mac1, and msg_β represents all bytes of msg prior to msg.mac2. The latest cookie received $\widetilde{L_*}$ seconds ago is represented by L_*. The msg.mac1 and msg.mac2 fields are populated as follows:

$$\begin{aligned}
&\text{msg.mac1} \coloneqq \textsc{Mac}(\textsc{Hash}(\textsc{Label-Mac1} \parallel S_{m'}^{pub}), \text{msg}_\alpha)\\
&\text{if } L_m = \epsilon \text{ or } \widetilde{L_m} \geq 120\text{:}\\
&\qquad \text{msg.mac2} \coloneqq 0^{16}\\
&\text{otherwise:}\\
&\qquad \text{msg.mac2} \coloneqq \textsc{Mac}(L_m, \text{msg}_\beta)
\end{aligned}$$

The value $\textsc{Hash}(\textsc{Label-Mac1} \parallel S_{m'}^{pub})$ above can be pre-computed.

5.4.5 Transport Data Key Derivation

After the above two messages have been exchanged, keys are calculated [23] by the initiator and responder for sending and receiving transport data messages (section 5.4.6):

$$\begin{aligned}
&(T_i^{send} = T_r^{recv}, T_i^{recv} = T_r^{send}) \coloneqq \textsc{Kdf}_2(C_i = C_r, \epsilon)\\
&N_i^{send} = N_r^{recv} = N_i^{recv} = N_r^{send} \coloneqq 0\\
&E_i^{priv} = E_i^{pub} = E_r^{priv} = E_r^{pub} = C_i = C_r \coloneqq \epsilon
\end{aligned}$$

On the last line, most prior states of the handshake are zeroed from memory (described in section 7.4), but the value $H_i = H_r$ is not necessarily zeroed, as it could potentially be useful in future revisions of Noise [23].

5.4.6 Subsequent Messages: Transport Data Messages

The initiator and the responder exchange transport data messages for exchanging encrypted encapsulated packets. The inner plaintext packet that is encapsulated is represented as P, of length $\|P\|$. Both peers send this message, msg:

<table>
<tr><td>type ≔ 0x4 (1 byte)</td><td>reserved ≔ 0^3 (3 bytes)</td></tr>
<tr><td colspan="2">receiver ≔ $I_{m'}$ (4 bytes)</td></tr>
<tr><td colspan="2">counter (8 bytes)</td></tr>
<tr><td colspan="2">packet ($\widehat{\|P\|}$ bytes)</td></tr>
</table>

The remaining fields are populated as follows:

$$\begin{aligned}
&P \coloneqq P \parallel 0^{16 \cdot \lceil \|P\|/16 \rceil - \|P\|}\\
&\text{msg.counter} \coloneqq N_m^{send}\\
&\text{msg.packet} \coloneqq \textsc{Aead}(T_m^{send}, N_m^{send}, P, \epsilon)\\
&N_m^{send} \coloneqq N_m^{send} + 1
\end{aligned}$$

The recipient of this messages uses $T_{m'}^{recv}$ to read the message. Note that no length value is stored in this header, since the authentication tag serves to determine whether the message is legitimate, and the inner IP packet already has a length field in its header. The encapsulated packet itself is zero padded (without modifying the IP packet's length field) before encryption to complicate traffic analysis, though that zero padding should never increase the UDP packet size beyond the maximum transmission unit length. Prior to msg.packet, there are exactly 16 bytes of header fields, which means that decryption may be done in-place and still achieve natural memory address alignment, allowing for easier implementation in hardware and a significant performance improvement on many common CPU architectures. This is in part the result of the 3 bytes of reserved zero fields, making the first four bytes readable together as a little-endian integer.

The msg.counter value is a *nonce* for the ChaCha20Poly1305 AEAD and is kept track of by the recipient using $N_{m'}^{recv}$. It also functions to avoid replay attacks. Since WireGuard operates over UDP, messages can sometimes arrive out of order. For that reason we use a sliding window to keep track of received message counters, in which we keep track of the greatest counter received, as well as a window of prior messages received, using the algorithm detailed by appendix C of RFC2401 [14] or by RFC6479 [26], which uses a larger bitmap while avoiding bitshifts, enabling more extreme packet reordering that may occur on multi-core systems.

5.4.7 Under Load: Cookie Reply Message

As mentioned in section 5.3, when a message with a valid msg.mac1 is received, but msg.mac2 is invalid or expired, and the peer is under load, the peer may send a cookie reply message. $I_{m'}$ is determined from the msg.sender field of the message that prompted this cookie reply message, msg:

type $:=$ 0x3 (1 byte)	reserved $:= 0^3$ (3 bytes)
receiver $:= I_{m'}$ (4 bytes)	
nonce $:= \rho^{24}$ (24 bytes)	
cookie ($\widehat{16}$ bytes)	

The secret variable, R_m, changes every two minutes to a random value, $A_{m'}$ represents a concatenation of the subscript's external IP source address and UDP source port, and M represents the msg.mac1 value of the message to which this is in reply. The remaining encrypted cookie reply field is populated as such:

$$\begin{aligned} &\tau := \textsc{Mac}(R_m, A_{m'}) \\ &\text{msg.cookie} := \textsc{Xaead}(\textsc{Hash}(\textsc{Label-Cookie} \parallel S_m^{pub}), \text{msg.nonce}, \tau, M) \end{aligned}$$

The value $\textsc{Hash}(\textsc{Label-Cookie} \parallel S_m^{pub})$ above can be pre-computed. By using M as the additional authenticated data field, we bind the cookie reply to the relevant message, in order to prevent peers from being attacked by sending them fraudulent cookie reply messages. Also note that this message is smaller than either the handshake initiation message or the handshake response message, avoiding amplification attacks.

Upon receiving this message, if it is valid, the only thing the recipient of this message should do is store the cookie along with the time at which it was received. The mechanism described in section 6 will be used for retransmitting handshake messages with these received cookies; this cookie reply message should not, by itself, cause a retransmission.

6 Timers & Stateless UX

From the perspective of the user, WireGuard appears stateless. The private key of the interface is configured, followed by the public key of each of its peers, and then a user may simply send packets normally. The maintenance of session states, perfect forward secrecy, handshakes, and so forth is completely behind the scenes, invisible to the user. While similar automatic mechanisms historically have been buggy and disastrous, WireGuard employs an extremely *simple* timer state machine, in which each state and transitions to all adjacent states are clearly defined, resulting in total reliability. There are no anomalous states or sequences of states; everything is accounted for. It has been tested with success on 10 gigabit intranets as well as on low-bandwidth high-latency transatlantic commercial airline Internet. The simplicity of the timer state machine is owed to the fact that only a 1-RTT handshake is required, that the initiator and responder can transparently switch roles, and that WireGuard breaks down traditional layering, as discussed in section 1, and can therefore use intra-layer characteristics.

6.1 Preliminaries

The following constants are used for the timer state system:

Symbol	**Value**
Rekey-After-Messages	$2^{64} - 2^{16} - 1$ messages
Reject-After-Messages	$2^{64} - 2^{4} - 1$ messages
Rekey-After-Time	120 seconds
Reject-After-Time	180 seconds
Rekey-Attempt-Time	90 seconds
Rekey-Timeout	5 seconds
Keepalive-Timeout	10 seconds

Under no circumstances will WireGuard send an initiation message more than once every *Rekey-Timeout*. A secure session is created after the successful receipt of a handshake response message (section 5.4.3), and the age of a secure session is measured from the time of processing this message and the immediately following derivation of transport data keys (section 5.4.5).

6.2 Transport Message Limits

After a secure session has first been established, WireGuard will try to create a new session, by sending a handshake initiation message (section 5.4.2), after it has sent *Rekey-After-Messages* transport data messages.

Likewise, if a peer is the initiator of a current secure session, WireGuard will send a handshake initiation message to begin a new secure session if, after transmitting a transport data message, the current secure session is *Rekey-After-Time* seconds old, or if after receiving a transport data message, the current secure session is (*Reject-After-Time* − *Keepalive-Timeout* − *Rekey-Timeout*) seconds old and it has not yet acted upon this event. This time-based opportunistic rekeying is restricted to the initiator of the current session, in order to prevent the "thundering herd" problem, in which both peers might try to establish a new session at the same time. Due to the passive keepalive feature, described in section 6.5, the initiation triggered by an old secure session *after* transmitting a transport data message should usually be sufficient to ensure new sessions are created every *Rekey-After-Time* seconds. However, for the case in which a peer has received data but does not have any data to send back immediately, and the *Reject-After-Time* second deadline is approaching in sooner than *Keepalive-Timeout* seconds, then the initiation triggered by an aged secure session occurs during the receive path.

After *Reject-After-Messages* transport data messages or after the current secure session is *Reject-After-Time* seconds old, whichever comes first, WireGuard will refuse to send or receive any more transport data messages using the current secure session, until a new secure session is created through the 1-RTT handshake.

6.3 Key Rotation

New secure sessions are created approximately every *Rekey-After-Time* seconds (which is far more likely to occur before *Rekey-After-Messages* transport data messages have been sent), due to the transport message limits described above in section 6.2. This means that the secure session is constantly rotating, creating a new ephemeral symmetric session key each time, for perfect forward secrecy. But, keep in mind that after an initiator receives a handshake response message (section 5.4.3), the responder cannot send transport data messages (section 5.4.6) until it has received the first transport data message from the initiator. And, further, transport data messages encrypted using the previous secure session might be in transit after a new secure session has been created. For these reasons, WireGuard keeps in memory the current secure session, the previous secure session, and the next secure session for the case of an unconfirmed session. Every time a new secure session is created, the existing one rotates into the "previous" slot, and the new one occupies the "current" slot, for the initiator, and for the responder, the "next" slot is used interstitially until the handshake is confirmed. The "previous-previous" one is then discarded and its memory is zeroed (see section 7.4 for a discussion of memory zeroing). If no new secure session is created after (*Reject-After-Time* × 3) seconds, the current secure session, the previous secure session, and potentially the next secure session are discarded and zeroed out, in addition to any possible partially-completed handshake states and ephemeral keys.

6.4 Handshake Initiation Retransmission

The first time the user sends a packet over a WireGuard interface, the packet cannot immediately be sent, because no current session exists. So, after queuing the packet, WireGuard sends a handshake initiation message (section 5.4.2).

After sending a handshake initiation message, because of a first-packet condition, or because of the limit conditions of section 6.2, if a handshake response message (section 5.4.3) is not subsequently received after *Rekey-Timeout* seconds, a new handshake initiation message is constructed (with new random ephemeral keys) and sent. This reinitiation is attempted for *Rekey-Attempt-Time* seconds before giving up, though this counter is reset when a peer explicitly attempts to send a *new* transport data message. Critically important future work includes adjusting the *Rekey-Timeout* value to use exponential backoff, instead of the current fixed value.

6.5 Passive Keepalive

Most importantly, and most elegant, WireGuard implements a passive keepalive mechanism to ensure that sessions stay active and allow both peers to passively determine if a connection has failed or been disconnected. If a peer has received a validly-authenticated transport data message (section 5.4.6), but does not have any packets itself to send back for *Keepalive-Timeout* seconds, it sends a *keepalive message*. A keepalive message is simply a transport data message with a zero-length encapsulated encrypted inner-packet. Since all other transport data messages contain IP packets, which have a minimum length of min(‖IPv4 header‖, ‖IPv6 header‖), this keepalive message can be easily distinguished by simple virtue of having a zero length encapsulated packet. (Note that the msg.packet field of the message will in fact be of length 16, the length of the Poly1305 [8] authentication tag, since a zero length plaintext still needs to be authenticated, even if there is nothing to encrypt.)

This passive keepalive is only sent when a peer has nothing to send, and is only sent in circumstances when another peer is sending authenticated transport data messages to it. This means that when neither side is exchanging transport data messages, the network link will be silent.

Because every transport data message sent warrants a reply of some kind—either an organic one generated by the nature of the encapsulated packets or this keepalive message—we can determine if the secure session is broken or disconnected if a transport data message has not been received for (*Keepalive-Timeout* + *Rekey-Timeout*) seconds, in which case a handshake initiation message is sent to the unresponsive peer, once every *Rekey-Timeout* seconds, as in section 6.4, until a secure session is recreated successfully or until *Rekey-Attempt-Time* seconds have passed.

6.6 Interaction with Cookie Reply System

As noted in sections 5.3 and 5.4.7, when a peer is under load, a handshake initiation message or a handshake response message may be discarded and a cookie reply message sent. On receipt of the cookie reply message, which will enable the peer to send a new initiation or response message with a valid msg.mac2 that will not be discarded, the peer is *not* supposed to immediately resend the now valid message. Instead, it should simply store the decrypted cookie value from the cookie reply message, and wait for the expiration of the *Rekey-Timeout* timer for retrying a handshake initiation message. This prevents potential bandwidth generation abuse, and helps to alleviate the load conditions that are requiring the cookie reply messages in the first place.

7 Linux Kernel Implementation

The implementation of WireGuard inside the Linux kernel has a few goals. First, it should be short and simple, so that auditing and reviewing the code for security vulnerabilities is not only easy, but also enjoyable; WireGuard is implemented in *less than 4,000 lines of code* (excluding cryptographic primitives). Second, it must be extremely fast, so that it is competitive with IPsec on performance. Third, it must avoid allocations and other resource intensive allocations in response to incoming packets. Fourth, it must integrate as natively and smoothly as possible with existing kernel infrastructure and userland expectations, tools, and APIs. And fifth, it must be buildable as an external kernel module without requiring any changes to the core Linux kernel. WireGuard is not merely an academic project with never-released laboratory code, but rather a practical project aiming for production-ready implementations.

7.1 Queuing System

The WireGuard device driver has flags indicating to the kernel that it supports generic segmentation offload (GSO), scatter gather I/O, and hardware checksum offloading, which in sum means that the kernel will hand "super packets" to WireGuard, packets that are well over the MTU size, having been priorly queued up by the upper layers, such as TCP or the TCP and UDP corking systems. This allows WireGuard to operate on batch groups of outgoing packets. After splitting packets into ≤MTU-sized chunks, WireGuard attempts to encrypt, encapsulate, and send over UDP all of these at once, caching routing information, so that it only has to be computed once per cluster of packets. This has the very important effect of also reducing cache misses: by waiting until all individual packets of a super packet have been encrypted and encapsulated to pass them off to the network layer, the very complicated and CPU-intensive network layer keeps instructions, intermediate variables, and branch predictions in CPU cache, giving in many cases a 35% increase in sending performance.

As well, as mentioned in section 6.4, sometimes outgoing packets must be queued until a handshake completes successfully. When packets are finally able to be sent, the entire queue of existing queued packets along are treated as a single super packet, in order to benefit from the same optimizations as above.

Finally, in order to prevent against needless allocations, all packet transformations are done *in-place*, avoiding the need for copying. This applies not only to the encryption and decryption of data, which occur in-place, but also to certain user space data and files sent using sendfile(2); these are processed using this zero-copy super packet queuing system.

Future work on the queuing system could potentially involve integrating WireGuard with the FlowQueue [12]-CoDel [21] scheduling algorithm.

7.2 Softirq & Parallelism

The xfrm layer, in contrast to WireGuard, has the advantage that it does not need to do cryptography in softirq, which opens it up to a bit more flexibility. However, there *is* precedent for doing cryptographic processing in softirq on the interface level: the mac802111 subsystem used for wireless WPA encryption. WireGuard, being a virtual interface that does encryption, is not architecturally so much different from wireless interfaces doing encryption at the same layer. While in practice it does work very well, it is not parallel. For this reason, the kernel's padata system is used for parallelizing into concurrent workers encryption and decryption operations for utilization of all CPUs and CPU cores. As well, packet checksums can be computed in parallel with this method. When sending packets, however, they must be sent in order, which means each packet cannot simply be sent immediately after it is encrypted. Fortunately, the padata API divides operations up into a parallel step, followed by an in-order serial step. This is also helpful for parallel decryption, in which the message counter must be checked and incremented in the order that packets arrive, lest they be rejected unnecessarily. In order to reduce latency, if there is only a single packet in a super packet and its length is less than 256 bytes, or if there is only one CPU core online, the packet is processed in softirq.

Likewise, handshake initiation and response messages and cookie reply messages are processed on a separate parallel low-priority worker thread. As mentioned in section 5.3, ECDH operations are CPU intensive, so it is important that a flood of handshake work does not monopolize the CPU. Low priority background workqueues are employed for this asynchronous handshake message handling.

7.3 RTNL-based Virtual Interface & Containerization

In order to integrate with the existing ip(8) utilities and the netlink-based Linux user space, the kernel's RTNL layer is used for registering a virtual interface, known inside the kernel as a "link". This easily gives access to the kernel APIs accessed by ip-link(8) and ip-set(8). For configuring the interface private key and the public keys and endpoints of peers, initially the RTM_SETLINK RTNL message was used, but this proved to be too limited. It proved to be much cleaner to simply implement an ioctl(2)-based API, passing a series of structures back and forth. While that approach was quite clean, the Linux networking stack is moving toward an entirely Netlink-based configuration API, so ultimately the Generic Netlink protocol was chosen. A separate user space tool, wg(8), is used for communicating over Netlink, and future plans involve integrating this functionality directly into ip(8).

The RTNL subsystem allows for moving the WireGuard virtual interface between network namespaces. This enables the sending and receiving sockets (for the outer UDP packets) to be created in one namespace, while the interface itself remains in another namespace. For example, a docker(1) or rkt(1) container guest could have as its sole network interface a WireGuard interface, with the actual outer encrypted packets being sent out of the real network interface on the host, creating end-to-end authenticated encryption in and out of the container.

7.4 Data Structures and Primitives

While the Linux kernel already includes two elaborate routing table implementations—an LC-trie [22] for IPv4 and a radix trie for IPv6—they are intimately tied to the FIB routing layer, and not at all reusable for other uses. For this reason, a very minimal routing table was developed. The authors have had success implementing the cryptokey routing table as an allotment routing table [11], an LC-trie [22], and a standard radix trie, with each one giving adequate but slightly different performance characteristics. Ultimately the simplicity of the venerable radix trie was preferred, having good performance characteristics and the ability to implement it with lock-less lookups, using the RCU system [19]. Every time an outgoing packet goes through WireGuard, the destination peer is looked up using this table, and every time an incoming packet reaches WireGuard, its validity is checked by consulting this table, so performance is in fact important here.

For all handshake initiation messages (section 5.4.2), the responder must lookup the decrypted static public key of the initiator. For this, WireGuard employs a hash table using the extremely fast SipHash2-4 [1] MAC function with a secret, so that upper layers, which may provide the WireGuard interface with public keys in a more complicated key distribution scheme, cannot mount a hash table collision denial of service attack.

While the Linux kernel's crypto API has a large collection of primitives and is meant to be reused in several different systems, the API introduces needless complexity and allocations. Several revisions of WireGuard used the crypto API with different integration techniques, but ultimately, using raw primitives with direct, non-abstracted APIs proved to be far cleaner and less resource intensive. Both stack and heap pressure were reduced by using crypto primitives directly, rather than going through the kernel's crypto API. The crypto API also makes it exceedingly difficult to avoid allocations when using multiple keys in the multifaceted ways required by Noise. As of writing, WireGuard ships with optimized implementations of ChaCha20Poly1305 for the various Intel Architecture vector extensions, with implementations for ARM/NEON and MIPS on their way. The fastest implementation supported by the hardware is selected at runtime, with the floating-point unit being used opportunistically. All ephemeral keys and intermediate results of cryptographic operations are zeroed out of memory after use, in order to maintain perfect forward secrecy and prevent against various potential leaks. The compiler must be specially informed about this explicit zeroing so that the "dead-store" is not optimized out, and for this the kernel provides the `memzero_explicit` function.

In contrast to crypto primitives, the existing kernel implementations of token bucket hash-based rate limiting, for rate limiting handshake initiation and response messages when under-load after cookie IP attribution has occurred, have been very minimal and easy to reuse in WireGuard. WireGuard uses the Netfilter `hashlimit` matcher for this.

7.5 FIB Considerations

In order to avoid routing loops, one proposed change for the Linux kernel—currently posted by the authors to the Linux kernel mailing list [9]—is to allow for FIB route lookups that exclude an interface. This way, the kernel's routing table could have `0.0.0.0/1` and `128.0.0.0/1`, for a combined coverage of `0.0.0.0/0`, while being more specific, sent to the `wg0` interface. Then, the individual endpoints of WireGuard peers could be routed using the device that a FIB lookup would return if `wg0` did not exist, namely one through the actual `0.0.0.0/0` route. Or more generally, when looking up the correct interface for routing packets to particular peer endpoints, a route for an interface would be returned that is *guaranteed* not to be `wg0`. This is preferable to the current situation of needing to add explicit routes for WireGuard peer endpoints to the kernel routing table when the WireGuard-bound route has precedence. This work is ongoing.

Another approach, alluded to above, is to use network namespaces to entirely isolate the WireGuard interface and routing table from the physical interfaces and routing tables. One namespace would contain the WireGuard interface and a routing table with a default route to send all packets over the WireGuard interface. The other namespace would contain the various physical interfaces (Ethernet devices, wireless radios, and so forth) along with its usual routing table. The incoming and outgoing UDP socket for the WireGuard interface would live *in the second physical interface namespace*, not the first WireGuard interface namespace. This way, packets sent in the WireGuard interface namespace are encrypted there, and then sent using a socket that lives in the physical interface namespace. This prevents all routing loops and also ensures total isolation. Processes living in the WireGuard interface namespace would have as their only networking means the WireGuard interface, preventing any potential clear-text packet leakage.

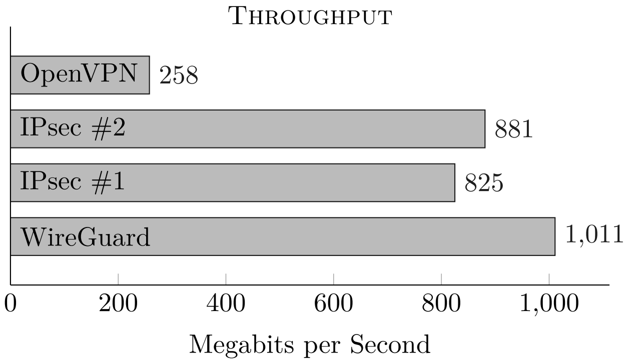
7.6 Potential Userspace Implementations

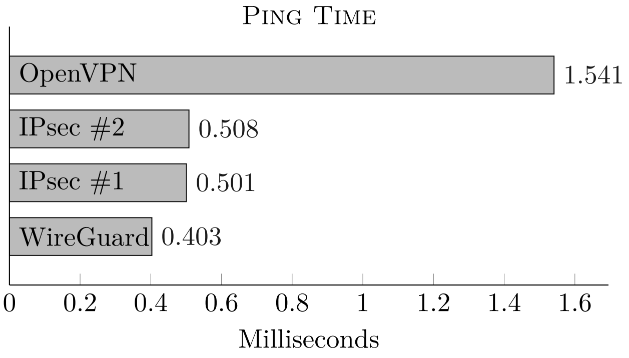
In order for WireGuard to have widespread adoption, more implementations than our current one for the Linux kernel must be written. As a next step, the authors plan to implement a cross-platform low-speed user space TUN-based implementation in a safe yet high-speed language like Rust, Go, or Haskell.

8 Performance

WireGuard was benchmarked alongside IPsec in two modes and OpenVPN, using `iperf3(1)` between an Intel Core i7-3820QM and an Intel Core i7-5200U with Intel 82579LM and Intel I218LM gigabit Ethernet cards respectively, with results averaged over thirty minutes. The results were quite promising:

Protocol	**Configuration**
WireGuard	256-bit ChaCha20, 128-bit Poly1305
IPsec #1	256-bit ChaCha20, 128-bit Poly1305
IPsec #2	256-bit AES, 128-bit GCM
OpenVPN	256-bit AES, HMAC-SHA2-256, UDP mode





For both metrics, WireGuard outperformed OpenVPN and both modes of IPsec. The CPU was at 100% utilization during the throughput tests of OpenVPN and IPsec, but was not completely utilized for the test of WireGuard, suggesting that WireGuard was able to completely saturate the gigabit Ethernet link.

While the AES-NI-accelerated AES-GCM IPsec cipher suite appears to outperform the AVX2-accelerated ChaCha20Poly1305 IPsec cipher suite, as future chips increase the width of vector instructions—such as the upcoming AVX512—it is expected that over time ChaCha20Poly1305 will outperform AES-NI [4]. ChaCha20Poly1305 is especially well suited to be implemented in software, free from side-channel attacks, with great efficiency, in contrast to AES, so for embedded platforms with no dedicated AES instructions, ChaCha20Poly1305 will also be most performant.

Furthermore, WireGuard already outperforms both IPsec cipher suites, due to the simplicity of implementation and lack of overhead. The enormous gap between OpenVPN and WireGuard is to be expected, both in terms of ping time and throughput, because OpenVPN is a user space application, which means there is added latency and overhead of the scheduler and copying packets between user space and kernel space several times.

9 Conclusion

In less than 4,000 lines, WireGuard demonstrates that it is possible to have secure network tunnels that are simply implemented, extremely performant, make use of state of the art cryptography, and remain easy to administer. The simplicity allows it to be very easily independently verified and reimplemented on a wide diversity of platforms. The cryptographic constructions and primitives utilized ensure high-speed in a wide diversity of devices, from data center servers to cellphones, as well as dependable security properties well into the future. The ease of deployment will also eliminate many of the common and disastrous pitfalls currently seen with many IPsec deployments. Described around the time of its introduction by Ferguson and Schneier [10], "IPsec was great disappointment to us. Given the quality of the people that [*sic*] worked on it and the time that was spent on it, we expected a much better result. [. . .] Our main criticism of IPsec is its complexity."

WireGuard, in contrast, focuses on simplicity and usability, while still delivering a scalable and highly secure system. By remaining silent to unauthenticated packets and by not making any allocations and generally keeping resource utilization to a minimum, it can be deployed on the outer edges of networks, as a trustworthy and reliable access point, which does not readily reveal itself to attackers nor provide a viable attack target. The cryptokey routing table paradigm is easy to learn and will promote safe network designs. The protocol is based on cryptographically sound and conservative principles, using well understood yet modern crypto primitives. WireGuard was designed from a practical perspective, meant to solve real world secure networking problems.

10 Acknowledgments

WireGuard was made possible with the great advice and guidance of many, in particular: Trevor Perrin, Jean-Philippe Aumasson, Steven M. Bellovin, and Greg Kroah-Hartman.

References

[1] Jean-Philippe Aumasson and Daniel J. Bernstein. “Progress in Cryptology - INDOCRYPT 2012: 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings”. In: ed. by Steven Galbraith and Mridul Nandi. Document ID: b9a943a805fbfc6fde808af9fc0ecdfa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. SipHash: A Fast Short-Input PRF, pp. 489–508. ISBN: 978-3-642-34931-7. DOI: 10.1007/978-3-642-34931-7_28. URL: https://cr.yp.to/siphash/siphash-20120918.pdf (cit. on p. 17).

[2] Jean-Philippe Aumasson et al. “BLAKE2: Simpler, Smaller, Fast As MD5”. In: *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*. ACNS’13. Banff, AB, Canada: Springer-Verlag, 2013, pp. 119–135. ISBN: 978-3-642-38979-5. DOI: 10.1007/978-3-642-38980-1_8. URL: https://blake2.net/blake2.pdf (cit. on p. 3).

[3] Daniel J. Bernstein. “ChaCha, a variant of Salsa20”. In: *SASC 2008*. Document ID: 4027b5256e17b97968 42e6d0f68b0b5e. 2008. URL: https://cr.yp.to/chacha/chacha-20080128.pdf (cit. on p. 3).

[4] Daniel J. Bernstein. *CPUs Are Optimized for Video Games*. URL: https://moderncrypto.org/mail-archive/noise/2016/000699.html (cit. on p. 18).

[5] Daniel J. Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *Public Key Cryptography – PKC 2006*. Ed. by Moti Yung et al. Vol. 3958. Lecture Notes in Computer Science. Document ID: 4230efdfa673 480fc079449d90f322c0. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. DOI: 10.1007/11745853_14. URL: https://cr.yp.to/ecdh/curve25519-20060209.pdf (cit. on p. 3).

[6] Daniel J. Bernstein. *Extending the Salsa20 nonce*. Document ID: c4b172305ff16e1429a48d9434d50e8a. 2011. URL: https://cr.yp.to/snuffle/xsalsa-20110204.pdf (cit. on p. 9).

[7] Daniel J. Bernstein. *TAI64, TAI64N, and TAI64NA*. URL: https://cr.yp.to/libtai/tai64.html (cit. on pp. 7, 10).

[8] Daniel J. Bernstein. “The Poly1305-AES Message-Authentication Code”. In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*. Vol. 3557. Lecture Notes in Computer Science. Document ID: 0018d9551b5546d97c340e0dd8cb5750. Springer, 2005, pp. 32–49. DOI: 10.1007/11502760_3. URL: https://cr.yp.to/mac/poly1305-20050329.pdf (cit. on pp. 3, 15).

[9] Jason A. Donenfeld. *Inverse of flowi{4,6}_oif: flowi{4,6}_not_oif*. URL: http://lists.openwall.net/netdev/2016/02/02/222 (cit. on p. 17).

[10] Niels Ferguson and Bruce Schneier. *A Cryptographic Evaluation of IPsec*. Tech. rep. Counterpane Internet Security, Inc, 2000. DOI: 10.1.1.33.7922. URL: https://www.schneier.com/cryptography/paperfiles/paper-ipsec.pdf (cit. on p. 18).

[11] Yoichi Hariguchi. *Allotment Routing Table: A Fast Free Multibit Trie Based Routing Table*. 2002. URL: https://github.com/hariguchi/art/blob/master/docs/art.pdf (cit. on p. 17).

[12] Toke Hoeiland-Joergensen et al. *The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm*. RFC. Internet Engineering Task Force, Mar. 2016, p. 23. URL: https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06 (cit. on p. 16).

[13] C. Kaufman et al. *Internet Key Exchange Protocol Version 2*. RFC 5996. RFC Editor, Sept. 2010. URL: http://www.rfc-editor.org/rfc/rfc5996.txt (cit. on pp. 3, 8).

[14] Stephen Kent and Randall Atkinson. *Security Architecture for IP*. RFC 2401. RFC Editor, Nov. 1998, p. 57. URL: http://www.rfc-editor.org/rfc/rfc2401.txt (cit. on p. 13).

[15] Hugo Krawczyk. "Advances in Cryptology – CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings". In: ed. by Tal Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. Cryptographic Extraction and Key Derivation: The HKDF Scheme, pp. 631–648. ISBN: 978-3-642-14623-7. DOI: 10.1007/978-3-642-14623-7_34. URL: https://eprint.iacr.org/2010/264.pdf (cit. on pp. 3, 10).

[16] Hugo Krawczyk. "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 400–425. DOI: 10.1007/978-3-540-45146-4_24. URL: http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1495/1495.pdf (cit. on p. 7).

[17] Adam Langley and Yoav Nir. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 7539. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7539.txt (cit. on pp. 3, 9).

[18] Kristin Lauter and Anton Mityagin. "Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings". In: ed. by Moti Yung et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Chap. Security Analysis of KEA Authenticated Key Exchange Protocol, pp. 378–394. ISBN: 978-3-540-33852-9. DOI: 10.1007/11745853_25. URL: http://research.microsoft.com/en-us/um/people/klauter/pkcspringer.pdf (cit. on p. 7).

[19] Paul E. McKenny et al. "Read-Copy Update". In: *Ottawa Linux Symposium*. June 2002, pp. 338–367. URL: http://www.rdrop.com/~paulmck/RCU/rcu.2002.07.08.pdf (cit. on p. 17).

[20] R. Moskowitz et al. *Host Identity Protocol Version 2*. RFC 7401. RFC Editor, Apr. 2015. URL: http://www.rfc-editor.org/rfc/rfc7401.txt (cit. on p. 9).

[21] Kathleen Nichols and Van Jacobson. "Controlling Queue Delay". In: *Commun. ACM* 55.7 (July 2012), pp. 42–50. ISSN: 0001-0782. DOI: 10.1145/2209249.2209264. URL: http://doi.acm.org/10.1145/2208917.2209336 (cit. on p. 16).

[22] Stefan Nilsson and Gunnar Karlsson. "IP-address lookup using LC-tries". In: *IEEE Journal on Selected Areas in Communications* 17.6 (June 1999), pp. 1083–1092. ISSN: 0733-8716. DOI: 10.1109/49.772439. URL: https://www.nada.kth.se/~snilsson/publications/IP-address-lookup-using-LC-tries/text.pdf (cit. on p. 17).

[23] Trevor Perrin. *The Noise Protocol Framework*. 2016. URL: http://noiseprotocol.org/noise.pdf (cit. on pp. 3, 7, 11, 12).

[24] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. RFC Editor, Jan. 2012. URL: http://www.rfc-editor.org/rfc/rfc6347.txt (cit. on p. 8).

[25] Keith Winstein and Hari Balakrishnan. "Mosh: An Interactive Remote Shell for Mobile Clients". In: *USENIX Annual Technical Conference*. Boston, MA, June 2012. URL: https://mosh.mit.edu/mosh-paper.pdf (cit. on p. 5).

[26] Xiangyang Zhang and Tina Tsou. *IPsec Anti-Replay Algorithm without Bit Shifting*. RFC 6479. RFC Editor, Jan. 2012, p. 9. URL: http://www.rfc-editor.org/rfc/rfc6479.txt (cit. on p. 13).