WireGuard Linux Kernel Integration Techniques

Netdev 2.2 — Seoul, Korea — November, 2017

www.wireguard.com

Jason A. Donenfeld

jason@zx2c4.com

Abstract

WireGuard is a formally verified secure tunneling protocol written and designed with the Linux Kernel as its primary operating context. Several design considerations at the protocol layer were considered in order to make it safe and simple to implement into an operating system kernel. In implementing WireGuard for Linux, several techniques are used for achieving performance and for doing cryptographic operations at the `struct net_device` level. This paper describes those techniques and protocol considerations, including a discussion of GSO, queueing, zero-copy, crypto API, FPU batching, multi-core algorithms, and socket routing semantics.

Keywords

wireguard, net_device, generic segmentation offload, queueing, multi-core algorithms, protocol design, kernel constraints, sticky socket routing

Preliminary

This paper presupposes quite a bit of knowledge of what WireGuard is, since other papers have covered that in detail already; [7] is readily available. This paper outlines a hodgepodge of kernel-specific implementation details relating to WireGuard, and is the textual companion of an in-person talk at Netdev 2.2.

Protocol Considerations

WireGuard is a secure networking tunnel written and designed originally for the Linux kernel [7]. While many protocols are developed by cryptographers in a vacuum, the WireGuard protocol was designed from the ground up to be *implementable* in a kernel setting, in a way that naturally yields defense-in-depth coding practices. While the cryptography is custom-tailored for WireGuard's needs, it is based on conservative cryptographic principles, and has been formally verified [8] using Tamarin [1].

Permanent ID of this document: `3e0a1a0ae12c6fcca0d251571444b7a9`. Static link: `wireguard.com/papers/wireguard-netdev22.pdf`. Date: November 8, 2017. This is draft revision `b023cb2`.

Memory Allocation

One objective of the protocol is to avoid allocations except at configuration time. WireGuard therefore allocates all the state memory it needs for the device and its peers during Netlink configuration. However, this imposes additional restrictions on the protocol: because no new memory can be allocated, existing memory must only be modifiable in relation to authenticated data. This in turn means that every packet must be authenticated, even the first, motivating the development of the Noise_IKpsk2 [10] 1-RTT handshake, which includes a static-static Diffie-Hellman calculation in the first message of the protocol, among others. However, putting a static-static Diffie-Hellman calculation in the first message opens the protocol up to CPU exhaustion denial of service attacks, and so a novel cookie mechanism, building on that of DTLS and IKEv2, is used, enabling WireGuard to fend off attacks without modifying statically allocated state memory. Without close coordination between the cryptography and the kernel considerations, the implementation would be required to utilize programming methods such as garbage collection and caches, which WireGuard entirely avoids.

Independent Handshakes

While older solutions such as IPsec placed the key exchange outside the kernel, WireGuard has chosen to collapse the key exchange and the transport layer into one extremely simple state machine. This greatly reduces the overall complexity, by removing the need for elaborate IPC. It also allows for sleek interactions between key exchange and transport, such as the timer state machine described in [7]. Through use of the Noise Protocol Framework [10], WireGuard achieves a very simple 1-RTT handshake using Curve25519 [5], an extremely fast elliptic curve. Because the handshake is only 1-RTT and because it is computationally inexpensive (in comparison to IKEv2, for example), it is possible to achieve forward secrecy by simply doing a new handshake with a session entirely independent from the prior session. This also enables a much simpler state machine and yields naturally to showing userspace a "stateless" interface, in which a device is either configured or not configured, but without any concept of a "connection".

Flow-agnostic Multicore Crypto

Traditionally, most network-layer encryption is per-flow single-threaded, in order to preserve packet ordering. While this delivers overall performance when many flows from many users are made in parallel, it performs poorly for large file downloads or for bandwidth intensive real time communications, such as video conferencing. WireGuard attempts to gain the best of both worlds—fast single flow performance and fast multiflow performance, while maintaining low latency.

Serial Queue, Parallel Queue

Each WireGuard device has global encryption and decryption queues. Each WireGuard peer has its own transmission and receptions queues. For outbound packets, via `ndo_start_xmit`, packets are first assigned a nonce, which also doubles as a sequence number, and are placed in the per-peer queue with an atomic flag set to "uncrypted". It is then added to the per-device queue. Next a kernel thread is started using `queue_work_on` to explicitly choose the CPU in round-robin. This thread processes items in the per-device queue until it is empty (which may never occur), and after each item is encrypted, it marks the packet flag as "crypted". It then calls `queue_work_on` on another work item, always set to run on the same per-peer transmission CPU, which simply dequeues items from the per-peer queue and transmits them, returning when there are either no more items to dequeue or when it attempts to dequeue an item that is in the state "uncrypted". Like this, packets are encrypted using all cores at once, but retain their order for their eventual non-parallel transmission. The ingress flow is more or less the same, with sequence number checking occurring at the end of the process, rather than the egress flow's nonce assignment at the beginning.

While this basic design is simple and achieves its goals, there are multiple sub-problems in implementing it. One option would be to design the parallel per-device queue by having one individual queue per CPU, and simply distributing packets round-robin amongst each per-CPU queue. This, in turn, would lend itself well to implementing the basic queue data structure using a linked list, since it is trivial to write a lock-free multi-producer single-consumer queue. However, a considerable downside is that it means the scheduler must visit all per-CPU queues in order to make consistent forward progress. If the scheduler lingers on one CPU, the per-peer transmission queue may very well be waiting to transmit packets that have not yet been encrypted on another CPU. In experiments with this approach, this was a massive source of latency.

Therefore a different approach was taken. Instead of having one queue per CPU, a global queue is used. This means the queue must be implemented using a ring buffer, because lock-free ring buffer algorithms are more efficiently implemented than lock-free linked-list queues. At the time of writing the `ptr_ring` structure is used, which does in fact use producer and consumer spinlocks, work is ongoing to make this lock-free. However, even with the spinlocks, performance is massively superior to the queue-per-CPU approach. If the scheduler is focused on one particular CPU, forward progress is still made on the per-device queue at large, and thus transmission remains low-latency and the latency does not increase in relation to the number of CPUs, as with a per-CPU queue.

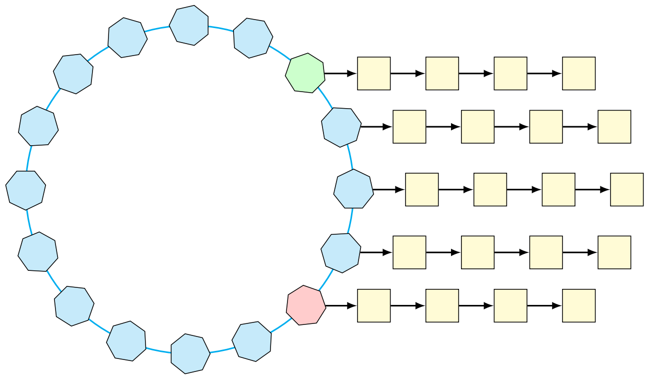

Figure 1: A ring of packet bundles, each of which is a linked list of GSO pieces.

Generic Segmentation Offload Batching

Whether the per-device queue uses a spinlock or is lock-free, contention is still always an issue, which means dequeueing and enqueueing must be batched. Furthermore, packets that arrive on a particular CPU often share similar caches, and if encryption must occur on a different CPU, it is best if the migration happens once and at the same time. It is thus evident that packets must be batched together in groups before they are added to the aforementioned queues. So it is question of how large should the batches be and at what interval should packets be bunched and released.

Fortunately, we do not have to actually answer that question or agonize over latency semantics of manual bunching. Instead we mark the `net_device` as supporting hardware based generic segmentation offload. This means that rather than pass the driver MTU-sized packets, we instead receive a very large "super-packet", usually around 65 kilobytes. We can then chunk that up itself, using `skb_gso_segment`, into MTU sized chunks, and group these packets together into a linked list. Each separate linked list of packet bunches is then put in our ring buffer, resulting in a ring of linked lists, as shown in figure 1.

On the receiving end, generic receive offload (GRO) will fit in evenly to this model, with clusters of packets from GRO being decrypted in bundles. This enables WireGuard to benefit from the generally fast single-threaded performance of modern processors while still receiving the overall speed boost from parallelism. Future work will involve modifying `skb_gso_segment` to take a dynamic MTU parameter, so that WireGuard can handle path MTU on a per-peer basis.

Context Batching

Due to the above semantics, it is possible to take care of all the packet transmission and reception in a single context, as well as all the encryption and decryption in a single context. For encryption and decryption, WireGuard makes heavy use of AVX instructions, using extremely high speed cryptographic primitives [2]. These require the use of the kernel FPU. While the ordinary crypto API generally turns the FPU on and off once per operation, we can simply turn it on and off once per the entire life cycle of the thread work item, saving valuable

CPU time ordinarily wasted for saving and restoring large vector registers.

Similarly for transmission and reception, we change into softirq context for the entire run of draining the per-peer queue. For reception, this enables us to call `netif_receive_skb` instead of `netif_rx`, which means we do not queue up packets to be delivered, but simply deliver them immediately. This causes the call to `netif_receive_skb` to block, slowing the ongoing dequeueing of packets. By not introducing another queue by way of `netif_rx`, we cause the per-peer queue to fill up instead when reception is a bottleneck. This then causes the enqueueing of new packets to fail when the per-peer queue is full. Counterintuitively, this is actually beneficial; it naturally pushes back on the per-device decryption queue by not adding more work to be done when the per-peer queue is full, lowering the amount of decryption work that would be discarded anyway later on in the pipeline.

Queue Lengths

At the moment, each queue is of a fixed length, but future work involves either changing this to use `struct dql`, in order to have dynamic queue lengths, or moving the entire `net_device` over to use qdiscs and in that way be able to employ `fq_codel` [9]. Early prototypes of WireGuard actually did make extensive use of subqueues, giving a subqueue to each peer, so that they could be started and stopped separately. However, since `fq_codel` already classifies based on flow and WireGuard's strong IP binding via the "allowed IPs" concept [7], a single queue will likely yield appropriate per-peer fairness, albeit with slightly different characteristics.

Granular Error Reporting

Being queueless, however, does have advantages: by skipping the qdisc subsystem, there is simply *less code* in the critical path, which usually is favorable from a performance perspective, and, more importantly, allows the WireGuard driver to return errors directly to userspace. For example, if a packet is transmitted by userspace to a peer for which there is no endpoint, the device can directly return `EDESTADDRREQ`, which then translates to an informative message to the user. While ICMP is also used—and is necessary in the case of forwarded packets that do not originate from userspace—the possible error codes from ICMP are less granular.

Socket Socket Routing

WireGuard does not enable the user to bind to a particular address. Instead, the user provides a listen port, which is then bound on all addresses. The underlying reasoning is that if a packet makes it to WireGuard and authenticates correctly via the cryptographic authenticator, Poly1305 [3], it does not matter by which interface or IP the packet entered. In fact, the means of ingress are most certainly *less trustworthy* than whether or not the authentication tag is correct. Poly1305 has a security proof [4], whereas IP packet addressing and physical cabling is manipulable by a variety of means.

However, since WireGuard does not allow the user to choose a bind address, it must determine the source address of packets automatically by itself. Additionally, WireGuard will reply to packets using the ingress source address as the egress destination address, which means the source address needed for a new possible destination is potentially dynamically changing.

When a packet arrives and is authenticated, the source address, destination address, and ingress interface of that packet are stored. When it comes time for transmission, the three pieces of information are used to construct a `flowi` for determining a routing table entry and final source address of the outgoing packet.

The source address corresponding to the ingress packet's destination is first checked to see whether it belongs to *any* interface on the system. While it may make sense to check instead of it belongs to strictly the WireGuard interface, many systems receive packets on one interface and transmit on another, or make use of dummy interfaces; thus we expand the check to be for all interfaces. If the source address does not belong to any interface, it and the egress interface are reset to zero, which indicates that the default routing choice should be made. Otherwise, it and the egress interface are used as values in the `flowi`, which is then passed to the routing table subsystem to find a route. If this fails, the failure propagates up and the packet is dropped only if either the egress interface or source address passed were zero. Otherwise, the two fields are zeroed, and the subsystem is re-queried. This helps keep the source address consistent as different interfaces come and go, which may share addresses or routes. For example, this logic does the right thing when switching from Ethernet to WiFi and back within the same network segment. At the same time, it does the right thing in complicated BGP routing scenarios, where packets may arrive and depart through drastically different routes. In this way, the socket routing is *sticky*, but is not overly sticky, in that it keeps up with network changes.

This sticky socket routing interface is at a middle point between being connection-oriented and connection-less, and maps in solidly with the overall design goal of appearing to be stateless to userspace, while perfectly managing all state transparently and making the most precise decisions.

Network Namespaces and Policy Routing

Like most Linux network interfaces, WireGuard integrates into the network namespace infrastructure. WireGuard does something a bit special with namespaces, though. When a WireGuard interface is created, it remembers the namespace in which it was created. Later, the interface can be moved to new namespaces, but it will still remember the namespace in which it originated.

WireGuard uses a UDP socket for actually sending and receiving encrypted packets. This socket always lives in the original birthplace namespace of the interface, not any new namespace to which the interface might be moved. This has several useful properties. One can create a WireGuard interface in one namespace, move it to another, and have cleartext packets sent from the latter namespace be transmitted encrypted through a UDP socket in former namespace. The most obvious usage of this is to give containers WireGuard interfaces as their sole means of networking. A less obvious usage is to use this characteristic of WireGuard for redirecting all ordinary Internet traffic over WireGuard, by having a WireGuard interface be the only interface in the `init_net` namespace, with its UDP socket living in a newly created "physical" namespace to where the actual physical Ethernet interfaces are moved.

The downside of this approach is that moving physical interfaces between namespaces means bringing them down and killing and restarting daemons that might be using them directly, such as `wpa_supplicant` ownership of `wlan0`. There are two current approaches for routing Internet without namespaces.

The first is by adding an explicit route to the tunnel endpoint using the physical interface and then overriding the default route with the WireGuard interface. The second is by using policy routing. WireGuard allows users to set the `fwmark` on outgoing encrypted UDP packets. This can then be used with `ip-rule` and its `suppress_prefixlength 0` directive to achieve the same thing as overriding the default route, but more reliably. The `fwmark` can also be used by Netfilter for a variety of uses, such as implementing a so-called "kill-switch" by disallowing packets that are not from either the WireGuard interface or are without the `fwmark`.

While this can be automated with a good deal of success, it is however complicated and a bit cumbersome to use. And routing table rules are not bound to any interface, which means when the WireGuard interface is deleted, these rules must manually be cleaned up. A proposed alternative is `not_oif` [6], a member of `flowi` and a corresponding socket option, `SO_NOT_OIF`, which would be a hint to the routing table to return routes that assume the passed interface index does not exist. This would eliminate circular routing and would allow for very basic and straight-forward routes that actually do directly what the user intended, without having to follow a roundabout process of applying routing rule policies.

Crypto API Improvements

WireGuard has a fixed cipher suite. It is a throughly designed protocol, rather than a generalized pluggable architecture. This has numerous security benefits and is considered by modern standards to be a best practice. However, it is a considerable departure from the heavily abstracted crypto API currently in use in the kernel. WireGuard also endeavors to use formally verified implementations of primitives, work for which is ongoing. The FPU context batching, too, requires semantics outside the current crypto API. WireGuard tries to avoid allocations, and the Noise handshake does many encryptions with differing keys; the current crypto API does not do well with changing keys and avoiding allocations. And finally, WireGuard focuses on using extremely simple and direct interfaces, instead of bug-prone heavy boilerplate code. For these reasons, WireGuard currently uses its own internal crypto API with its own primitives, but it is planned to merge these improvements back into the crypto API in order to improve the entire kernel crypto ecosystem.

Netlink and In-Band Messaging

WireGuard uses Generic Netlink for configuration between the user and the kernel. However, between peers themselves, all IP assignment and configuration is static. There is no push configuration, nor should such a thing be in the kernel. WireGuard's key exchange and configuration model is intentionally out-of-band.

However, it is sometimes useful to send messages in-band before IPs are known, directly to a peer, addressable by the peer's public key. There are two possible ways to achieve this moving forward. One would be to implement a Netlink command to send a buffer directly to a peer's public key, not encapsulated in an IP packet. On the receiving end, a Netlink multicast event would be triggered. Alternatively, a new address family, `AF_WIREGUARD` could be created, where `sendto` and `recvfrom` take a `struct sockaddr_wg` for a 32-byte peer public key. While this has the benefit of being more elegant, adding a new address family must be very carefully considered. And since this is meant to be used for application-specific configuration, not for general data transfer, perhaps Netlink is more appropriate. However, using an address family would allow for all kinds of interesting and useful unforeseen uses of WireGuard.

Concluding Remarks

Designing a tunneling protocol from the ground up for the facilities of the Linux kernel leads to interesting and unique possibilities, excellent performance, and stronger security semantics. This talk addresses some of those kernel puzzles that were part of the design process.

References

[1] *Tamarin Prover Manual*. URL `https://tamarin-prover.github.io/manual/`.

[2] Daniel J. Bernstein. Cpus are optimized for video games. URL `https://moderncrypto.org/mail-archive/noise/2016/000699.html`.

[3] 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*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005. doi: 10.1007/11502760_3. URL `https://cr.yp.to/mac/poly1305-20050329.pdf`. Document ID: `0018d9551b5546d97c340e0dd8cb5750`.

[4] Daniel J. Bernstein. Stronger security bounds for wegman-carter-shoup authenticators. 2005. URL `https://cr.yp.to/antiforgery/securitywcs-20050227.pdf`. Document ID: `2d603727f69542f30f7da2832240c1ad`.

[5] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, Berlin, Heidelberg, 2006. Springer-Verlag Berlin Heidelberg. ISBN 978-3-540-33852-9. doi: 10.1007/11745853_14. URL `https://cr.yp.to/ecdh/curve25519-20060209.pdf`. Document ID: `4230efdfa673480fc079449d90f322c0`.

[6] Jason A. Donenfeld. Inverse of flowi{4,6}_oif: flowi{4,6}_not_oif. URL `http://lists.openwall.net/netdev/2016/02/02/222`.

[7] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *Proceedings of the 2017 Network and Distributed System Security Symposium*, NDSS'17, San Diego, CA, 2 2017. ISBN 1-891562-46-0. doi: 10.14722/ndss.2017.23160. URL `https://www.wireguard.com/papers/wireguard.pdf`. Document ID: `4846ada1492f5d92198df154f48c3d54205657bc`.

[8] Jason A. Donenfeld. Formal verification of the wireguard protocol, 2017. URL `https://www.wireguard.com/papers/wireguard-formal-verification.pdf`. Document ID: `d376f649d7f4b68f616e05e5f64d660e9b23d7af`.

[9] Toke Hoeiland-Joergensen, Paul McKenney, Dave Taht, Jim Gettys, and Eric Dumazet. The flowqueue-codel packet scheduler and active queue management algorithm. Rfc, Internet Engineering Task Force, 03 2016. URL `https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06`.

[10] Trevor Perrin. The noise protocol framework, 2016. URL `http://noiseprotocol.org/noise.pdf`.