A Control Plane for WireGuard

Jordi Paillisse*†, Alejandro Barcia*, Albert Lopez*, Alberto Rodriguez-Natal†, Fabio Maino†, Albert Cabellos*
*Computer Architecture Department, UPC-BarcelonaTech, Barcelona, Spain
Email {jordip, alopez, acabello}@ac.upc.edu, alejandro.barcia@estudiant.upc.edu
†Cisco, San Jose, CA, USA. Email: {natal, fmaino}@cisco.com

***Abstract*—WireGuard is a VPN protocol that has gained significant interest recently. Its main advantages are: (i) simple configuration (via pre-shared SSH-like public keys), (ii) mobility support, (iii) reduced codebase to ease auditing, and (iv) Linux kernel implementation that yields high performance. However, WireGuard (intentionally) lacks a control plane. This means that each peer in a WireGuard network has to be manually configured with the other peers' public key and IP addresses, or by other means.**

In this paper we present an architecture based on a centralized server to automatically distribute this information. In a nutshell, first we manually establish a WireGuard tunnel to the centralized server, and ask all the peers to store their public keys and IP addresses in it. Then, WireGuard peers use this secure channel to retrieve on-demand the information for the peers they want to communicate to. Our design strives to: (i) offer a key distribution scheme simpler than PKI-based ones, (ii) limit the number of public keys sent to the peers, and (iii) reduce tunnel establishment latency by means of an UDP-based protocol. We argue that such automation can help the deployment in enterprise or ISP scenarios. We also describe in detail our implementation and analyze several performance metrics. Finally, we discuss possible improvements regarding several shortcomings we found during implementation.

***Index Terms*—dynamic VPN, wireguard, secure overlays, control plane**

I. Introduction

Virtual Private Networks (VPN) are one of the most pervasive services offered in the Internet. As such, VPNs are considered a mature technology with well-understood mechanisms and widely deployed and accepted protocols. Notable examples of current VPNs are OpenVPN [1], IKEv2 (RFC 4306), or IPSec (RFC 4301).

Surprisingly, WireGuard (WG, [2]) is a recent VPN protocol that has managed to disrupt this mature field [3]–[6]. One of the main reasons behind WireGuard's success is that it tradeoffs flexibility for simplicity. As opposed to traditional VPNs that support a large set of cipher suites and that negotiate its capabilities before establishing the secure connection, WireGuard only supports one cipher suite, specifically -at the time of this writing- ChaCha20 and Poly1305 (RFC 8439). This greatly simplifies the architecture as well as the code, providing also important performance advantages.

In short, in WireGuard users only have to exchange the public keys of the endpoints and the IP address of one of them to start exchanging data. In fact, the setup is very similar to that of SSH. The underlying implementation takes care of the cryptography aspects of a typical VPN: key derivation, re-keying timers, anti-replay protection, etc. In addition, WireGuard supports mobility and sends keepalives to maintain NAT holes open.

With its ease of deployment, native mobility support and performance, WireGuard is gaining popularity and it is becoming the *de facto* VPN standard for certain Internet communities. However, WireGuard lacks some critical features to enter Internet professional scenarios, such as ISPs or Enterprise networks. On the other side, end users that would like to add confidentiality to their connections usually have to rely on deploying IPsec in the underlay, with its associated complexity.

One of the main limitations is that WireGuard lacks a control plane. Hence, users have to manually setup the public keys and IP addresses in all of their endpoints, i.e. when we add a new device to the network we have configure its public key in *all* the existing devices. This process is time-consuming and error-prone.

Taking this into account, in this paper we aim to design, prototype and evaluate a control plane for WireGuard. Specifically, we focus on three key points. First, automate the distribution of WireGuard keys *without* relying on PKI-based schemes or DNS-based schemes (e.g. IETF DANE [7], [8]). These systems require complex configuration or additional infrastructure like a DNS server. On the other hand, although there exist commercial systems that can offer equivalent functionality, such as Dynamic Multipoint VPN [9], or modern SD-WAN systems [10], their details are not public.

Second, reduce as much as possible the number of public keys stored in the endpoints, in order to reduce communication overhead and avoid sharing unnecessary cryptographic information. Also, we want to be as fast as possible when creating new tunnels. And third, understand and pinpoint other potential limitations of the WireGuard protocol to be considered for Enterprise or ISP scenarios.

The proposed control plane works as follows: we store the public keys of all peers in a central server, that sends them to the WireGuard peers upon request (fig. 1). This way, users only

This work was partially supported by the Spanish MINECO under contract TEC2017-90034-C2-1-R (ALLIANCE) and the Catalan Institution for Research and Advanced Studies (ICREA).

Cite as: J. Paillisse, A. Barcia, A. Lopez, A. Rodriguez-Natal, F. Maino, and A. Cabellos, "A control plane for wireguard," in 2021 International Conference on Computer Communications and Networks (ICCCN), 2021, pp. 1–8, doi: 10.1109/ICCCN52240.2021.9522315.

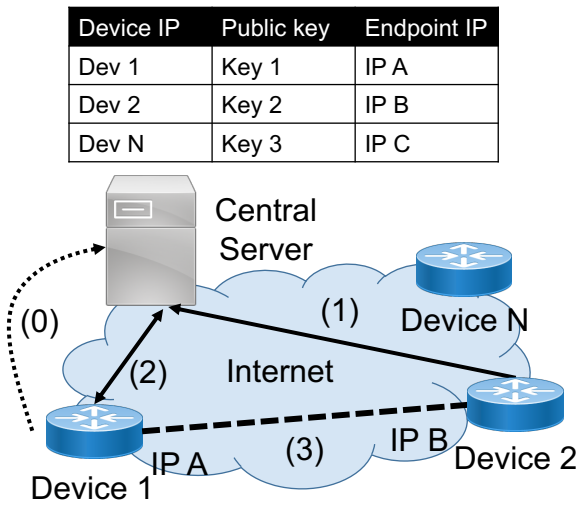

Figure 1. Global design. Dashed lines represent data plane traffic flow, solid lines control plane traffic.

have to setup a secure connection between the peer and the central server on bootstrap (0); new peers store their public key and endpoint IP address in the server (1). Afterwards, peers retrieve any information from the central server (2) and can establish WireGuard tunnels with this information (3).

We must remark that the idea of storing public keys in a centralized server is not new, but rather a well-know topic (sec. II-A). On the contrary, the contributions of this paper lie on the practical implementation side, specifically: (i) the design caveats of a centralized key distribution control plane, (ii) the selection of a protocol to reduce the query latency when requesting a public key, (iii) implementation details, and (iv) a performance evaluation. To the best of our knowledge, this is the first control plane for WireGuard-based VPNs. Finally, we plan to open-source this control plane implementation.

II. Related Work

A. Key Distribution

The problem of storing public key data in a centralized server, and distributing it to establish secure tunnels is a well-know topic and has been previously explored, such as in Kerberos [11], or the IETF DNS-Based Authentication of Named Entities (DANE [7], [8]), that makes it possible to add different types of public keys in DNS records like IPsec keys (RFC 4322). Furthermore, there is extensive research in dynamic VPN configuration and management [12]. More recently, SDN-inspired solutions propose to configure IPsec endpoints in a centralized fashion [13], as well as the widespread usage of Software-Defined WANs that automatically setup IPsec tunnels [10]. The most closely related work is a theoretical analysis to configure WireGuard with OpenFlow extensions [14]. However, it does not provide an implementation or performance data, rather focusing on the design of control plane messages.

Table I
Comparison of common dataplane encryption protocols

Protocol	NAT-trav.	Mobility	Overhead (Bytes)
IPsec tunnel ESP	×	×	28
OpenVPN DTLS	√	×	38
WireGuard	√	√	38

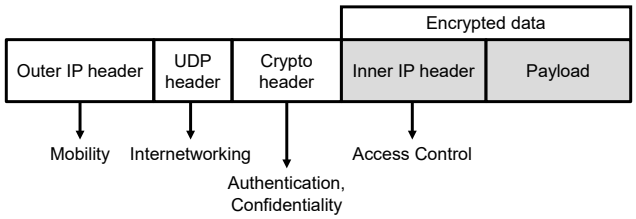

Figure 2. WireGuard header structure

B. Secure Data Planes

Table I compares several features of two of the most popular L3 VPN protocols (IPsec and OpenVPN), and WireGuard, along with the overhead of their headers. We can see that WireGuard supports both NAT traversal and mobility, with the same overhead of OpenVPN with DTLS. OpenVPN does not support mobility but can deal with NAT with additional configuration [15]. Finally, IPsec does not offer NAT traversal nor mobility but incurs in a smaller overhead. However, we must note that IPsec can handle both NAT traversal and mobility with additional extensions: IPsec over UDP, and the MOBIKE extension (RFC 4555), respectively.

Regarding other data plane encapsulations, such as VXLAN (RFC 7348), VXLAN-GPE [16], GENEVE [17] or ILA [18], it is interesting to remark that they do not usually consider security or mobility (with the notable exception of ILA mobility), as opposed to the protocols in table I.

III. Background: WireGuard

From a network architecture perspective, WireGuard adds two additional headers to a standard IP datagram (fig. 2). First, an outer IP header, that contains the peer IP address. This header supports mobility; its IP address can change arbitrarily as the peer roams across different networks (e.g Wifi, LTE). Afterwards, there is a UDP header plus a custom header that contains the message type, cryptography information, and the encrypted payload. Inside the payload we find the inner IP header, the one used by applications. This header is used on the receiving end to perform access control, i.e. we can specify a list of allowed IP prefixes.

The key advantage of this architecture is that the three headers are independent: we can modify any of them without impacting the others. For example, in case that a peer changes its IP address due to a mobility event, the crypto header and the encrypted data will be successfully decrypted in the destination. In summary, each header performs a different task:

Outer IP header: mobility, can change freely
Crypto header: authentication, confidentiality

Table II
SAMPLE WIREGUARD CRYPTOKEY ROUTING TABLE

Allowed Source IP	Public key	Internet Endpoint
10.10.0.0/16, 10.11.0.0/16	Peer A key	80.80.80.80
172.16.1.0/24, 172.16.2.2/32	Peer B key	100.128.128.128
192.168.4.0/24	Peer C key	40.0.0.0

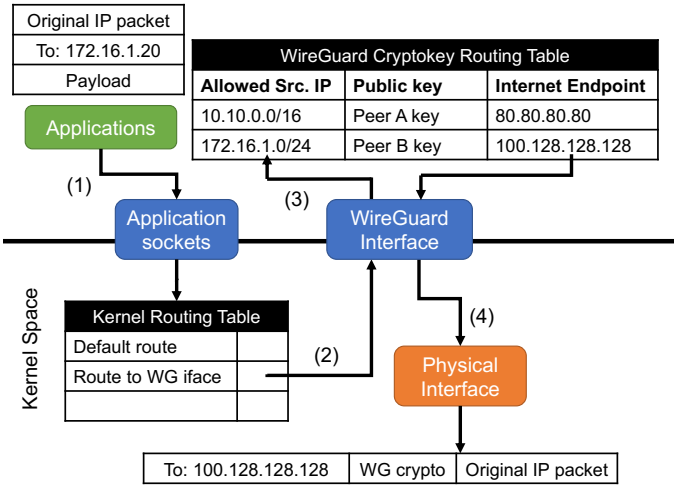

Figure 3. WireGuard packet transmission

Inner IP header: access control

The key element of WireGuard's operation is the *cryptokey routing table*, that binds source IP addresses (usually IP addresses in the private range) to peer public keys (table II). In other words, the source IP is used to determine the encryption key and the receiving peer. Additionally, it stores the Internet IP address of the peer, that is used in the outer IP header. Figure 3 presents the packet flow of an outgoing WireGuard packet. First, users configure the cryptokey routing table with the peers and adjust the Linux routing table to forward this packets to the WireGuard interface (2). This way, new packets destined to the peers (1) are forwarded to the WireGuard interface (2, 3). The interface does a reverse IP lookup on the cryptokey routing table to find the peer's key and Internet endpoint. Using this data, it encrypts, encapsulates and sends the packet to the physical interface (4).

Upon reception, the previous process is reversed. First packets are decrypted. WireGuard uses a local identifier (equivalently to IPsec) to match incoming packets to the their security association. If decryption is successful, the WireGuard interface performs access control. Specifically, it verifies that the source IP address of the packets is in the list of IP prefixes of the associated public key. This determines if packets are accepted or dropped. Finally, they are forwarded back to the kernel routing table, and in turn to the corresponding application socket.

IV. ARCHITECTURE

A. Threat Model

For this specific use-case, either for end users or enterprise networks, we assume we can trust both the endpoints and the centralized server, i.e. we consider legitimate any control plane message they send. On the other hand, the underlying network is not trustworthy.

In order to protect the signaling between clients and server, we establish a WireGuard tunnel between them (sec. IV-C). The keys for these tunnels are part of the configuration stage and are exchanged out of band. This way client and server have a way to authenticate each other. In addition, the centralized server stores only the endpoint location (Internet IP address), and its corresponding public key. These assumptions raise a set of security concerns in both the centralized server and the endpoints that we discuss below.

1) Central server:

- **Single point of failure:** using a centralized server creates a single point of failure, especially vulnerable to DDoS attacks. We can mitigate this risk with: (i) replicating the central server, and (ii) since the centralized server uses WireGuard tunnels, they include an anti-DDoS mechanism, detailed in sec. 5.3 of [2]. This mechanism is based on a cookie from the server that the client has to re-send when establishing a connection.
- **Leak of peer locations and/or public keys:** in this situation, although an attacker can initiate connections with the rest of the peers, it won't be able to finalize the handshake because the peers won't receive the attacker's public key from the central server (sec. IV-C). Moreover, in case of compromise it is possible to remove the affected key from the central server so the rest of the peers cannot establish communication.
- **Leak of the private key of the central server:** this is the worst scenario, since an attacker can impersonate the central server and mount a monkey-in-the-middle-attack. The secrecy of the private key depends on the OS security precautions.

2) Endpoints:

- **Leak of peer locations and/or public keys:** similarly to the central server, this leak does not allow an attacker to connect to any peer, because it will not be able to successfully authenticate in the central server. It *may* be able to take advantage of peers that that previously stored its public key in their cache, though.
- **Leak of the private key of the peer:** on the contrary, here the attacker will be able to impersonate the peer when connecting to the central server, retrieve connection data and successfully establish connections to the rest of endpoints. Likewise, it is possible remove this key from the central server to prevent future connections.

B. Discussion

The key architecture decision we took was the state distribution model: send all the keys to all peers (*push* model) versus

each peer retrieving only the keys it needed (*pull* model).

We chose the pull approach for two reasons. First, security, because we avoid sharing all the public keys with all the peers. Second, minimizing the amount of state in the data plane. Since we only download the keys we need, the WireGuard cryptokey routing table is less crowded. In turn, this reduces control plane signalling overhead and increases scalability. We must remark that we are assuming that the traffic pattern among the peers is *not* full mesh, i.e., peers do not need *all* the existing keys, but just a bunch of them. On the other hand, the main drawback of a pull architecture is an increased connection establishment time. In order to minimize this delay, we chose an UDP-based protocol to retrieve the keys. Section VI-A quantifies this delay.

C. Architecture Description

In a nutshell, our architecture lays two elements on top of a WireGuard deployment (fig. 1). First, a centralized database that contains, for each WireGuard peer, its associated public key and endpoint IP address. Second, a secure control channel between the centralized database and the WireGuard peers, that we use to retrieve the aforementioned WireGuard configuration (solid lines). The central database indexes peers by their inner IP address, i.e. the one used by applications and encrypted by WireGuard.

The centralized database distributes the configuration data on demand (fig. 1): when a peer does not have the public key for a particular destination IP (i.e. we don't have an entry in the WG crypto table), we request the public key and endpoint IP via the control channel, and configure WG appropriately (step (2) in fig. 1 in order to communicate with Device 2). The central server answers this request but also pushes Device 1 public key and Endpoint IP to Device 2. This is due to the fact that, in order to successfully establish a new WireGuard tunnel, both peers need to have each others' public key. In other words, we use a pull-and-push approach: the node that initiates a connection pulls the key, while we push the key to the receiving node. Finally, once both peers have each other's data, they can establish the WireGuard tunnel (3, dashed line).

Taking this into account, we defined the following messages for the control channel to interact with the database: (i) Store a peer public key + endpoint IP, (ii) Request a peer public key, and (iii) Send a public key to a peer.

V. Implementation

In order to implement the protocol for the control channel, we leveraged the control plane part of the Locator/ID Separation Protocol (LISP, RFC 6833). LISP is a mature and standardized protocol, designed to dynamically create network overlays. This makes it an excellent candidate for this application, because LISP's control messages present nearly a 1:1 match to our requirements (table III). Moreover, the LISP architecture includes the Mapping System, a centralized server that stores pairs of overlay to underlay IP addresses. However, it is possible to adapt other SDN southbound protocols such as P4runtime or OpenFlow [19] for this use-case, or even use HTTP-based interfaces, like REST (RFC 7231), WebSocket (RFC 6455) or gRPC [20]. However, we chose to use LISP due to performance concerns: since LISP runs on UDP, it offers lower latency when compared to TCP and HTTP-based protocols. We consider the control plane latency relevant for our use case since it is in the critical path to establish new connections (sec. VI-B compares such delay with gRPC). On the other hand, we must remark that the aforementioned interfaces offer more extensibility than LISP.

Table III
Equivalence of sec. IV messages and LISP messages

Design message	LISP message
Store key	Map Register
Request key	Map Request
Send key	Map Notify

Our prototype is based on an open-source LISP implementation, Open Overlay Router (OOR, [21]). OOR is implemented in C and works in Linux user space, which makes it easy to implement new features [22]. In a nutshell, we modified OOR to: (i) detect flows that do not have an active WireGuard tunnel configuration, (ii) request its associated public key with the aforementioned control plane messages, and (iii) configure accordingly the WireGuard interface.

A. Endpoints

Fig. 4 presents a diagram of our implementation. First, we configure a WireGuard interface to tunnel all control plane packets to the central server (i.e. server endpoint IP and pubic key). We use a statically defined IP prefix to identify and route such requests through this interface. We also create a TUN interface and add a default route in the kernel routing table pointing to it. The purpose of the TUN interface is allowing OOR to connect, capture packets and examine their destination IP address. If the IP address is not in its local database, OOR will send a LISP Map Request to the central server to retrieve the key. We must remark that the OOR database is a copy of the WireGuard Cryptokey Routing table. Once it receives this information, OOR configures the new peer in a second WireGuard interface that handles only data plane traffic. Note here that the prototype uses two WireGuard interfaces, one for data plane traffic, and another for control plane traffic. OOR also adds a more specific route in the kernel routing table pointing to this second WireGuard Data Interface. New traffic for this prefix will bypass the TUN interface and go directly to the WireGuard Data interface. This is especially relevant since it avoids copying packets from kernel to user space and viceversa, significantly improving performance.

In case a peer does not want to tunnel all its traffic through the dynamically-created WireGuard tunnels (split tunneling), it can bypass them by adding more specific routes in the kernel routing table pointing to an alternative destination.

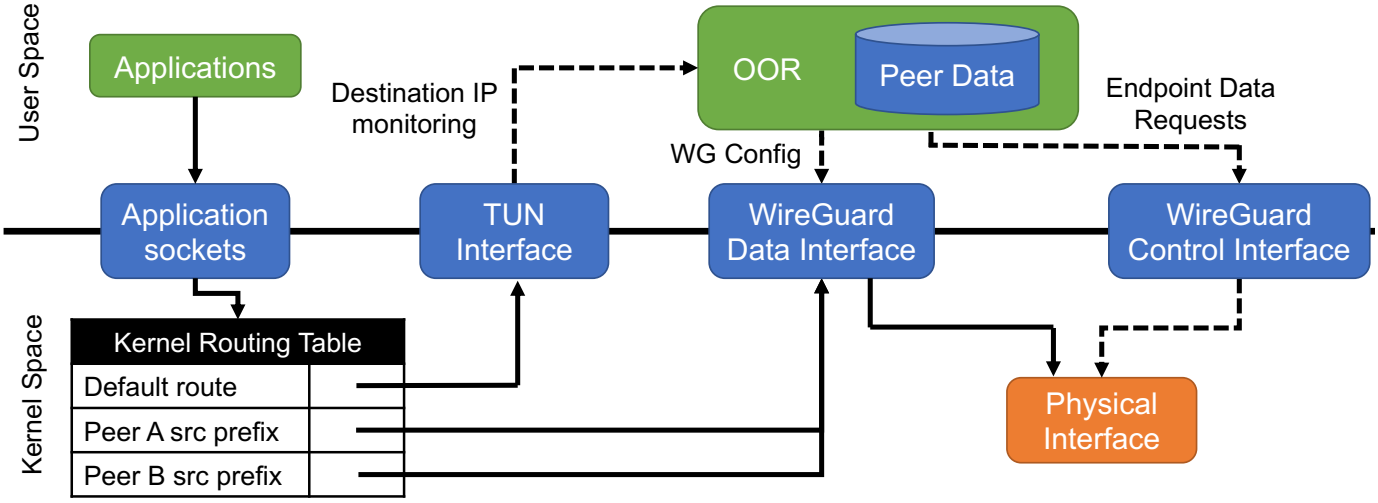

Figure 4. Simplified prototype operation diagram. Solid lines represent data plane traffic flow, dashed lines control plane traffic.

B. Central Server

We implemented the aforementioned centralized server as a LISP Mapping System, more specifically a single Map Server. In other words, all peers send their requests to the same Map Server. This Map Server is running a modified version of the OOR Map Server, in which incoming Map Requests trigger a Map Notify to the destination WireGuard peer. This way, the destination peer receives the sender key and the tunnel can be established. In order to carry the peer public key in the LISP messages, we leveraged the LISP Canonical Address Format, LCAF (RFC 8060), an extension of such protocol that allows adding extra data in the Map Request/Map Reply messages (LCAF 11 - security key).

Finally, in case a peer wants to update its public key, we can easily update this information with a LISP Map Register message.

VI. Evaluation

We evaluated different metrics of our implementation in order to quantify the overhead of adding security (i.e. the WireGuard cryptography), as well as the delay of the pull architecture. Since OOR also implements the LISP data plane, i.e. it can create unencrypted network overlays, we took OOR's performance (not modified) as a baseline. We carried out all tests in a lab setup with Gb Ethernet interfaces, and we considered the network delay negligible. The servers were running Linux Ubuntu 16.04 on an Intel Xeon E5-2650 v4 @ 2.20 GHz.

A. End-to-end Delay

Fig. 5 presents the CDF of the delay to establish a tunnel, i.e the time since we send a probe packet until we receive the first response. The setup consisted of two instances of our implementation in the same LAN, directly connected with each other and also to the central server. We setup a data rate of 10000 pings/s to have an adequate time resolution. The measured time includes all the control plane signaling and

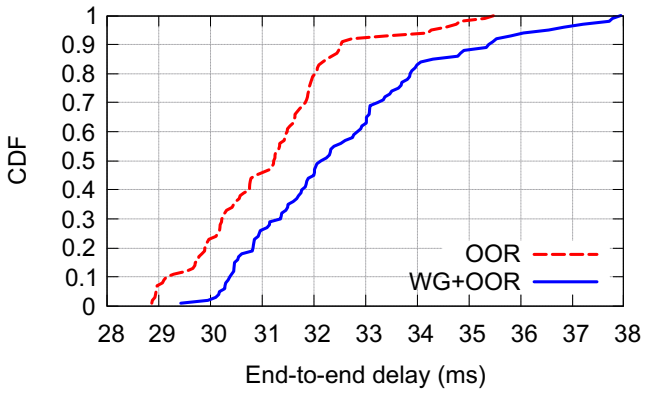

Figure 5. Tunnel Establishment end-to-end delay CDF

the establishment of the WireGuard tunnels (for OOR+WG) or the LISP tunnels (for OOR). We repeated this experiment 100 times. We can see that WireGuard adds an extra delay of approximately 1 ms due to the cryptography. We consider this delay acceptable, taking into account that we are comparing an unencrypted connection (OOR) and an encrypted one (OOR+WG). We must remark that OOR packets go through user space, while in OOR+WG they stay in kernel space. The former adds an extra delay because data packets are copied from kernel to user space and vice-versa (fig. 6).

B. Control Channel Performance

We evaluated the performance of our control plane implementation and compared it to OOR with LISP data plane, and a prototype that uses gRPC instead of LISP to retrieve the public key. The prototype is based on a C version of gRPC [23], and uses a WireGuard secure tunnel to communicate with the central server, like the OOR+WG prototype. Figure 7 presents the CDF of the end-to-end delay to add data for a new endpoint, or connection establishment time. We measured the time it takes to retrieve the public key for a new endpoint

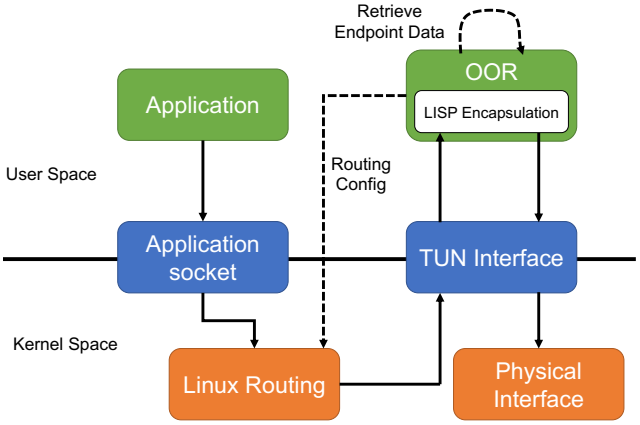

Figure 6. Out of the box OOR implementation diagram. Solid lines represent data plane traffic flow, dashed lines control plane traffic. As opposed to the OOR+WG implementation in fig. 4, data plane traffic is copied between kernel and userspace, adding overhead.

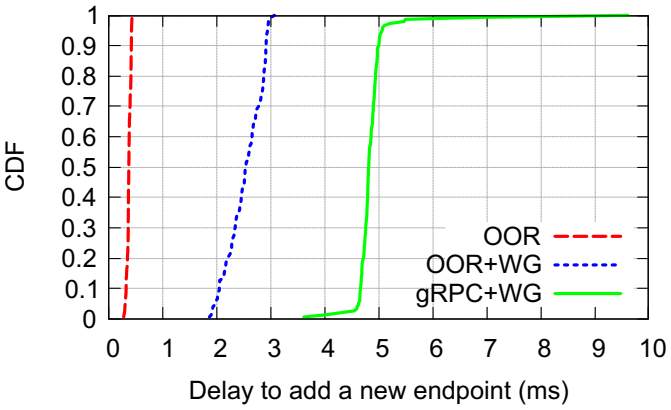

Figure 7. Delay to request and configure a new endpoint

from the central server. We repeated the experiment 150 times for the three scenarios. We can appreciate that OOR with WireGuard takes approximately 1.5 ms more than OOR to complete this operation. This is due to the fact that we are retrieving more information (WireGuard needs the public key, apart from the endpoint IP address) and the overhead of configuring the WireGuard interface (as opposed to OOR, in which we can start sending packets directly). Nevertheless, since the WireGuard public keys are 32 bytes, the performance penalty of downloading this extra information is negligible compared to typical Internet latency [24]. On the other hand, the gRPC prototype presents a delay double to OOR+WG, due to the overhead of its HTTP/2 transport.

C. Central Server Scalability

We also evaluated the scalability of the centralized server. Figure 8 presents the delay to answer a request for endpoint data for both OOR and OOR with WireGuard, depending on the number of elements in the central server. Specifically, we measured the processing delay of the central server, i.e. the time since it receives a data request until it generates the corresponding response. To perform this measurement,

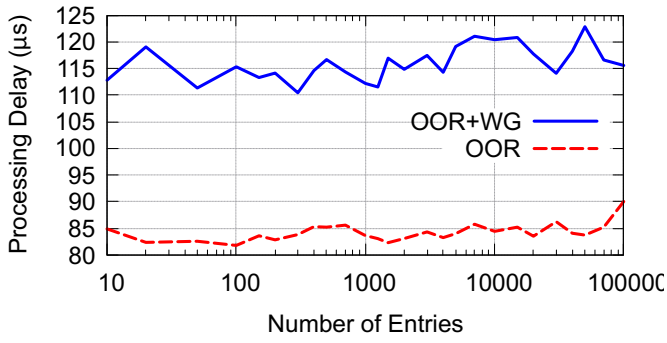

Figure 8. Central Server Response Delay

we artificially generated LISP messages towards the central server using LIG, a tool that can create different kinds of LISP messages [25]. Note that we are measuring the performance of the central server, in other words, both OOR and OOR+WG messages were unencrytped and there were no tunnels between the central server and the machine sending the requests. We repeated this experiment 50 times for each number of elements in the server. We can appreciate that the delay does not depend on the number of elements because the server implementation uses a Patricia Trie [26]. In other words, the maximum number of peers a single server can handle will be limited by its CPU, not the database size. Additionally, this experiment quantifies the overhead of adding a public key in the central server database. We can see that it increases the delay on average 40 μs. Again, this modest increase is due to the aforementioned extra 32 bytes.

D. Handover Delay

Regarding mobility events, figure 9 presents the handover delay for OOR and OOR+WireGuard. We modified the setup of sec. VI-A by connecting one of the two peers via WiFi through two possible access points (fig. 10), and we triggered handovers by manually changing the access point of this peer. We ran a ping of 10 requests/ms between the two machines and measured the amount time without ICMP replies. We repeated this experiment 20 times. This test shows that WireGuard presents a handover delay nearly one order of magnitude less than OOR. This is due to the fact that the WireGuard handover: (i) operates in kernel space, and (ii) it does not require control plane signaling, as opposed to OOR, that issues a new Map Request / Map Register in this situation.

From an architecture point of view, we observe that adding a security association simplifies mobility, because we can learn new endpoints through the data plane with virtually no signalling. In addition, if the endpoint IP address is spoofed, the payload is protected by the WG crypto header. On the other hand, this technique does not cover corner cases such as double jump. This is in line with the design philosophy of WireGuard, that sacrifices flexibility for simplicity.

E. Data Plane Performance

Finally, we measured the data plane throughput. Fig. 11 presents the input and output rate for our OOR+WireGuard

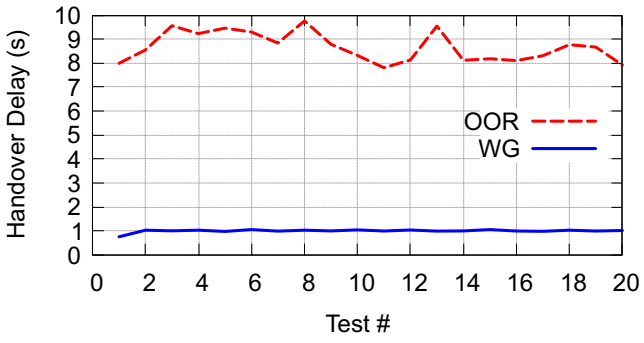

Figure 9. Handover delay

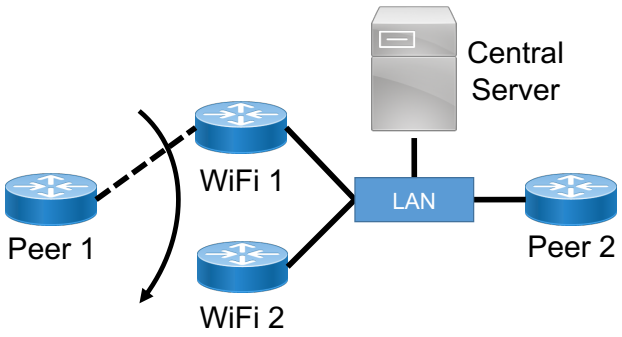

Figure 10. Handover evaluation setup

implementation, IPsec, and out-of-the box OOR. In the same setup as in sec. VI-A, we configured a tunnel between the two servers, and we used the `nuttcp` utility to determine real throughput between them for several input rates, and for each protocol. We sent UDP packets, and adjusted the lengths of the send/receive buffers taking into account the size of the different headers of each protocol, in order to fill in the packets but avoid L3 fragmentation. We configured IPsec in tunnel mode with the same encryption algorithm as WireGuard (ChaCha20 and Poly1305). We can see that both OOR+WG and IPsec offer similar performance, and start to degrade gracefully around 900 Mbps. On the other hand, the OOR version that uses the LISP data plane degrades abruptly at 800 Mbps due to the overhead of copying form kernel to user space.

VII. Lessons Learnt

During the implementation and evaluation of our prototype, we found a few shortcomings in WireGuard. Some are especially relevant for enterprise (Virtual Networking) and ISP (mulithoming) scenarios. In this section we comment on them and outline possible approaches. Some of them can be addressed with our proposed control plane, others require modifying the WireGuard data plane.

A. Virtual Networking

It is common to support several instances of the same addressing space, especially in enterprise environments. The most common solution is adding an extra field in the headers

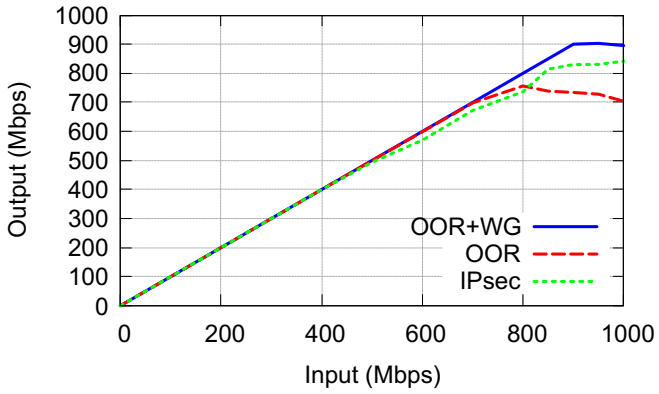

Figure 11. Throughput comparison

that acts as identifier, e.g. VXLAN Network Identifier [27]. Since WireGuard packets do not include a Virtual Network Identifier, it is not possible to make this distinction. However, it can be accomplished by adding this field to the WireGuard header, or adding a VXLAN header before sending packets to the WireGuard interface, for instance. We believe this is a relevant feature that could be considered among the WireGuard community.

B. Multihoming

Our implementation cannot send encrypted traffic from multiple Internet endpoints at the same time (i.e. multihoming). The main reason is that WireGuard only supports a single endpoint IP at any moment in time. If the endpoint IP changes it is interpreted as a mobility event, not traffic coming from different sources. A possible solution consists in leveraging the control plane channel to advertise the set of possible Internet endpoints to both peers. However, this means that the WireGuard data plane cannot update the Internet endpoint of a peer when it receives a packet with a different IP, thus making mobility more complex. This feature is especially interesting in the context of increased mobile device usage, e.g. WiFi and LTE interfaces in a smartphone, or ISPs that offer VPN solutions.

C. Mobility Double Jump

In case two peers change their Internet endpoint at the same time, it is not possible to re-establish connection only with data plane mechanisms. However, thanks to our control plane we can account for this situation. When a peer detects it has not received packets after some time, it sends a new key request to the control plane. Since the other peer will have updated its new endpoint in the control plane, the former peer will receive the new endpoint IP and communication will resume.

VIII. Conclusion

In this paper we have presented a control plane for nodes using the WireGuard VPN. Such control plane stores the WireGuard public keys in a centralized server and distributes them on-demand. The proposed architecture offers two main advantages: (i) automates the configuration of WireGuard

tunnels, and (ii) reduces state in the data plane by creating security associations only if a particular tunnel is required. This control plane can be used to automate deployment, especially for enterprise or ISP scenarios with a large amount of peers. We also took advantage of WireGuard to secure the control plane channel between the nodes and the centralized server. In addition, we have discussed the main challenges, such as pushing the sender's public key to the destination, and possible additions like virtual networking or multihoming, in order to support the aforementioned use cases. Finally, we have presented the design of our implementation, an evaluation of the overhead of such control plane, and performance measurements to validate that we retain the throughput and mobility benefits of WireGuard.

REFERENCES

[1] (2020, May) Openvpn project. [Online]. Available: https://openvpn.net/

[2] J. A. Donenfeld, "Wireguard: Next generation kernel network tunnel." in *NDSS*, 2017.

[3] (2020, February) What is wireguard? why linux users going crazy over it? It's FOSS. [Online]. Available: https://itsfoss.com/wireguard/

[4] (2020, March) Linux's wireguard vpn is here and ready to protect you. ZDNet. [Online]. Available: https://www.zdnet.com/article/linuxs-wireguard-vpn-is-here-and-ready-to-protect-you/

[5] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the wireguard virtual private network protocol," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 231–246.

[6] A. Hülsing, K.-C. Ning, P. Schwabe, F. Weber, and P. R. Zimmermann, "Post-quantum wireguard," no. 2020/379, 2020. [Online]. Available: https://eprint.iacr.org/2020/379.pdf

[7] P. E. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," RFC 6698, Aug. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6698.txt

[8] P. Wouters, "DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP," RFC 7929, Aug. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7929.txt

[9] (2020, February) Dynamic multipoint vpn configuration guide. Cisco. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/sec_conn_dmvpn/configuration/xe-16/sec-conn-dmvpn-xe-16-book/sec-conn-dmvpn-dmvpn.html

[10] S. Gordeychik, D. Kolegov, and A. Nikolaev, "Sd-wan internet census," *arXiv preprint 1808.09027*, 2018. [Online]. Available: http://arxiv.org/abs/1808.09027

[11] B. C. Neuman and T. Ts'o, "Kerberos: An authentication service for computer networks," *IEEE Communications magazine*, vol. 32, no. 9, pp. 33–38, 1994.

[12] M. Rossberg and G. Schaefer, "A survey on automatic configuration of virtual private networks," *Computer Networks*, vol. 55, no. 8, pp. 1684 – 1699, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128611000053

[13] D. Carrel and B. Weis, "IPsec Key Exchange using a Controller," Internet Engineering Task Force, Internet-Draft draft-carrel-ipsecme-controller-ike-01, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-carrel-ipsecme-controller-ike-01

[14] A. S. Braadland, "Key management for data plane encryption in sdn using wireguard," Master's thesis, NTNU, 2017. [Online]. Available: http://hdl.handle.net/11250/2457832

[15] (2021, February) Openvpn reference manual. [Online]. Available: https://openvpn.net/community-resources/reference-manual-for-openvpn-2-4/

[16] F. Maino, L. Kreeger, and U. Elzur, "Generic Protocol Extension for VXLAN," Internet Engineering Task Force, Internet-Draft draft-ietf-nvo3-vxlan-gpe-09, Dec. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-nvo3-vxlan-gpe-09

[17] J. Gross, I. Ganga, and T. Sridhar, "Geneve: Generic Network Virtualization Encapsulation," Internet Engineering Task Force, Internet-Draft draft-ietf-nvo3-geneve-16, Mar. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-nvo3-geneve-16

[18] (2020, May) Ila kernel documents. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/ila.txt

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[20] (2020, October) grpc - a high-performance, open source universal rpc framework. Google. [Online]. Available: https://grpc.io/

[21] (2019, April) Open overlay router project. [Online]. Available: https://openoverlayrouter.org/

[22] A. Rodriguez-Natal, J. Paillisse, F. Coras, A. Lopez-Bresco, L. Jakab, M. Portoles-Comeras, P. Natarajan, V. Ermagan, D. Meyer, D. Farinacci *et al.*, "Programmable overlays via openoverlayrouter," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 32–38, 2017.

[23] lixiangyun. (2021, March) C implementation of grpc layered of top of core libgrpc. [Online]. Available: https://github.com/lixiangyun/grpc-c

[24] R. Durairajan, S. K. Mani, J. Sommers, and P. Barford, "Time's forgotten: Using ntp to understand internet latency," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015, pp. 1–7.

[25] (2020, June) Lispmob lig-lispmob. LISPmob and D. Meyer. [Online]. Available: https://github.com/LISPmob/lig-lispmob

[26] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, p. 514–534, Oct. 1968. [Online]. Available: https://doi.org/10.1145/321479.321481

[27] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," RFC 7348, Aug. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7348.txt