Incorporating Certificates with the Wireguard VPN Protocol

Aya Abdelgawad and Soham Roy

May 6, 2022

Abstract

We present an extension to the key exchange portion of the Wireguard protocol that enables the use of certificates. While the Wireguard VPN protocol has improved upon many aspects of older VPN solutions, it lacks the ability to scale efficiently for enterprise use due to a lack of support for certificates. This paper outlines the new authenticated key exchange (AKE) construction and proves its security under the extended Canetti-Krawczyk model. While our proposed construction maintains the fundamental security properties of Wireguard, some performance and security tradeoffs were made in order to add functionality.

1 Introduction

Wireguard is a relatively new VPN protocol available with kernel support on numerous platforms that aims to improve upon existing protocols by being "faster, simpler, and leaner" [Don22]. The project emphasizes maintaining a small and easy-to-audit code base, but the current feature set is insufficient for operating at scale. In particular, Wireguard currently relies on pre-shared static keys rather than certificates. Enterprises prefer to separate authorizing and authenticating users from issuing access, and certificates provide a mechanism to achieve this by using cryptographic signatures to delegate trust to a central authority. In this investigation, we propose and analyze an extension of the Wireguard VPN protocol that incorporates certificate support, concluding with a discussion on practical implementation considerations.

2 Preliminaries

We will refer to the party that sends the first message of a key exchange as the **initiator**, and the **responder** is the recipient. A **session** is a specific instance of a key exchange protocol from the perspective of a particular party, and it can be uniquely identified by the identity of the two parties involved, the party whose perspective this is from, and the messages exchanged $\big($i.e., the tuple $(id_{\mathsf{initiator}}, id_{\mathsf{responder}}, \mathsf{role} \in \{\mathsf{initiator},\ \mathsf{responder}\}, m_1, ..., m_n)\big)$. A session is **complete** once the final message of the protocol is sent. Upon completion, each party will have the tuple (k, id), where k is the session key and id is the identity of the other party they believe they are communicating with. Once a party erases k (and any other session state variables) from their memory storage, we say that the key is **expired** for that party (note that a session can be expired for one party but not necessarily the other).

We define **correctness** of an authenticated key exchange to mean that when two parties (stereotypically, we shall call them Alice and Bob) complete a session, they end up with the same shared key and are able to correctly identify the other party. Essentially, upon completing the session, Alice outputs (k_A, Bob), Bob ouputs $(k_B, Alice)$, and $k_A = k_B$.

To define security of an authenticated key exchange, we will be using the extended Canetti-Krawczyk model [LLM07]. In the original Canetti-Krawczyk [CK01] model, a PPT adversary $\mathcal{M}$ has full control over the network, with the ability to read and modify messages, delay when they reach the recipient (or block them entirely), and craft their own messages. $\mathcal{M}$ can also force an instance of a key exchange to begin between two parties. Lastly, the original model allows for certain times when adversaries can **corrupt** parties by forcing them to leak all their secrets (both long-term and session-specific). The extended model adds to the adversary's capabilities by allowing for more fine-grained revelations of secrets at *any* time. Specifically, $\mathcal{M}$ can make any of the following queries during their run:

- MasterKeyReveal(A): reveals the master private key of party A
- EphemeralKeyReveal(A, sid): reveals the ephemeral private key of party A in a (possibly incomplete) session identified by sid
- SessionKeyReveal(sid): reveals the session key for a completed session sid

We define the notion of a **clean** session as one where $\mathcal{M}$ cannot trivially compute the associated session key. Specifically, a session $sid = (A, B, \text{role}, m_1, ..., m_n)$ is clean if *neither* of the following conditions apply:

- $\mathcal{M}$ chooses or reveals both the static and ephemeral secret keys of either party.
- $\mathcal{M}$ runs SessionKeyReveal(sid).

The extended Canetti-Krawczyk model defines the AKE Game with the following rules:

1. There is a certificate authority $\mathcal{CA}$ that all parties can reliably communicate with ($\mathcal{M}$ cannot interfere with a party's communication with $\mathcal{CA}$). At the start of the experiment, all honest parties generate and register their static public keys with the $\mathcal{CA}$.
2. The adversary may then create arbitrarily many certificates for adversary-controlled parties using any public key of their choosing (even one owned by an honest party).
3. The adversary initiates network communications between parties of its choosing, making any of the queries mentioned earlier. At any point during the game, $\mathcal{M}$ can run Test(sid) on a completed session sid. This query returns with equal probability either the session key for sid or a random value. Afterwards, the adversary may continue making any of the queries mentioned in the previous step.
4. The game ends once the adversary outputs a guess of whether this is the true session key or not. $\mathcal{M}$ wins the game if they guess correctly and sid is a clean session.

We say that an AKE protocol is **secure** in the extended Canetti-Krawczyk model if the adversary's probability of winning the game is only negligibly greater than 1/2 (with respect to the security parameter).

Finally, we define notation of certificates as a tuple $(S_{id}^{pk}, id, sig_{id})$ where S_{id}^{pk} is id's public key, id is a string encoding an identity, and sig_{id} is a signature of the tuple (S_c^{pk}, id). Verifying a certificate is done by verifiyng sig_{id} using the trusted certificate authority (CA) key.

3 Current Wireguard Key Exchange Protocol

We shall give a brief overview of the current Wireguard key exchange protocol before explaining how we shall modify it. For a thorough description of the protocol, see [Don20].

3.1 Relevant Operators and Functions

- $||$: concatenation
- **DH.Keygen**() $\rightarrow sk, pk$: generates a random ECDH secret key and associated public key, specifically using Curve25519 [Ber06]
- **DH**$(sk, pk) \rightarrow x$: computes the point multiplication on Curve25519 given secret key sk and public key pk
- **KDF**$_n(k, x) \rightarrow (\tau_1, ..., \tau_n)$: key derivation function using HKDF to create an n-tuple with the provided key k and input x [Kra10]
- **HASH**$(x) \rightarrow h$: hashes x using BLAKE2s (a faster, more compact version of SHA-3) [ANWOW13]
- **AEAD**$(k, i, m, \tau) \rightarrow ct$: ChaCha20Poly1305 AEAD encryption of the message m with its associated authentication tag τ, using key k and counter i
- **TIMESTAMP**() $\rightarrow t$: returns the current time

3.2 Relevant Variables

- i, r - decorators of other variables denoting whether they belong to the initiator or responder, respectively
- S_*^{pk}, S_*^{sk} - the static public and secret key, respectively, of party * (i or r)
- E_*^{pk}, E_*^{sk} - the ephemeral public and secret key, respectively, of party *
- H, C - hash output and chaining key
- $\mathsf{Const}_1, \mathsf{Const}_2$ - string constants used during setup

3.3 Overview of Key Exchange

The initiator sends the first message by doing the following computations (there are more components and steps than what we present here, but for the sake of brevity and simplicity, we will only be going over the core components of the initiator's message)

$$
\begin{aligned}
C &\leftarrow \mathbf{HASH}(\mathsf{Const}_1) \\
H &\leftarrow \mathbf{HASH}\big(\mathbf{HASH}(C||\mathsf{Const}_2)||S_r^{pk}\big)
\end{aligned}
$$

$$
\begin{aligned}
E_i^{sk}, E_i^{pk} &\leftarrow \mathbf{DH.Keygen()} \\
C &\leftarrow \mathbf{KDF}_1(C, E_i^{pk}) \\
H &\leftarrow \mathbf{HASH}(H||E_i^{pk})
\end{aligned}
$$

$$
\begin{aligned}
(C,k) &\leftarrow \mathbf{KDF}_2\big(C, \mathbf{DH}(E_i^{sk}, S_r^{pk})\big) \\
c &\leftarrow \mathbf{AEAD}(k, 0, S_i^{pk}, H) \\
H &\leftarrow \mathbf{HASH}(H||c)
\end{aligned}
$$

$$
\begin{aligned}
(C,k) &\leftarrow \mathbf{KDF}_2\big(C, \mathbf{DH}(S_i^{sk}, S_r^{pk})\big) \\
t &\leftarrow \mathbf{AEAD}(k, 0, \mathbf{TIMESTAMP}(), H) \\
H &\leftarrow \mathbf{HASH}(H||t)
\end{aligned}
$$

The initiator then sends $m_1 = (E_i^{pk}, c, t)$. The responder will verify the message by decrypting the ciphertexts, doing the same computations, and checking that they match the provided hashes (it also needs to do these computations so its state variables end up matching the initiator's). If the responder successfully verifies the message, they construct their response by doing the following (oversimplified) computations:

$$
\begin{aligned}
E_r^{sk}, E_r^{pk} &\leftarrow \mathbf{DH.Keygen}() \\
C &\leftarrow \mathbf{KDF}_1(C, E_r^{pk}) \\
H &\leftarrow \mathbf{HASH}(H||E_r^{pk}) \\
\\
C &\leftarrow \mathbf{KDF}_1\big(C, \mathbf{DH}(E_r^{sk}, E_i^{pk})\big) \\
C &\leftarrow \mathbf{KDF}_1\big(C, \mathbf{DH}(E_r^{sk}, S_i^{pk})\big) \\
(C, \tau, k) &\leftarrow \mathbf{KDF}_3(C, 0) \\
H &\leftarrow \mathbf{HASH}(H||\tau) \\
\epsilon &\leftarrow \mathbf{AEAD}(k, 0, \emptyset, H)
\end{aligned}
$$

The responder then sends back $m_2 = (E_r^{pk}, \epsilon)$ to be verified by the initiator.

4 Proposed Construction for Certificate Support

Our proposed protocol, which we call cert-WG, extends the Wireguard protocol by one round trip to send over certificates. Recall that the first message in the Wireguard key exchange protocol consists of

$m_1 = (E_i^{pk}, c, t)$, where c and t are encryptions of the initiator's static public key and the timestamp, respectively. The key to encrypt both c and t is generated using the responder's master public key, requiring knowledge of the key before initiating the protocol. However, this is not a reasonable expectation in a certificate setting as clients may not know the keys associated with peers ahead of time. Therefore, in order to preserve the structure of the existing Wiregaurd protocol, both parties must first exchange certificates.

Our extension to Wireguard consists of inserting two messages at the beginning of the protocol. In cert–WG, the first message the initiator's certificate, C_i. The responder will only respond with their certificate C_r if the initiator's certificate is valid. The initiator will use the responder's public key $S_{c,r}^{pk}$ in C_r as the identity of the responder in the original Wireguard protocol (and vice versa for the responder using the initiator's key). After the first two messages are exchanged, the Wireguard protocol begins as normal, with the initiator sending m_1. The responder must then verify that the static public key S_i^{pk} they decrypt from m_1 equals $S_{c,i}^{pk}$ that they extracted from the C_i. Without this step, there is a potential identity misbinding attack. Finally, the responder can send the second message in the Wireguard handshake, concluding the handshake portion of cert-WG.

(See the Appendix for a visual summary of the protocol.)

5 Proof of Security

The Wireguard key exchange protocol was proven to be secure in the extended Canetti-Krawkczyk model [DP18][1]. We claim that if the Wireguard key exchange is secure, then cert-WG is, too. Suppose there exists an efficient adversary μ who has a non-negligible advantage ϵ in the AKE Game for cert-WG. We can then construct an adversary $\mathcal{M}$ for Wireguard's key exchange that has an advantage of ϵ in the AKE Game by using μ and simulating cert-WG to it. $\mathcal{M}$ will do the following:

1. $\mathcal{M}$ simulates being all honest parties to μ. $\mathcal{M}$ also simulates the $\mathcal{CA}$ and distributes certificates to those parties (as well as any fictitious parties μ wishes to register with the $\mathcal{CA}$).

2. $\mathcal{M}$ runs μ and allows them to influence the simulated network, copying μ's actions in the real network and relaying the responses back into the simulated network. For example, if μ tries to start a cert-WG key exchange with an honest party in the simulated network, $\mathcal{M}$ will start a Wireguard key exchange with that party in the real network (the only difference being that $\mathcal{M}$ must first simulate sending over that party's certificate and verifying the certificate received).

3. Once the simulated protocol reaches the Wireguard portion of the protocol, $\mathcal{M}$ is relying on the real parties to produce valid messages in the protocol. $\mathcal{M}$ stops any message m in the real network before it reaches its intended destination and mimics having it sent in the simulated network. If μ makes any modifications to m to produce m', $\mathcal{M}$ will also modify m in the real network so that m' is sent instead.

4. $\mathcal{M}$ continues simulating the cert-WG protocol until μ outputs a guess b', which $\mathcal{M}$ will echo as their guess.

[1]The paper actually proved a minimally modified version of the Wireguard protocol secure due to the protocol's implicit reliance on the first message in the transport data for authenticating the initiator. To get around this, they had the initiator send an additional message at the end to confirm they have the same key value. In order for the proof to unfold nicely, we implicitly apply the same modification to cert-WG by also adding the key-confirmation message at the end and reduce this modified version of cert-WG to the modified Wireguard protocol.

Note that $\mathcal{M}$ is able to simulate μ's expected environment almost perfectly, with the only exception possibly happening after m_1 is sent. At this stage in the cert-WG protocol, the responder must check that the public key in the initiator's certificate matches the key after decrypting c. Since $\mathcal{M}$ may not be able to decrypt c themselves to check this, they will rely on a policy of aborting the protocol if μ attempts to tamper with c, assuming that this check will fail. Since c is an AEAD ciphertext, we get ciphertext integrity guarantees that μ would only have a negligible chance of tampering the ciphertext undetected anyways. Thus, $\mathcal{M}$ perfectly simulates the cert-WG protocol except with negligible probability, so $\mathcal{M}$'s advantage in the game is only negligibly smaller than ϵ.

6 Conclusion

In this paper, we have shown that the Wireguard key exchange is quite easily extended to support certificates while still maintaining security in the extended Canetti-Krawczyk model. There are a few costs that come with this model, though. First, it requires an entire additional round trip (although the protocol can be modified to 1.5 RTT by having the responder send both their certificate and m_1 in one message). Second, the certificates can be verified but not be authenticated (i.e., proving knowledge of the associated private key) until getting to the Wireguard part of the protocol. This clashes with Wireguard's design principle "to not send any responses to unauthenticated packets" [Don20]. Lastly, certificates in practice are much more complicated than we presented in our paper. Practical considerations usually necessitate additional sophistication in the construction of certificates, including key usage, certificate constraints, validity periods, transparency logs, revocation support, etc (all elements we considered out of scope of this paper). Incorporating all the code needed for proper certificate verification would vastly expand Wireguard's relatively small codebase, a trait it prides itself in.

All these factors push for a more modular implementation of cert-WG. Rather than directly extending the Wireguard protocol to support certificates, a more practical approach is to add a layer of certificate support above the Wireguard key exchange. Due to the robust design of the Wireguard protocol, extensions that add functionality can be added with relative simplicity. It is possible to tailor these extensions to control the trade-off between functionality and security, while maintaining much of the underlying performance and security properties of the original protocol. An extension such as cert-WG can be implemented to augment the key exchange with zero overhead to data transit.

References

[ANWOW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: Simpler, smaller, fast as md5. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, pages 119–135, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Ber06] 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*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 453–474, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Don20] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[Don22] Jason A. Donenfeld. Wireguard, 2022.

[DP18] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the wireguard protocol. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 3–21, Cham, 2018. Springer International Publishing.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 631–648, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LLM07] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security*, pages 1–16, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

A Diagram of cert-WG