A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol

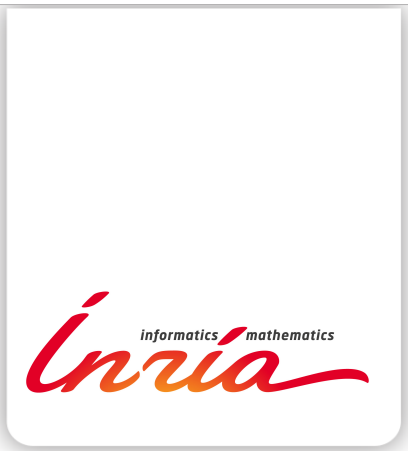
Benjamin Lipp, Bruno Blanchet, Karthikeyan Bhargavan

A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol

Benjamin Lipp, Bruno Blanchet, Karthikeyan Bhargavan

RESEARCH REPORT
N° 9269
April 2019
Project-Team Prosecco

ISSN 0249-6399 ISRN INRIA/RR--9269--FR+ENG



A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol

Benjamin Lipp, Bruno Blanchet, Karthikeyan Bhargavan

Project-Team Prosecco

Research Report n° 9269 — April 2019 — 45 pages

Abstract: WireGuard is a free and open source Virtual Private Network (VPN) that aims to replace IPsec and OpenVPN. It is based on a new cryptographic protocol derived from the Noise Protocol Framework. This paper presents the first mechanised cryptographic proof of the protocol underlying WireGuard, using the CryptoVerif proof assistant.
We analyse the entire WireGuard protocol as it is, including transport data messages, in an ACCE-style model. We contribute proofs for correctness, message secrecy, forward secrecy, mutual authentication, session uniqueness, and resistance against key compromise impersonation, identity mis-binding, and replay attacks. We also discuss the strength of the identity hiding provided by WireGuard.
Our work also provides novel theoretical contributions that are reusable beyond WireGuard. First, we extend CryptoVerif to account for the absence of public key validation in popular Diffie-Hellman groups like Curve25519, which is used in many modern protocols including WireGuard. To our knowledge, this is the first mechanised cryptographic proof for any protocol employing such a precise model. Second, we prove several indifferentiability lemmas that are useful to simplify the proofs for sequences of key derivations.

Key-words: security protocols, verification, computational model, VPN

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Une preuve cryptographique mécanisée du protocole de réseau privé virtuel WireGuard

Résumé : WireGuard est un logiciel de réseau privé virtuel (VPN) gratuit et *open source* qui cherche à remplacer IPsec et OpenVPN. Il est fondé sur un nouveau protocole cryptographique dérivé de la famille de protocoles Noise. Ce document présente la première preuve cryptographique mécanisée du protocole de WireGuard, obtenue avec l'assistant de preuve CryptoVerif.

Nous analysons le protocole WireGuard en entier, tel qu'il est, y compris les messages de transport de données, dans un modèle du style ACCE. Nous obtenons des preuves de correction, secret des messages, *forward secrecy*, authentification mutuelle, unicité des sessions, et résistance contre des attaques d'imposture par compromis de clés, de mauvaise liaison d'identités, et de rejeu. Nous discutons également la robustesse de la protection des identités fournie par WireGuard.

Notre travail fournit aussi de nouvelles contributions théoriques, réutilisables au-delà de WireGuard. Premièrement, nous étendons CryptoVerif pour tenir compte de l'absence de validation des clés publiques dans des groupes Diffie-Hellman populaires comme Curve25519, qui est utilisé dans beaucoup de protocoles modernes dont WireGuard. À notre connaissance, c'est la première preuve cryptographique mécanisée qui utilise un modèle aussi précis. Deuxièmement, nous prouvons plusieurs lemmes d'indifférentiabilité qui sont utiles pour simplifier les preuves de suites de dérivations de clés.

Mots-clés : protocoles cryptographiques, vérification, modèle calculatoire, VPN

1 Introduction

The traditional distinction between a secure intranet and the untrusted Internet is becoming less relevant as more and more enterprises host internal services on cloud-based servers distributed across multiple data centres. Sensitive data that used to travel only between physically proximate machines within secure buildings is now sent across an unknown number of network links that may be controlled by malicious entities.

To maintain the security of such *distributed intranets*, the most powerful tools at the disposal of system administrators are Virtual Private Network (VPN) protocols that set up low-level secure channels between machines, and hence can be used to transparently protect all the data exchanged between them. Indeed, all leading cloud providers now offer VPN gateways, so that enterprises can treat cloud-based servers as if they were located within their intranet.[1]

Standards vs. Custom Protocols. Most popular VPN solutions are based on Internet standards like IPsec [35] and TLS [21], for several reasons. First, these protocols typically have multiple interoperable implementations that are available on all mainstream operating systems, so the VPN software can be easily built as a layer on top. Second, standards are designed to be future-proof by relying on versioning and *cryptographic agility*, so that a VPN protocol can easily move from one protocol version or cryptographic algorithm to another if (say) a weakness were found on some configuration. Third, published standards typically have been closely scrutinised by numerous interested parties, and hence are believed to be less likely to contain obvious security flaws.

Conversely, using a standard protocol also has its disadvantages. Standardisation takes time, and so a standard protocol may not use the most modern cryptographic algorithms. On the contrary, the need for interoperability and backwards compatibility often force implementations to continue support for obsolete cryptographic algorithms, leading to cryptanalytic attacks [9] and software flaws [6]. Over time, standards and their implementations can grow to an unmanageable size that can no longer be studied as a whole, allowing logical flaws to hide in unused corners of the protocol [8].

Consequently, many new secure channel protocols eschew standardisation in favour of a lean design that uses only modern cryptography and supports minimal cryptographic agility. The succinctness of the protocol description aids auditability, and the lack of optional features reduces complexity. Examples of this approach are the Signal protocol [29] used in many secure messaging systems and the Noise protocol framework [33].

WireGuard is a VPN protocol that adopts this design philosophy [18]. It implements and extends a secure channel protocol derived from the Noise framework, and it chooses a small set of modern cryptographic primitives. By making these choices, WireGuard is able to provide a high-quality VPN in a few thousand lines of code, and is currently being considered for adoption within the Linux kernel. The design of WireGuard is detailed and informally analysed in [18], but a protocol of such importance deserves a thorough security analysis.

A Need for Mechanised Proofs. Having a succinct, well-documented description is a good basis for understanding, auditing, and implementing a custom cryptographic protocol, but in itself is no guarantee that the protocol is secure. Symbolic analysis with tools like ProVerif [13] and Tamarin [30] can help find logical flaws, and WireGuard already has been analysed using Tamarin [19]. However, symbolic analyses do not constitute a full cryptographic proof. For example, they cannot demonstrate the absence of cryptanalytic attacks on secure channels and

[1] `https://cloud.google.com/vpn/docs/concepts/overview`, `https://docs.aws.amazon.com/vpc/latest/userguide/vpn-connections.html`, `https://azure.microsoft.com/en-us/services/vpn-gateway/`

VPNs (e.g. [9].)

Cryptographic proofs provide the highest form of formal assurance, but writing proofs by hand requires significant expertise and effort, especially if the proof is to account for the precise low-level details of a real-world protocol. And as proofs get larger, the risk of introducing proof errors becomes non-negligible. All this effort is hard to justify for a custom protocol which may change as the software evolves. For example, a manual cryptographic proof for the WireGuard protocol appears in [20], but this proof would need to be carefully reviewed and adapted if the WireGuard protocol were to change in any way or if a variant of WireGuard were to be proposed.

We advocate the use of mechanised provers to build cryptographic proofs, so that they can be checked for errors, and can be easily modified to accommodate different variants of the protocol. In this paper, we rely on the CryptoVerif protocol verifier [10, 11] to build a proof of WireGuard. CryptoVerif relies on a computational model of cryptography, and generates machine-checkable proofs by sequences of games, like those manually written by cryptographers.

Uncovering Real-World Cryptographic Assumptions. A mechanised proof also allows the analyst to experiment with a variety of cryptographic assumptions and discover the precise set of assumptions that a protocol's security depends on.

In some cases, a protocol may require an unusual assumption about a hash function, or a stronger assumption about encryption than one may have expected, and these cases can provide a guide to implementers on what concrete cryptographic algorithms should or should not be used to instantiate the protocol. For example, in our analysis of WireGuard, we find that most of the standard properties require only standard assumptions about the underlying authenticated encryption scheme (AEAD) but identity hiding requires a stronger assumption, which is satisfied by the specific algorithms used by WireGuard, but may not be provided by other AEAD constructions.

In other cases, a protocol's use of a cryptographic primitive may motivate a new, more precise model of the primitive. Protocols like WireGuard seek to depend on a small set of primitives and reuse them in different ways. For example, WireGuard relies on the Curve25519 elliptic curve Diffie-Hellman operation for an ephemeral key exchange as well as for entity authentication. It uses Curve25519 public keys both as identities and as unique nonces to identify sessions. To verify that Curve25519 is appropriate for all these usages, and to prove the absence of attacks such as replays, identity mis-binding, and key compromise impersonation, we need to account for the details of the Curve25519 group, rather than rely on a generic Diffie-Hellman assumption. Hence, we propose a new model for Curve25519 in CryptoVerif and prove WireGuard secure against this model.

Contributions. We present the first mechanised proof for the cryptographic design of the WireGuard VPN, including the Noise IKpsk2 secure channel protocol it uses. Our analysis is done on WireGuard v1 as specified in [18]. In addition to classic key exchange security for IKpsk2, we examine the identity hiding and denial-of-service protections provided by WireGuard. We conclude with a discussion of the strengths and weaknesses of WireGuard, and propose improvements that would allow for stronger security theorems.

Our work also provides contributions reusable beyond the proof of WireGuard. To the best of our knowledge, this is the first mechanised proof for any cryptographic protocol that takes into account the precise structure of the Curve25519 group. We also prove a series of indifferentiability results that allow us to simplify sequences of random oracle calls, and we made several extensions to CryptoVerif that we mention in the rest of the paper when we use them. These extensions are included in CryptoVerif 2.01 available at `http://cryptoverif.inria.fr/`.

Our models of WireGuard are available at `http://cryptoverif.inria.fr/WireGuard`.

2 WireGuard

WireGuard [18] establishes a VPN tunnel between two remote hosts in order to securely encapsulate all Internet Protocol (IP) traffic between them. The main design goals of WireGuard are to be simple, fast, modern, and secure. In order to establish a tunnel, a system administrator only needs to configure the IP address and long-term public key for the remote host. With this information, WireGuard can establish a secure channel, using a protocol derived from the Noise framework, instantiated with fast, modern cryptographic primitives like Curve25519 and BLAKE2. The full WireGuard VPN is implemented in a few thousand lines of code that can run on multiple platforms, but for performance, is usually run within the operating system kernel. In particular, WireGuard is in the process of being incorporated into the Linux kernel (most likely Linux 4.2/5.0), as an alternative to IPsec.

In this section, we focus on the cryptographic design of WireGuard. We begin by describing the secure channel component, then the extensions WireGuard makes for denial-of-service and stealthy operation. We end the section by detailing the concrete cryptographic algorithms used by WireGuard and the list of informal security goals it seeks to achieve.

2.1 Secure Channel Protocol: Noise IKpsk2

Noise [33] is a framework for building two-party cryptographic protocols that are secure by construction. Using the building blocks in this framework, a designer can create a new protocol that matches a desired subset of security guarantees: mutual or optional authentication, identity hiding, forward secrecy, etc. The Noise specification also includes a list of curated pre-defined protocols, with an informal analysis of their message-by-message security claims. WireGuard instantiates one of these protocols, which is called IKpsk2, and extends it to provide further guarantees needed by VPNs.

The secure channel protocol is depicted in Figure 1a, and the cryptographic computations are detailed in Figure 1b, using notations similar to [18]. Before the protocol begins, the initiator i and the responder r are assumed to have exchanged their long-term *static public keys* (S_i^{pub}, S_r^{pub}). Optionally, they may have also established a *pre-shared symmetric key* (*psk*); if this key is absent it is set to a key-sized bitstring of zeros.

Message Exchange. The protocol begins when i sends the first handshake message to r, which includes the following components:

- I_i: a fresh session identifier, generated by i,
- E_i^{pub}: a fresh ephemeral public key, generated by i,
- $S_{i\,\blacksquare}^{pub}$: i's static public key, encrypted for r,
- $ts_{\blacksquare}$: a timestamp, encrypted with a key that can be computed only by i and r, and
- mac_1, mac_2: message authentication codes (see §2.2).

In response, r sends the second handshake message containing:

- I_i: i's session identifier,
- I_r: a fresh session identifier, generated by r,
- E_r^{pub}: a fresh ephemeral public key, generated by r,

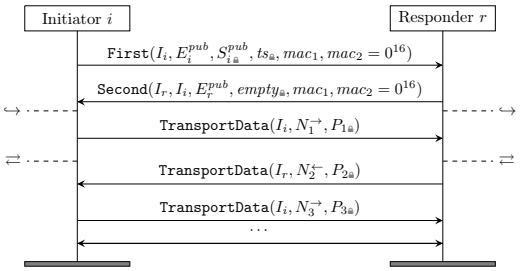
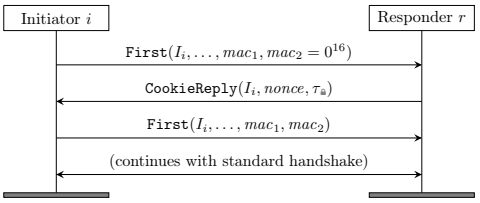


Figure 1a: WireGuard's protocol messages.



- - - - - - - - - CookieReply - - - - - - - - -

$R_r \leftarrow\!\$\ \{0,1\}^{256}$ (refresh only every 2 minutes)

$\tau \leftarrow \mathsf{mac}(R_r, A_i)$ with $A_i = \mathsf{IP}_i \| \mathsf{Port}_i$

$nonce \leftarrow\!\$\ \{0,1\}^{192}$

$\tau_{🔒} \leftarrow \mathsf{xaenc}(\mathsf{hash}(\mathsf{label}_{\mathsf{cookie}} \| S_r^{pub}), nonce, \tau, mac_1)$

- - - - - - - - - Non-zero mac_2 - - - - - - - -

$mac_2 \leftarrow \mathsf{mac}(\tau, msg_\beta)$

Figure 1c: Cookie mechanism under load.

- - - - - - - - - First - - - - - - - - - - - -

$E_i^{pub}: (E_i^{priv}, E_i^{pub}) \leftarrow\!\$\ \mathsf{keygen}()$

$C_0 \leftarrow \mathsf{hash}(\mathsf{protocol\_name})$

$H_0 \leftarrow \mathsf{hash}(C_0 \| \mathsf{prologue})$

$H_1 \leftarrow \mathsf{hash}(H_0 \| S_r^{pub})$

$C_1 \leftarrow \mathsf{hkdf}_1(C_0, E_i^{pub})$

$H_2 \leftarrow \mathsf{hash}(H_1 \| E_i^{pub})$

$C_2 \| k_1 \leftarrow \mathsf{hkdf}_2(C_1, \mathsf{dh}(E_i^{priv}, S_r^{pub}))$

$S_{i🔒}^{pub}$: $S_{i🔒}^{pub} \leftarrow \mathsf{aenc}(k_1, 0, S_i^{pub}, H_2)$

$H_3 \leftarrow \mathsf{hash}(H_2 \| S_{i🔒}^{pub})$

$C_3 \| k_2 \leftarrow \mathsf{hkdf}_2(C_2, \mathsf{dh}(S_i^{priv}, S_r^{pub}))$

$ts_{🔒}$: $ts_{🔒} \leftarrow \mathsf{aenc}(k_2, 0, \mathsf{timestamp}(), H_3)$

$H_4 \leftarrow \mathsf{hash}(H_3 \| ts_{🔒})$

mac_1: $mac_1 \leftarrow \mathsf{mac}(\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_r^{pub}), msg_\alpha)$

- - - - - - - - - Second - - - - - - - - - - -

$E_r^{pub}: (E_r^{priv}, E_r^{pub}) \leftarrow\!\$\ \mathsf{keygen}()$

$C_4 \leftarrow \mathsf{hkdf}_1(C_3, E_r^{pub})$

$H_5 \leftarrow \mathsf{hash}(H_4 \| E_r^{pub})$

$C_5 \leftarrow \mathsf{hkdf}_1(C_4, \mathsf{dh}(E_r^{priv}, E_i^{pub}))$

$C_6 \leftarrow \mathsf{hkdf}_1(C_5, \mathsf{dh}(E_r^{priv}, S_i^{pub}))$

$C_7 \| \pi \| k_3 \leftarrow \mathsf{hkdf}_3(C_6, psk)$

$H_6 \leftarrow \mathsf{hash}(H_5 \| \pi)$

$empty_{🔒}$: $empty_{🔒} \leftarrow \mathsf{aenc}(k_3, 0, empty, H_6)$

$H_7 \leftarrow \mathsf{hash}(H_6 \| empty_{🔒})$

mac_1: $mac_1 \leftarrow \mathsf{mac}(\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_i^{pub}), msg_\alpha)$

- - - - - - - - - Key Derivation - - - - - - - -

$T^{\rightarrow} \| T^{\leftarrow} \leftarrow \mathsf{hkdf}_2(C_7, empty)$

- - - - - - - - - TransportData - - - - - - - - -

$P_{1🔒} \leftarrow \mathsf{aenc}(T^{\rightarrow}, N_1^{\rightarrow} = 0, P_1, empty)$

$P_{2🔒} \leftarrow \mathsf{aenc}(T^{\leftarrow}, N_2^{\leftarrow} = 0, P_2, empty)$

$P_{3🔒} \leftarrow \mathsf{aenc}(T^{\rightarrow}, N_3^{\rightarrow} = 1, P_3, empty)$

Figure 1b: Cryptographic Computations for Protocol Messages.

Figure 1: (a) An overview of WireGuard's main protocol messages; (b) the cryptographic computations used to create these messages; they need to be adapted accordingly for the receiving side; and (c) the cookie mechanism used by WireGuard to protect hosts against Denial-of-Service attacks. We write $x_{🔒}$ for a variable containing an encryption of x; $x_{🔒}$ is just a variable identifier. msg_α refers to all the bytes of a message up to but not including mac_1, msg_β is the same but including mac_1. Session key derivation takes places after the second protocol message, symbolised by $\hookrightarrow$, at which point the initiator can send messages. The end of the handshake is symbolised by $\rightleftarrows$, after which transport data messages can be sent in both directions. The cookie mechanism is depicted in one direction, initiator to responder, but can actually be used by either initiator or responder, whichever is under load.

- $empty_{\text{🔒}}$: an empty bytestring encrypted with a key that can be computed only by i and r, and
- mac_1, mac_2: message authentication codes (see §2.2).

The encrypted payloads in the two messages serve as authenticators: by computing the corresponding encryption key, each party proves that it knows the private key for its static public key. The encryption key for the second message also requires knowledge of the optional *psk* providing an additional authentication guarantee. The two ephemeral keys add fresh session-specific key material that can be used to compute (forward) secret session keys known only to i and r.

At the end of these two messages, i and r derive authenticated encryption keys $(T^{\rightarrow}, T^{\leftarrow})$ that can be used to transport IP traffic in the two directions. Importantly, i sends the first transport message, hence confirming the successful completion of the handshake to r, before r sends it any encrypted traffic. Each of these transport messages includes:

- I_i or I_r: the recipient's session identifier,
- $N_j^{\leftarrow}$ or $N_j^{\rightarrow}$: the current message counter,
- P_j: an IP datagram, encrypted under the traffic key.

Cryptographic Computations. Figure 1b describes how each of these message components and traffic keys are computed. As the handshake proceeds, i and r compute a sequence of *transcript hashes* $(H_0, H_1, \ldots, H_7)$ that hashes in all the public data used in the two handshake messages, including:

- $\mathsf{protocol\_name}, \mathsf{prologue}$: strings identifying the protocol,
- E_i^{pub}, E_r^{pub}: both ephemeral public keys,
- $S_r^{pub}, S_{i\text{🔒}}^{pub}$: both static public keys, but with the initiator's key in encrypted form,
- $ts_{\text{🔒}}$, $empty_{\text{🔒}}$: both encrypted handshake payloads, and
- π: an identifier derived from the pre-shared key.

These transcript hashes serve as unique identifiers for the current stage of the session. In particular, no two completed WireGuard sessions should have the same H_7.

Both parties also derive a sequence of *chaining keys* $(C_0, C_1, \ldots, C_7)$ by mixing in all the key material, including:

- $\mathsf{protocol\_name}$, E_i^{pub}, E_r^{pub},
- $\mathsf{dh}(E_i^{priv}, S_r^{pub}) = \mathsf{dh}(S_r^{priv}, E_i^{pub})$: the *ephemeral-static* Diffie-Hellman shared secret computed using the initiator's ephemeral key (named first in *ephemeral-static*) and the responder's static key (named second in *ephemeral-static*),
- $\mathsf{dh}(S_i^{priv}, S_r^{pub}) = \mathsf{dh}(S_r^{priv}, S_i^{pub})$: the static-static shared secret,
- $\mathsf{dh}(E_i^{priv}, E_r^{pub}) = \mathsf{dh}(E_r^{priv}, E_i^{pub})$: the ephemeral-ephemeral shared secret,
- $\mathsf{dh}(S_i^{priv}, E_r^{pub}) = \mathsf{dh}(E_r^{priv}, S_i^{pub})$: the static-ephemeral shared secret, and
- psk: the (optional) pre-shared key.

The function dh is the elliptic curve scalar multiplication, taking a private key and a public key as argument, permitting the computation of a *shared secret* [27]. In the preceding list, the initiator uses the first function call, and the responder the second one, respectively.

The protocol uses all four combinations of static and ephemeral Diffie-Hellman shared-secret computations to maximally protect against the compromise of some of these keys. The *psk* also serves as a defensive countermeasure against quantum adversaries who may be able to break the Diffie-Hellman construction, but not hkdf. Hence, by using a frequently updated *psk*, WireGuard users can protect current sessions against future quantum adversaries.

Each chaining key is mixed into the next chaining key via an hkdf key derivation that also outputs encryption keys as needed. This chain of key derivations outputs two encryption keys (k_1, k_2) for the first handshake message, an encryption key (k_3) and a PSK identifier (π) for the second message, and traffic keys $(T^{\leftarrow}, T^{\rightarrow})$ for all subsequent transport messages.

To encrypt each message, WireGuard uses an authenticated encryption scheme with associated data (AEAD) that takes a key, a counter, a plaintext (padded up to the nearest blocksize) and an optional hash value as associated data. The encryptions in the handshake messages use the current transcript hash (H_2, H_3, H_6) as associated data, which guarantees that the two participants have a consistent session transcript. Transport messages use an empty string as associated data. The message counter is initially set to 0 for each AEAD key and incremented by 1 every time the key is reused.

Relationship with IKpsk2. The secure channel protocol described above is a direct instantiation of Noise IKpsk2, with five notable differences. First, WireGuard adds local session identifiers (I_i, I_r) for the initiator and responder. Second, WireGuard fixes the payload of the first message to a timestamp, and the one of the second message to the empty string. Third, WireGuard stipulates that the first traffic message is sent from the initiator to the responder. Fourth, WireGuard excludes zero Diffie-Hellman shared secrets to avoid points of small order, while Noise recommends not to perform this check. Fifth, WireGuard adds two message authentication codes to the handshake messages, to provide stealth and to protect against DoS, as described in the next section. We also observe that although this protocol is superficially similar to other popular Noise protocols like IK (which is used in WhatsApp), there are important differences between these variants and a proof for one does not translate to the other.

2.2 Extensions for Stealth and Denial-of-Service

A VPN protocol operates at a low-level in the networking stack and hence needs to not only protect against cryptographic attacks, but also real-world network-level attacks such as *denial of service* (DoS). Indeed, a cryptographic protocol like IKpsk2 that needs to perform two expensive Diffie-Hellman operations before it can authenticate a handshake message is even more vulnerable to DoS: an adversary can send bogus messages that tie up computing resources on the recipient. A further security goal for WireGuard is that its VPN endpoints should be *stealthy*, in the sense that it should not be possible for a network adversary to blindly scan for WireGuard services.

To support stealthy operation, WireGuard endpoints do not respond to any handshake message unless the sender can prove that it knows the static public key of the recipient. This proof is incorporated in the mac_1 field included in each handshake message, which contains a message authentication code (MAC) computed over the prefix of the current handshake message up to but not including mac_1, using a MAC key derived from the recipient's static public key. The recipient verifies this MAC before processing the message, and stays silent if the MAC fails. Hence, a network adversary who does not know the public key cannot detect whether WireGuard is running on a machine, and at the same time cannot force the recipient to perform two finally useless Diffie-Hellman operations.

To protect more actively against DoS, WireGuard incorporates a cookie-based protocol (depicted in Figure 1c) that a host can use when it is under load. For example, if the responder suspects it is under a DoS attack, it can refuse to process the first handshake message and instead send back an initiator-specific fresh *cookie* (τ) that is computed from a frequently rotated secret key (R_r) (known only to the responder) and the initiator's IP address (IP_i) and source port (Port_i). The responder encrypts this cookie for the initiator, using a key derived from the initiator's static public key, a fresh nonce, and the mac_1 field of the first message as associated data.

The initiator decrypts τ and then retries the handshake by sending the first message again, but this time with a second field mac_2 that contains a MAC over the full message up to and including mac_1, using τ as the MAC key. After verifying this MAC, the responder continues with the standard handshake.

However, to obtain τ, an adversary must be able to read messages on the network path between the initiator and responder and must also know the initiator's static key (which is never sent in the clear by the protocol). And even if the adversary has both these capabilities, it is required to perform session specific cryptographic computations for every handshake message it sends to the responder, significantly limiting its ability to mount a DoS attack. Hence, this cookie protocol protects the recipient from brute-force network attacks.

Note that the mac_2 field is included in both handshake messages, and hence can be used in both directions, to protect both the initiator and responder from DoS attacks.

The two MACs are WireGuard-specific mechanisms which are not present in IKpsk2. Since they do not use any of the session keys (or hashes or chaining keys) that are used in IKpsk2, adding these mechanisms should, in principle, not affect the security of the secure channel protocol. However, since the static public keys of the two hosts are used in the two MACs, we need to carefully study their impact on the identity-hiding guarantees of IKpsk2.

2.3 Instantiating the Cryptographic Algorithms

WireGuard uses a small set of cryptographic constructions and instantiates them with modern algorithms, carefully chosen to provide strong security as well as high performance:

- dh: all Diffie-Hellman operations use the Curve25519 elliptic curve [27], that uses private and public keys that are 32 bytes long;
- hash: all hash operations use the BLAKE2s hash function [36], which returns a 32-byte hash;
- aenc: authenticated encryption for handshake and traffic message uses the AEAD scheme ChaCha20Poly1305 [31], where the key has 32 bytes, the 96-bit nonce is composed of 32 bits of zeroes followed by the 64-bit little-endian value of the message *counter*, the plaintext is padded with zeroes up to the nearest 16-byte block, and the associated data is a hash value of 32-bytes;
- xaenc: cookie encryption uses an *extended* AEAD construction using XChaCha20Poly1305, which incorporates a 192-bit random nonce [5] into the standard ChaCha20Poly1305 construction;
- mac: all MAC operations use the keyed MAC variant of the BLAKE2s hash function, which returns a 16-byte tag;
- hkdf_n: all key derivations use the HKDF construction [26], using BLAKE2s as the underlying hash function.

WireGuard also uses some constants to indicate the specific algorithms it uses and to disambiguate different uses of the mac and xaenc primitives. The protocol_name field is set to the UTF-8 string ‘‘Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s’’ while the prologue is set to ‘‘WireGuard v1 zx2c4 Jason@zx2c4.com’’. The mac_1 computation uses the UTF-8 string ‘‘mac1--’’ as $\mathsf{label}_{\mathsf{mac1}}$, and the cookie computation uses ‘‘cookie-’’ as $\mathsf{label}_{\mathsf{cookie}}$.

2.4 Security Goals, Informally

Using the mechanisms described in this section, WireGuard seeks to provide the following set of strong security guarantees, inheriting the security claims of Noise IKpsk2 [33] and extending them with the additional DoS and stealth goals of WireGuard [18]. In the following, we use *honest* to refer to a party that follows the protocol specification, and *dishonest* to a party that doesn't, i.e. that is controlled by the adversary. Most properties are defined to hold within a *clean* session; we define this notion formally in Section §5.1.

- **Correctness:** If an honest initiator and an honest responder complete a WireGuard handshake and the messages are not altered by an adversary, then the transport data keys $(T^{\rightarrow}, T^{\leftarrow})$ and the transcript hash H_7 are the same on both hosts.
- **Secrecy:** If a transport data message P is sent over a tunnel between two honest hosts, then this message is kept confidential from the adversary. Furthermore, the traffic keys for this tunnel are also confidential.
- **Forward Secrecy:** Secrecy for a session holds even if both the static private keys (S_i^{priv}, S_r^{priv}) and the pre-shared key (psk) become known to the adversary, but only after the session has been completed and all its traffic keys and chaining keys are deleted by both parties.

  Secrecy also holds even if the static and ephemeral keys are compromised (e.g. by a quantum adversary), as long as the pre-shared key is not compromised.
- **Mutual Authentication:** If an honest initiator (resp. responder) completed a handshake (ostensibly) with an honest peer, then that peer must have participated in this handshake. Moreover, if a host A receives a plaintext message over a WireGuard tunnel that claims to be from host B, then B must have (intentionally) sent this message to A.
- **Resistance against Key Compromise Impersonation (KCI):** The recipient of a message can authenticate the message's sender even if the recipient's static key is compromised.
- **Resistance against Identity Mis-Binding:** If two honest parties derive the same traffic keys in some WireGuard session, then they agree on each other's identities, even if one or both of them have been interacting with a dishonest party or a honest party with compromised keys. This property is also called resistance against unknown key-share attacks.
- **Resistance against Replay:** Any protocol message sent may be accepted at most once by the recipient.
- **Session Uniqueness:** There is at most one honest initiator session and at most one honest responder session for a given traffic key. Similarly, there is at most one honest initiator session and at most one honest responder session for given handshake messages.
- **Channel Binding:** Two sessions that have the same final session transcript hash H_7 share the same view and the same session keys.

- **Identity Hiding:** Just by looking at the messages transmitted over the network, a passive adversary cannot infer the static keys involved in a session. (However, these identities are not forward secret: If the responder's static key gets compromised, the adversary can later decrypt the initiator's static public key that was transmitted in the first message.)
- **DoS Resistance:** The adversary cannot have a message accepted by a recipient under load without having first made a round trip with that recipient. In practice, this means that the adversary has to be at the claimed address. Because we assume that the adversary controls the network, we cannot prove more than enforcing a round trip.

The security goals above are stated in terms of completed WireGuard sessions, with most security guarantees only applying after the third message, when both initiator and responder start freely sending and receiving data. In particular, the first transport data message (i.e. the third message) serves as key confirmation to the responder, and is needed to prove that the initiator has control over its ephemeral key. This is why, in WireGuard, the responder does not send any data until it sees this third message. In the rest of this paper, we investigate whether WireGuard achieves the goals set out above.

3 Cryptographic Assumptions

This section presents the assumptions that we make on the cryptographic primitives used by WireGuard. For most primitives, the desired assumption is already present in the library of primitives of CryptoVerif, so we just need to call a macro to use that assumption. Still, we had to design a new model for Curve25519, detailed below.

3.1 Random Oracle Model

We assume that BLAKE2s is a random oracle [3]. This assumption is justified in [28] using a weak ideal block cipher. In particular, BLAKE2s uses a prefix-free Merkle-Damgård construction, thanks to the use of finalisation flags. Therefore, extension attacks which apply to pure Merkle-Damgård constructions do not apply to BLAKE2s.

3.2 IND-CPA and INT-CTXT for AEAD

We assume that the ChaCha20Poly1305 AEAD scheme [31] is IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) [2], provided the same nonce is never used twice with the same key. IND-CPA means that the adversary has a negligible probability of distinguishing encryptions of two distinct messages of the same length that it has chosen. INT-CTXT means that an adversary with access to encryption and decryption oracles has a negligible probability of forging a ciphertext that decrypts successfully and has not been returned by the encryption oracle. These properties are justified in [34], assuming ChaCha20 is a PRF (pseudo-random function) and Poly1305 is an ϵ-almost-Δ-universal hash function. The latter property is shown to hold in [4].

3.3 Curve25519 and Gap Diffie-Hellman

WireGuard uses the elliptic curve Curve25519 [27] for Diffie-Hellman key exchanges. This curve is a group G of order kq where $k = 8$ (cofactor) and q is a large prime. The base point g has prime order q; we denote by G_{sub} the prime order subgroup generated by g. In WireGuard and typical implementations of Curve25519 as specified by RFC 7748 [27], the incoming public keys are not

verified, so they may be any element of G and may not belong to G_{sub}, and all exponents are non-zero multiples of k modulo kq. For each public key X in G, there are k public keys Y in G such that $X^k = Y^k$ and only one of these public keys is in G_{sub}. (We write point multiplication exponentially.) We say that public keys X and Y such that $X^k = Y^k$ are *equivalent*, because they yield the same Diffie-Hellman shared secrets: for any exponent $z = kz'$, $X^z = X^{kz'} = Y^{kz'} = Y^z$. Moreover, the public keys may be 0, the neutral element of G and G_{sub}, and $0^x = 0$ for all x.

While most proofs of Diffie-Hellman key agreements assume a prime order group, that assumption is not correct for most implementations of Curve25519. For instance, the identity mis-binding issue that we discuss in Section 6 would not appear in a prime order group. Therefore, we need to provide a new model that takes into account the properties mentioned above.

The main idea of our model is to rely on a Diffie-Hellman assumption in the prime order subgroup G_{sub}, and so to work as much as possible with elements in G_{sub}. We rewrite the computations in G into computations in G_{sub} by first raising the public keys to the power k, and we rely on standard properties of prime order groups for G_{sub}.

In CryptoVerif, we first define the following types:

type G [bounded, large].
type G_{sub} [bounded, large].
type Z [bounded, large, nonuniform].

The type G represents the group G; it is bounded because it is represented by bitstrings of bounded length, and large because collisions between randomly chosen elements in G have a negligible probability. Similarly, the type G_{sub} represents the group G_{sub}, and the type Z corresponds to non-zero integers multiple of k modulo kq. When honest participants choose exponents, they are chosen uniformly in a *subset* of Z: they are of the form $2^{254} + 8n$ for $n \in \{0, \ldots, 2^{251} - 1\}$ and $kq > 2^{255}$. Therefore, the distribution for choosing random exponents inside the whole Z is non-uniform, which is indicated by the annotation nonuniform.

We define functions:

fun $exp(G, Z) : G$.
fun $mult(Z, Z) : Z$.
equation builtin commut($mult$).

We have $exp(X, y) = X^y$, and $mult$ is the product modulo kq, in Z. Since its two arguments are non-zero multiples of k, so is its result, and it is in Z. The last line states that the function $mult$ is commutative. (We could add associativity and other properties, like existence of inverses, but commutativity is typically sufficient to prove security of basic Diffie-Hellman key exchanges. More algebraic properties may be needed to prove group Diffie-Hellman protocols, for instance. Note that not modelling these does not restrict the adversary in the computational model.)

fun $pow\_k(G) : G_{sub}$.
fun $exp\_div\_k(G_{sub}, Z) : G_{sub}$.
fun $G_{sub}2G(G_{sub}) : G$ [data].
equation forall $x : G_{sub}, x' : G_{sub}; (G_{sub}2G(x)) = pow\_k(G_{sub}2G(x'))) = (x = x')$.

We have $pow\_k(X) = X^k$, and it is in G_{sub} for all X in G. We have $exp\_div\_k(X, y) = X^{y/k}$. This function operates on G_{sub} and is convenient since the exponents in Z are always multiples of k. The function $G_{sub}2G$ is the identity from G_{sub} to G; it is necessary to convert elements of type G_{sub} to type G. The annotation data tells CryptoVerif that it is injective. The last equation says that pow_k is injective when restricted to the subgroup G_{sub}, of order q. Indeed, k is prime to q, so it can be inverted modulo q.

We also define constants:

const $zero : G$.
const $zero_{sub} : G_{sub}$.
equation $zero = G_{sub}2G(zero_{sub})$.
const $g : G$.
const $g\_k : G_{sub}$.
equation $pow\_k(g) = g\_k$.
equation $g\_k \neq zero_{sub}$.

The neutral element is $zero$ as an element of G and $zero_{sub}$ as an element of G_{sub}. The base point is g, and $g\_k = g^k$.

We also state equations that hold on these functions:

equation forall $X : G, y : Z; exp(X, y) = G_{sub}2G(exp\_div\_k(pow\_k(X), y))$. (1)

equation forall $X : G_{sub}, y : Z, z : Z;$
$exp\_div\_k(pow\_k(G_{sub}2G(exp\_div\_k(X, y))), z) = exp\_div\_k(X, mult(y, z))$. (2)

Equation (1) says that $X^y = (X^k)^{y/k}$ and Equation (2) that $((X^{y/k})^k)^{z/k} = X^{y.z/k}$. Equation (2) applies in particular to simplify $exp(exp(X, y), z)$ after applying (1): $exp(exp(X, y), z) = G_{sub}2G($ $exp\_div\_k(pow\_k(G_{sub}2G(exp\_div\_k(pow\_k(X), y))), z)) = G_{sub}2G(exp\_div\_k(pow\_k(X),$ $mult(y, z)))$. These equations are used by CryptoVerif as rewrite rules, to rewrite the left-hand side into the right-hand side. They allow to rewrite computations in the group G into computations that happen in the subgroup G_{sub}, after raising the public key to the power k. In particular, $exp(g, y) = G_{sub}2G(exp\_div\_k(g\_k, y))$ and $exp(exp(g, y), z) = G_{sub}2G(exp\_div\_k(g\_k, mult(y, z)))$.

The next equations allow CryptoVerif to simplify equality tests with the neutral element, which are used by some protocols, including WireGuard, to exclude that element from the allowed public keys.

equation forall $X : G_{sub}, y : Z; (exp\_div\_k(X, y) = zero_{sub}) = (X = zero_{sub})$.
equation forall $X : G_{sub}, y : Z; (exp\_div\_k(X, y) \neq zero_{sub}) = (X \neq zero_{sub})$.

When $y \in Z$, $y = ky'$ for some y' not multiple of q, so y' is invertible modulo q. Therefore, $X^{y/k} = 0$ if and only if $X^{y'} = 0$ if and only if $(X^{y'})^{1/y'} = 0^{1/y'}$, that is, $X = 0$.

Other properties serve to simplify equalities between Diffie-Hellman values in G_{sub}, with the goal of showing that these equalities are false. When the Diffie-Hellman shared secrets are passed to a random oracle, these equality tests appear after using the random oracle assumption: we compare the arguments of each call to the random oracle with arguments of previous calls, to know whether the random oracle should return the result of a previous call.

equation forall $X : G_{sub}, X' : G_{sub}, y : Z;$
$(exp\_div\_k(X, y) = exp\_div\_k(X', y)) = (X = X')$. (3)

equation forall $X : G_{sub}, x' : Z, y : Z;$
$(exp\_div\_k(X, y) = exp\_div\_k(g\_k, mult(x', y))) =$
$(X = pow\_k(G_{sub}2G(exp\_div\_k(g\_k, x'))))$. (4)

equation forall $X : G_{sub}, y : Z, z : Z;$
$(exp\_div\_k(X, y) = exp\_div\_k(X, z)) = ((y = z) \vee (X = zero_{sub}))$. (5)

collision $x \stackrel{R}{\leftarrow} Z$; **forall** $X : G_{sub}, Y : G_{sub};$
return$(exp\_div\_k(X, x) = Y) \approx_{\text{Pcoll1rand}(Z)}$ **return**$((X = zero_{sub}) \wedge (Y = zero_{sub}))$ (6)
if X independent-of $x \wedge Y$ independent-of x.

Equation (3) holds because $y = ky'$ for some y' invertible modulo q as shown above. In particular, using (1), injectivity of $G_{sub}2G$, and (3), $exp(X, y) = exp(X', y)$ simplifies into $pow\_k(X) = pow\_k(X')$. In contrast, in a prime order group, $exp(X, y) = exp(X', y)$ implies $X = X'$. This is the reason why, in the identity mis-binding issue of Section 6, we fail to prove equality of the public keys $X = X'$ and can only prove $pow\_k(X) = pow\_k(X')$.

Equation (4) is a particular case of (3) when $X' = g\_k^{x'} = pow\_k(G_{sub}2G(exp\_div\_k(g\_k, x')))$. However, CryptoVerif would not apply (3) to a term of the form $exp\_div\_k(X, y) = exp\_div\_k(g\_k, mult(x', y))$.

Equation (5) holds because, when $X \in G_{sub}$ is different from 0, X is a generator of G_{sub}, so all elements $X^{y'}$ for $y' \in [1, q-1]$ are distinct, hence all elements $X^{y/k}$ as well.

In the collision statement (6), Pcoll1rand(Z) is the probability that a randomly chosen element x in Z is equal to an element of Z independent of x. For Curve25519, since random exponents are chosen uniformly among a set of 2^{251} elements, Pcoll1rand(Z) $= 2^{-251}$. Statement (6) means that the probability of distinguishing $exp\_div\_k(X, x) = Y$ from $(X = zero_{sub}) \wedge (Y = zero_{sub})$ is at most Pcoll1rand(Z) assuming X is chosen randomly in Z ($x \stackrel{R}{\leftarrow} Z$) and X and Y are independent of x. Indeed, suppose that $exp\_div\_k(X, x) = Y$ differs from $(X = zero_{sub}) \wedge (Y = zero_{sub})$. If $X = 0$, then $X^{x/k} = 0$, so both expressions reduce to $Y = 0$, so they cannot differ. Therefore, $X \neq 0$. The second expression is then false. Moreover, X is a generator of G_{sub}, so $Y = X^y$ for some y independent of x. The equality $X^{x/k} = Y = X^y$ holds if and only if $x/k = y \mod q$ so $x = ky \mod kq$ with ky independent of x, so this happens with probability Pcoll1rand(Z). So the first expression is true with probability Pcoll1rand(Z), and the two expressions differ with that probability. The support for side-conditions in collision statements is an extension of CryptoVerif that we implemented.

Finally, our model includes properties for simplifying equalities between products in Z. Such equalities appear for instance after simplification of equalities $exp\_div\_k(g\_k, mult(x, y)) = exp\_div\_k(g\_k, mult(x', y'))$.

$$\textbf{equation forall } x : Z, y : Z, y' : Z; (mult(x, y) = mult(x, y')) = (y = y'). \tag{7}$$

$$\begin{array}{l}\textbf{collision } x \stackrel{R}{\leftarrow} Z; \textbf{forall } y : Z, z : Z; \\ \quad \textbf{return}(mult(x, y) = z) \approx_{\mathrm{Pcoll1rand}(Z)} \textbf{return}(\mathsf{false}) \\ \quad \textbf{if } y \text{ independent-of } x \wedge z \text{ independent-of } x.\end{array} \tag{8}$$

$$\begin{array}{l}\textbf{collision } x \stackrel{R}{\leftarrow} Z; y \stackrel{R}{\leftarrow} Z; [\text{random\_choices\_may\_be\_equal}] \textbf{ forall } z : Z; \\ \quad \textbf{return}(mult(x, y) = z) \approx_{2 \times \mathrm{Pcoll1rand}(Z)} \textbf{return}(\mathsf{false}) \\ \quad \textbf{if } z \text{ independent-of } x \vee z \text{ independent-of } y.\end{array} \tag{9}$$

Equation (7) holds because $x = kx'$ for some x' invertible modulo q. The equality $xy = xy'$ in Z means $kx'y = kx'y' \mod kq$, that is, $x'y = x'y' \mod q$, so $y = y' \mod q$, so $y = y' \mod kq$ since y and y' are multiples of k.

In statement (8), $x = kx'$, $y = ky'$, and $z = kz'$ for some x', y', and z' invertible modulo q. Therefore, the equality $x.y = z$ means $kx'ky' = kz' \mod kq$, that is, $x'ky' = z' \mod q$, so $x' = z'/(ky') \mod q$ since k is also invertible modulo q. Letting $x'' = z'/(ky') \mod q$, we have $x = k.x'' \mod kq$. Since $x'' = z'/(ky') \mod q$ is independent of x, so is $k.x''$, and therefore the equality $x = k.x'' \mod kq$ has probability Pcoll1rand(Z) to happen.

In statement (9), the annotation [random_choices_may_be_equal] means that the random choices $x \stackrel{R}{\leftarrow} Z$ and $y \stackrel{R}{\leftarrow} Z$ may be either independent or the same random choice. (Without this annotation, they would necessarily be independent.) When they are independent, this statement is a consequence of the previous one. Suppose that they are the same random choice: $x = y$. Let $x = kx'$ and $z = kz'$ for some x' and z' invertible modulo q. If $xx = z$, then $kx'.kx' = kz'$

mod kq, that is, $x'^2 = z'/k \mod q$. (k is invertible modulo q.) In $\mathbb{Z}_q^*$, half of the elements are square, and each square has two square roots. If z'/k is not a square, the equality $x'^2 = z'/k$ is false. If z'/k is a square, let z'' such that $z'/k = z''^2$ in $\mathbb{Z}_q^*$. Then $x'^2 = z''^2 \mod q$ if and only if $x' = z'' \mod q$ or $x' = -z'' \mod q$, that is, $x = kz'' \mod kq$ or $x = -kz'' \mod kq$. Moreover, z'' is independent of x, so this happens with probability at most $2 \times \mathrm{Pcoll1rand}(Z)$. Grouping the two cases in a single collision statement allows CryptoVerif to apply it even when it cannot determine whether the random choices are independent or identical: when x and y are two cells of the same array with indices that may be different or equal.

When CryptoVerif transforms a game using a Diffie-Hellman assumption on Curve25519, it renames *exp_div_k* into *exp_div_k'*, in order to prevent the repeated application of the same game transformation. Hence, we define a symbol *exp_div_k'* with the same equations and collision statements as *exp_div_k*.

This model is included as a macro in CryptoVerif's library of cryptographic primitives, so that it can easily be reused. It also applies to other curves that have a similar structure, for instance Curve448, which is also used by the Noise framework, and by other protocols like TLS 1.3.

We assume that the prime order subgroup G_{sub} satisfies the gap Diffie-Hellman (GDH) assumption [32]. This assumption means that given a generator g, g^a, and g^b for random a, b, the adversary has a negligible probability to compute g^{ab}, even when the adversary has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$. It was already modelled in CryptoVerif.

In contrast, in their cryptographic proof of WireGuard, Dowling and Paterson [20] use the PRF-ODH assumption. We use the GDH and random oracle assumptions instead because CryptoVerif cannot currently use the PRF-ODH assumption in scenarios with key compromise. While in principle the PRF-ODH assumption is weaker, Brendel et al. [14] show that it is implausible to instantiate the PRF-ODH assumption without a random oracle, so our assumptions and the one of [20] are in fact fairly similar.

4 Indifferentiability of Hash Chains

Before modelling WireGuard, we first present a different, equally precise, formulation of hash chains that is more amenable to a mechanised proof in CryptoVerif. Indeed, WireGuard makes many hash oracle calls to BLAKE2s, and at each call to a random oracle, CryptoVerif tests whether the arguments are the same as in any other previous random oracle call (to return the previous result of the random oracle). Therefore, using directly BLAKE2s as a random oracle would introduce a very large number of cases and yield exaggeratedly large cryptographic games. In order to avoid that, we simplify the random oracle calls using indifferentiability lemmas. These lemmas are not specific to WireGuard and can be used to simplify sequences of random oracle calls in other protocols, including other Noise protocols and Signal [29]. In the future, these lemmas may serve as a basis for an indifferentiability prover inside CryptoVerif, which would simplify random oracle calls before proving the protocol.

Specifically, WireGuard uses HKDF in a chain of calls to derive symmetric keys at different stages of the protocol:

$$
\begin{array}{ll}
C_0 & \leftarrow \mathsf{const} \\
C_1 & \leftarrow \mathsf{hkdf}_1(C_0, v_0) \\
C_2 \| k_1 & \leftarrow \mathsf{hkdf}_2(C_1, v_1) \\
C_3 \| k_2 & \leftarrow \mathsf{hkdf}_2(C_2, v_2) \\
C_4 & \leftarrow \mathsf{hkdf}_1(C_3, v_3)
\end{array}
$$

$$\begin{aligned}
C_5 &\leftarrow \mathsf{hkdf}_1(C_4, v_4)\\
C_6 &\leftarrow \mathsf{hkdf}_1(C_5, v_5)\\
C_7\|\pi\|k_3 &\leftarrow \mathsf{hkdf}_3(C_6, v_6)\\
T^{\rightarrow}\|T^{\leftarrow} &\leftarrow \mathsf{hkdf}_2(C_7, v_7)
\end{aligned}$$

We show, using the indifferentiability lemmas of this section, that hkdf_n is indifferentiable from a random oracle, and that the chain above is indifferentiable from:

$$\begin{aligned}
k_1 &\leftarrow \mathsf{chain}'_1(v_0, v_1)\\
k_2 &\leftarrow \mathsf{chain}'_2(v_0, v_1, v_2)\\
\pi\|k_3\|T^{\rightarrow}\|T^{\leftarrow} &\leftarrow \mathsf{chain}'_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)
\end{aligned} \tag{10}$$

Thus, we obtain a much simpler computation, which we use in our CryptoVerif model of WireGuard. Previous analyses of WireGuard did not use such a result because they do not rely on the random oracle model: [20] relies on the PRF-ODH assumption, [19] uses the symbolic model.

4.1 Definition of Indifferentiability

Indifferentiability can be defined as follows. This definition is an extension of [16] to several independent oracles.

Definition 1 (Indifferentiability)**.** *Functions $(F_i)_{1\leq i\leq n}$ with oracle access to independent random oracles $(H_j)_{1\leq j\leq m}$ are $(t_D, t_S, (q_{H_j})_{1\leq j\leq m}, (q_{F_i})_{1\leq i\leq n}, (q_{H'_i})_{1\leq i\leq n}, \epsilon)$-indifferentiable from independent random oracles $(H'_i)_{1\leq i\leq n}$ if there exists a simulator S such that for any distinguisher D*

$$|\Pr[D^{(F_i)_{1\leq i\leq n},(H_j)_{1\leq j\leq m}} = 1] - \Pr[D^{(H'_i)_{1\leq i\leq n},S} = 1]| \leq \epsilon$$

The simulator S has oracle access to $(H'_i)_{1\leq i\leq n}$, makes at most $q_{H'_i}$ queries to H'_i, and runs in time t_S. The distinguisher D runs in time t_D and makes at most q_{H_j} queries to H_j for $1 \leq j \leq m$ and q_{F_i} queries to F_i for $1 \leq i \leq n$.

In the game $G_0 = D^{(F_i)_{1\leq i\leq n},(H_j)_{1\leq j\leq m}}$, the distinguisher interacts with the real functions F_i and the random oracles H_j from which the functions F_i are defined. In the game $G_1 = D^{(H'_i)_{1\leq i\leq n},S}$, the distinguisher interacts with independent random oracles H'_i instead of F_i, and with a simulator S, which simulates the behaviour of the random oracles H_j using calls to H'_i. (We may also present S as m simulators S_j that each simulate a single random oracle H_j using calls to H'_i, $1 \leq i \leq n$; these simulators share a common state.) Indifferentiability means that these two games are indistinguishable.

4.2 Basic Lemmas

In this section, we show several basic indifferentiability lemmas, which are not specific to WireGuard. The proofs that are not included in this section can be found in the appendix. Lemma 1 shows that random oracle calls with disjoint domains are indifferentiable from calls to independent random oracles.

Lemma 1 (Version of [24, Lemma 2] with more precise evaluation of numbers of oracle calls)**.** *If H is a random oracle, then the functions $H_1, \ldots, H_n$ defined as H on disjoint subsets $D_1, \ldots, D_n$ of the domain D of H are $(t_D, t_S, q_H, (q_{H_i})_{1\leq i\leq n}, (q'_{H_i})_{1\leq i\leq n}, 0)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q_H)$ assuming one can determine in constant time to which subset D_i an element belongs, and q'_{H_i} is the number of requests to H in domain D_i made by the distinguisher. Hence $q'_{H_1} + \cdots + q'_{H_n} \leq q_H$, so in the worst case q'_{H_i} is bounded by q_H.*

Lemma 2 shows that the concatenation of two independent random oracle calls is indifferentiable from a random oracle.

Lemma 2. *If H_1 and H_2 are independent random oracles with the same domain that return bitstrings of length l_1 and l_2 respectively, then the concatenation H' of H_1 and H_2 is $(t_D, t_S, (q_{H_1}, q_{H_2}), q_{H'}, q_{H_1} + q_{H_2}, 0)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_{H_1} + q_{H_2})$.*

Conversely, Lemma 3 shows that splitting the output of a random oracle into two fixed length outputs yields independent random oracles.

Lemma 3. *If H is a random oracle that returns bitstrings of length l, then the function H'_1 returning the first l_1 bits of H and the function H'_2 returning the last $l - l_1$ bits of H are $(t_D, t_S, q_H, (q_{H'_1}, q_{H'_2}), (q_H, q_H), 0)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q_H)$.*

As a particular consequence, Lemma 4 shows that the truncation of a random oracle is indifferentiable from a random oracle.

Lemma 4 (Already stated in [24, Lemma 3])**.** *If H is a random oracle that returns bitstrings of length l, then the truncation H' of H to length $l' < l$ is $(t_D, t_S, q_H, q_{H'}, q_H, 0)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_H)$.*

Lemmas 5 and 6 deal with the composition of two random oracle calls in sequence. We extended CryptoVerif to be able to prove indistinguishability between two games given by the user. Thanks to this extension, CryptoVerif helps considerably with the proof of these lemmas: it shows the indistinguishability result between the games G_0 and G_1 described in Section 4.1, which implies the indifferentiability result. We present the proof of Lemma 5 as an illustration.

Lemma 5. *If $H_1 : S_1 \to S'_1$ and $H_2 : S'_1 \times S_2 \to S'_2$ are independent random oracles, then H_3 defined by $H_3(x, y) = H_2(H_1(x), y)$ is $(t_D, t_S, (q_{H_1}, q_{H_2}), q_{H_3}, q_{H_2}, \epsilon)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_{H_1} q_{H_2})$ and $\epsilon = (2q_{H_2} q_{H_1} + q_{H_1}^2 + q_{H_2} q_{H_3} + q_{H_3}^2)/|S'_1|$.*

Proof. Consider

- the game G_0 in which H_1 and H_2 are independent random oracles, and $H_3(x, y) = H_2(H_1(x), y)$, and
- the game G_1 in which H_3 is a random oracle; two lists L_1 and L_2 that are initially empty; $H_1(x)$ returns y if $(x, y) \in L_1$ for some y, and otherwise chooses a fresh random r in S'_1, adds (x, r) to L and returns r; $H_2(y, z)$ returns $H_3(x, z)$ if $(x, y) \in L_1$ for some x, otherwise returns u if $((y, z), u) \in L_2$ for some u, and otherwise chooses a fresh random r in S'_2, adds $((y, z), r)$ to L_2 and returns r.

CryptoVerif shows that the games G_0 and G_1 are indistinguishable, up to probability ϵ. □

Lemma 6. *If $H_1 : S_1 \to S'_1$ and $H_2 : S'_1 \times S_1 \to S'_2$ are independent random oracles, then $H'_1 = H_1$ and H'_2 defined by $H'_2(x) = H_2(H_1(x), x)$ are $(t_D, t_S, (q_{H_1}, q_{H_2}), (q_{H'_1}, q_{H'_2}), (q_{H_1} + q_{H_2}, q_{H_2}), \epsilon)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q_{H_2})$ and $\epsilon = q_{H_2}(2q_{H_1} + 2q_{H'_1} + q_{H'_2} + 1)/|S'_1|$.*

4.3 Indifferentiability of HKDF

The hkdf key derivation function is defined as follows [26]:

$$\begin{aligned}&\mathsf{hkdf}_n(salt, key, info) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad prk = \mathsf{hmac}(salt, key)\\ &\quad k_1 = \mathsf{hmac}(prk, info \| i_0)\\ &\quad k_{i+1} = \mathsf{hmac}(prk, k_i \| info \| i + i_0) \text{ for } 1 \leq i < n\end{aligned}$$

where $n \leq 255$, and $i_0 = 0x01$ and i are of size 1 byte. In WireGuard, *info* is always empty, so we omit it in Section 2.

We suppose that hmac is a random oracle, and we show that hkdf_n is indifferentiable from a random oracle, with the additional assumption that the calls to hmac use disjoint domains. (We show that this assumption is necessary and give a full proof of the result in the appendix.) Let $\mathcal{S}$, $\mathcal{K}$, and $\mathcal{I}$ be the sets of possible values of *salt*, *key*, and *info* respectively, and $\mathcal{M}$ the output of hmac.

Lemma 7. *If* hmac *is a random oracle and* $\mathcal{K} \cap (\mathcal{I}\|i_0 \cup \bigcup_{i=1}^{n-1} \mathcal{M}\|\mathcal{I}\|i + i_0) = \emptyset$ *then* hkdf_n *with domain* $\mathcal{S} \times \mathcal{K} \times \mathcal{I}$ *is* $(t_D, t_S, q_{\mathsf{hmac}}, q_{\mathsf{hkdf}_n}, 2q_{\mathsf{hmac}}, \epsilon)$*-indifferentiable from a random oracle, where* $t_S = \mathcal{O}(q_{\mathsf{hmac}}^2)$ *and* $\epsilon = (2q_{\mathsf{hmac}} + q_{\mathsf{hkdf}_n})^2/|\mathcal{M}|$*,*

This result extends the proof given for hkdf_2 in [24, Lemma 1]. Moreover, our proof is modular and partly made using CryptoVerif, thanks to the basic lemmas of Section 4.2.

Proof sketch. Since the domains are disjoint, by Lemma 1, the $(n + 1)$ calls to hmac are indifferentiable from independent random oracles $H_0, \ldots, H_n$. The constant $i + i_0$ can be removed from the arguments of H_{i+1} since it is fixed for a given H_{i+1}. By Lemma 6, the computation of $k_2 = H_2(H_1(prk, info), prk, info)$ is indifferentiable from a random oracle $k_2 = H_2'(prk, info)$. Applying this reasoning n times, the computation of k_i for $1 \leq i \leq n$ is indifferentiable from independent random oracles $k_i = H_i'(prk, info)$. By Lemma 2, concatenation of H_i' for $1 \leq i \leq n$ is indifferentiable from a random oracle H, so $\mathsf{hkdf}_n(salt, key, info) = k_1 \| \ldots \| k_n = H(prk, info)$, where $prk = H_0(salt, key)$. By Lemma 5, we conclude that hkdf_n is indifferentiable from a random oracle. □

4.4 Indifferentiability of a Chain of Random Oracle Calls

In this section, we prove the indifferentiability of a chain of random oracle calls defined as follows.

Definition 2 (Chain)**.** *Let* $m \geq 1$ *be a fixed integer, let* C *and* C_j *with* $0 \leq j \leq m+1$ *be bitstrings of length* l'*, let* v_j *with* $0 \leq j \leq m$ *be bitstrings of arbitrary length, let* l *be the length of the output of* $\mathsf{H}(C_j, v_j)$*, and let* r_j *with* $0 \leq j \leq m$ *be bitstrings of length* $(l - l')$*. (*l *and* l' *are functions of the security parameter.) We define the functions* $\mathsf{chain}_n, 0 \leq n < m$ *and the function* chain_m *in the following way:*

$$\begin{aligned}&\mathsf{chain}_n(v_0, \ldots, v_n) =\\ &\quad C_0 = \mathsf{const}\\ &\quad \textit{for } j = 0 \textit{ to } n \textit{ do } C_{j+1} \| r_j = \mathsf{H}(C_j, v_j)\\ &\quad \textit{return } r_n\end{aligned} \tag{11}$$

$$\begin{aligned}&\mathsf{chain}_m(v_0, \ldots, v_m) =\\ &\quad C_0 = \mathsf{const}\\ &\quad \textit{for } j = 0 \textit{ to } m \textit{ do } C_{j+1} \| r_j = \mathsf{H}(C_j, v_j)\\ &\quad \textit{return } C_{m+1} \| r_m\end{aligned} \tag{12}$$

The functions $\mathsf{chain}_n, n < m$*, have an output of length* $(l - l')$*, and the output length of* chain_m *is* l*.*

Lemma 8. *If* H *is a random oracle, then* chain_n*, for* $n \leq m$*, are* $(t_D, t_S, q_\mathsf{H}, (q_{\mathsf{chain}_n})_{0 \leq n \leq m}, (q_\mathsf{H})_{0 \leq n \leq m}, \epsilon)$*-indifferentiable from independent random oracles, where* $t_S = \mathcal{O}(q_\mathsf{H}^2)$ *and* $\epsilon = \left(\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}\right) \cdot q_\mathsf{H} + q_\mathsf{H}^2\right) / 2^{l'}$*.*

This lemma is proved in the appendix. We could probably prove it for small values of m using CryptoVerif, but the generic result requires a manual proof because CryptoVerif does not support loops.

4.5 Application to WireGuard

WireGuard employs BLAKE2s [1] both directly as the function hash and indirectly as hash function in hmac and thus also in hkdf. In our proof, we assume that hash is collision-resistant and use the random oracle assumption for usages of BLAKE2s via hkdf. Rigorously, to be able to use two distinct assumptions, we need the domains of these two uses to be disjoint. This is true in WireGuard: the length of the argument of hash is 64 bytes for $\mathsf{hash}(H_0 \| S_r^{pub})$, $\mathsf{hash}(H_1 \| E_i^{pub})$, $\mathsf{hash}(H_4 \| E_r^{pub})$, and $\mathsf{hash}(H_5 \| \pi)$, 80 bytes for $\mathsf{hash}(H_2 \| S_{i\,\blacksquare}^{pub})$, 60 bytes for $\mathsf{hash}(H_3 \| ts_\blacksquare)$, 48 bytes for $\mathsf{hash}(H_6 \| empty_\blacksquare)$, 40 bytes for $\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_r^{pub})$, $\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_i^{pub})$, and $\mathsf{hash}(\mathsf{label}_{\mathsf{cookie}} \| S_m^{pub})$. In contrast, the length of the argument of $\mathsf{BLAKE2s}$ in $\mathsf{hmac}(k, m)$ is 96 bytes or $64 + \mathsf{length}(m)$, and in the computation of hkdf, *info* is empty in WireGuard, so the length of m is 32 bytes (*key*), 1 byte ($info \| i_0$) or 33 bytes ($k_i \| info \| i + i_0$), so the length of the argument of $\mathsf{BLAKE2s}$ in the computation of hkdf is 96, 65, or 97 bytes.

Then by Lemma 1, we can consider two independent random oracles, hash for the direct uses and hash' for the uses via hkdf. Since hash is a random oracle, it is a fortiori collision-resistant.

Since hash' is a random oracle, $\mathsf{hmac\text{-}hash}'$ is indifferentiable from a random oracle by [17, Theorem 3]

Moreover, the domains of the calls to hmac in hkdf_n are disjoint. Indeed, $\mathcal{K}$ consists of bitstrings of length 32 bytes, $\mathcal{I} \| i_0$ consists of bitstrings of length 1 byte, and $\mathcal{M} \| \mathcal{I} \| i + i_0$ consists of bitstrings of length 33 bytes. By Lemma 7, hkdf_n is indifferentiable from a random oracle.

In this section, we use the $\|$ operator is used to concatenate blocks of 32 bytes and use the placeholder _ for one unnamed 32-byte block. Since we prove Lemma 8 for a chain of calls to the same hkdf_n function, we rewrite the chain of hkdf calls in WireGuard to use only calls to hkdf_3, as 3 is the maximum number of outputs needed:

$$
\begin{array}{ll}
C_0 & \leftarrow \mathsf{const} \\
C_1 \| \_ \| \_ & \leftarrow \mathsf{hkdf}_3(C_0, v_0) \\
C_2 \| k_1 \| \_ & \leftarrow \mathsf{hkdf}_3(C_1, v_1) \\
C_3 \| k_2 \| \_ & \leftarrow \mathsf{hkdf}_3(C_2, v_2) \\
C_4 \| \_ \| \_ & \leftarrow \mathsf{hkdf}_3(C_3, v_3) \\
C_5 \| \_ \| \_ & \leftarrow \mathsf{hkdf}_3(C_4, v_4) \\
C_6 \| \_ \| \_ & \leftarrow \mathsf{hkdf}_3(C_5, v_5) \\
C_7 \| \pi \| k_3 & \leftarrow \mathsf{hkdf}_3(C_6, v_6) \\
T^{\rightarrow} \| T^{\leftarrow} \| \_ & \leftarrow \mathsf{hkdf}_3(C_7, v_7)
\end{array}
$$

Because of the way hkdf_n is constructed, this is actually the same computation.

By Lemma 8, the computation above can be replaced with the following one:

$$\begin{array}{ll}
_\|_ & \leftarrow \mathsf{chain}_0(v_0) \\
k_1\|_ & \leftarrow \mathsf{chain}_1(v_0, v_1) \\
k_2\|_ & \leftarrow \mathsf{chain}_2(v_0, v_1, v_2) \\
_\|_ & \leftarrow \mathsf{chain}_3(v_0, v_1, v_2, v_3) \\
_\|_ & \leftarrow \mathsf{chain}_4(v_0, v_1, v_2, v_3, v_4) \\
_\|_ & \leftarrow \mathsf{chain}_5(v_0, v_1, v_2, v_3, v_4, v_5) \\
\pi\|k_3 & \leftarrow \mathsf{chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \\
T^{\rightarrow}\|T^{\leftarrow}\|_ & \leftarrow \mathsf{chain}_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{array}$$

where chain_i for $i \leq 7$ are independent random oracles.

The output of the random oracles can be truncated by Lemma 4 to avoid having to throw away parts of the output:

$$\begin{array}{ll}
k_1 & \leftarrow \mathsf{chain}'_1(v_0, v_1) \\
k_2 & \leftarrow \mathsf{chain}'_2(v_0, v_1, v_2) \\
\pi\|k_3 & \leftarrow \mathsf{chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \\
T^{\rightarrow}\|T^{\leftarrow} & \leftarrow \mathsf{chain}'_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{array}$$

In WireGuard, $v_7 = \mathit{empty}$, so $T^{\rightarrow}$ and $T^{\leftarrow}$ only depend on $v_0, \ldots, v_6$, as do π and k_3 in the previous line. By Lemma 2, we can replace the last two lines by one random oracle call:

$$\begin{array}{ll}
k_1 & \leftarrow \mathsf{chain}'_1(v_0, v_1) \\
k_2 & \leftarrow \mathsf{chain}'_2(v_0, v_1, v_2) \\
\pi\|k_3\|T^{\rightarrow}\|T^{\leftarrow} & \leftarrow \mathsf{chain}'_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)
\end{array}$$

5 Modelling WireGuard

This section presents our model of the WireGuard protocol in CryptoVerif. We prove security properties for that model in Section 6.

5.1 Execution Environment

In our model, we consider two honest entities A and B. In the initial setup, we generate the static key pairs for these two entities and publish their public keys, so that the adversary can use them. After this setup, we run parallel processes that represent a number of executions of A and B polynomial in the security parameter.

The entities A and B can play both the initiator and responder role. These two entities can run WireGuard between each other, but also with any number of dishonest entities included in the adversary: for each session, the adversary sends to the initiator its *partner public key*, that is, the public key of the entity with which it should start a session; the adversary sends to the responder the set of partner public keys that it accepts messages from.

This setting allows us to prove security for any sessions between two honest entities, in a system that may contain any number of (honest or dishonest) other entities. We prove security

for sessions in which A is the initiator and B is the responder. We do not explicitly prove security for sessions in which B is the initiator and A is the responder, but the same security properties hold by symmetry.

The processes for the entities A and B model the entire protocol, including the first two protocol messages, the key confirmation message from the initiator, and then a number of transport data messages polynomial in the security parameter, in both directions between initiator and responder. The model also includes random oracles, and we allow the adversary to call any of the random oracles that we use.

We consider 3 variants of this model:

Variant 1. This variant does not rely at all on the pre-shared key for proving security, so A and B receive a pre-shared key chosen by the adversary at the beginning of each execution. That allows the adversary to model both the absence of a pre-shared key (by choosing the value 0) or a compromised pre-shared key of its choice.

We model the dynamic compromise of the private static key of A (resp. B) by a process that the adversary can call at any time and that returns the private key of A (resp. B) and records the compromise by defining a particular variable, so that it can be tested in the security properties that we consider.

In WireGuard, four Diffie-Hellman operations and the pre-shared key contribute to the session keys. If the pre-shared key is not used or compromised, security is based on the four Diffie-Hellman operations. If one of them cannot be computed by the adversary, then the session keys are secret. Therefore, we consider all combinations of compromises but those where both keys on one side are compromised, that is:

1. A and B's private static keys may be dynamically compromised;
2. A's private static key may be dynamically compromised and B's private ephemeral key is compromised (by sending it to the adversary as soon as it is chosen);
3. B's private static key may be dynamically compromised and A's private ephemeral key is compromised;
4. A and B's private ephemeral keys are compromised.

We prove most security properties for *clean* sessions, that is, intuitively, sessions between honest entities; cleanliness is the minimal assumption needed to hope for security. A session of A is clean when either B's private static key is not compromised yet and A's partner public key is equivalent to B's static public key, or B's private static key is compromised and the public ephemeral key received by A is equivalent to a non-compromised ephemeral generated by B. B's session cleanliness is defined symmetrically. Intuitively, when B's private static key is not compromised, A can rely on that key to authenticate B, so A thinks she talks to B when she runs a session with B's public key. We consider a public key equivalent to B's public key rather than equal to B's public key to strengthen the properties: the authentication property shown in Section 6 then implies that when A successfully runs a session with a partner public key equivalent to B's public key, then these two keys are in fact equal. (We find an interesting scenario concerning equivalent public keys and identity mis-binding with variant 3 of our model, we discuss it in §6.) When B's private static key is compromised, A cannot authenticate B, but we can still prove security when the ephemeral key received by A has been generated by B. Like for static keys, when A successfully runs a session with a received ephemeral equivalent to an ephemeral generated by B, then these two ephemerals are in fact equal. (Instead of considering compromised ephemeral keys, we could also have modelled dishonestly generated ephemeral keys. We expect that some properties shown in §6, such as session uniqueness, would not hold in this case.)

Variant 2. This variant relies exclusively on the pre-shared key for security. In that variant, we consider all private static and ephemeral keys as always compromised. We choose a pre-shared key randomly in the initial setup, and run sessions between A and B with that pre-shared key. In this model, A's partner public key is always B's public key and symmetrically, and these sessions between A and B are always considered clean. The adversary can run A's and B's sessions with other entities since A and B's private static keys are compromised and these sessions use a different pre-shared key.

Variant 3. In this variant, all keys are compromised: all private static and ephemeral keys are always compromised and the pre-shared key is chosen by the adversary for each session. This model is useful for proving properties that do not rely on session cleanliness, that is, properties that hold even for sessions involving dishonest participants.

With this model, we analyse the whole WireGuard protocol as it is, tying together the authenticated key exchange and the transport data phase. A similar approach was chosen by the creators of the Authenticated and Confidential Channel Establishment (ACCE) [23] model to analyse TLS. Instead of reasoning about key indistinguishability, ACCE looks at the security of the messages exchanged encrypted using the key. We do the same, for the key confirmation and all subsequent transport data messages.

In ACCE, the adversary has to choose one clean test session in which it tries to break security by determining the secret bit. In all other sessions, it is allowed to reveal the session keys. In our model, all clean sessions are test sessions, and we explicitly reveal the session keys in sessions that are not clean.

5.2 Modelling Tricks

Apart from the HKDF chains where we prove that the way we model them is indifferentiable from the real protocol in Section 4, we use the following modelling tricks:

- Timestamps: CryptoVerif has no support for time, so instead of generating the timestamp, we input it from the adversary. In other words, we delegate the task of timestamp generation to the adversary. In order to model replay protection for the first message, the responder stores a global table (that is, a list) of triples containing the received timestamp, the partner public key for that session, as well as its own public key. (This is equivalent to having a distinct table of timestamps and partner public keys for each responder, represented by its public key.) The responder rejects the first message when the triple (received timestamp, partner public key, and responder public key) is already in the table.

- Nonces for the AEAD scheme: The nonces in WireGuard are computed by incrementing a counter. CryptoVerif has no support for that, so we receive the desired value of the counter from the adversary. We guarantee that the same counter is never used twice in the same session for sending messages by storing all counters used for sending messages in a table of pairs (session index, counter), where the session index identifies the session uniquely: it indicates whether A or B is running, as initiator or as responder, and contains a unique integer index for the execution of that entity in that role. This is equivalent to having a distinct table of counters for each session. The message is not sent when the adversary provides a counter that is already in the table. We guarantee that the same counter is never used twice for receiving messages in the same way, using a separate table.

- We omit the MACs mac_1 and mac_2 in our model. This simplifies the proof but preserves its soundness, since they can be computed and verified by the adversary: we deliver the

messages without MACs to the adversary, and the adversary can add the MACs; conversely, the adversary can remove the MACs before delivering messages to the protocol model. We let the adversary choose the key R_r that the responder uses for computing cookies. All other elements needed to compute the MACs are public: constants and static public keys. We reintroduce the MACs in a separate model that we use for proving resistance against DoS.

Importantly, these modelling tricks increase the power of the adversary: the implementation done in WireGuard is a particular case of what the adversary can do in our model, in which the adversary chooses the current time as timestamp, increases the counter for sending messages at each emission, accepts incoming counters in a sliding window, and computes and verifies mac_1 and mac_2 by itself. As a result, a security proof in our model remains valid in WireGuard.

6 Verification Results

In order to prove authentication properties, we insert events in our model, to indicate when each message is sent or received by the protocol. Specifically, we insert events sent1, sent2, sent_msg_initiator, and sent_msg_responder just before sending message 1, message 2, and transport messages on the initiator and responder sides respectively, and corresponding events rcvd1, rcvd2, rcvd_msg_responder, and rcvd_msg_initiator when these messages have been received and successfully decrypted. The event rcvd2 and the events for transport messages are executed only in clean sessions.

Mutual key and message authentication, resistance against KCI, resistance against replay from message 2. We show authentication for all messages starting from the second protocol message, by proving the following correspondence properties between events, in the first two variants of our CryptoVerif model of Section 5.1:

$$\begin{aligned}
&\mathsf{inj\text{-}event}(\mathsf{rcvd2}(S_r^{pub}, E_i^{pub}, S_{i\,\blacksquare}^{pub}, S_i^{pub}, ts_{\blacksquare}, ts, E_r^{pub}, empty_{\blacksquare}, T^{\rightarrow}, T^{\leftarrow}))\\
\Rightarrow\ &\mathsf{inj\text{-}event}(\mathsf{sent2}(S_r^{pub}, E_i^{pub}, S_{i\,\blacksquare}^{pub}, S_i^{pub}, ts_{\blacksquare}, ts, E_r^{pub}, empty_{\blacksquare}, T^{\rightarrow}, T^{\leftarrow}))\,,\\
&\mathsf{inj\text{-}event}(\mathsf{rcvd_msg_responder}(S_r^{pub}, E_i^{pub}, S_{i\,\blacksquare}^{pub}, S_i^{pub}, ts_{\blacksquare}, ts,\\
&\qquad E_r^{pub}, empty_{\blacksquare}, T^{\rightarrow}, T^{\leftarrow}, N^{\rightarrow}, P_{\blacksquare}, P))\\
\Rightarrow\ &\mathsf{inj\text{-}event}(\mathsf{sent_msg_initiator}(S_r^{pub}, E_i^{pub}, S_{i\,\blacksquare}^{pub}, S_i^{pub}, ts_{\blacksquare}, ts,\\
&\qquad E_r^{pub}, empty_{\blacksquare}, T^{\rightarrow}, T^{\leftarrow}, N^{\rightarrow}, P_{\blacksquare}, P))\,,
\end{aligned}$$

We also prove a third query (similar to the second one above) for transport data messages in the other direction, with events rcvd_msg_initiator and sent_msg_responder. A proven correspondence between two injective events (inj-event) means that each execution of the left-hand event corresponds to a distinct execution of the right-hand event.

The first query means that, if the initiator session is clean and the initiator has received the second message, then the responder sent it, and initiator and responder agree on their static and ephemeral public keys, session keys, timestamp, and communicated ciphertexts. This authenticates the responder to the initiator.

The second and third queries mean that, if the receiver session is clean and the receiver received a transport packet, then a sender sent that transport packet, and the receiver and the sender agree on their static and ephemeral public keys, session keys, timestamp, sent plaintext, message counter, and communicated ciphertexts. In particular, for the key confirmation message, this

authenticates the initiator to the responder. These queries also provide message authentication for the transport data messages.

All these properties hold when the pre-shared key is not compromised (variant 2 of Section 5.1). They also hold when neither both S_i^{priv} and E_i^{priv} nor both S_r^{priv} and E_r^{priv} are compromised and the receiver session is clean; this is true, in particular, when the sender's static private key is not compromised yet (variant 1 of Section 5.1).

The above queries include resistance against replays because the correspondences are injective: each reception corresponds to a *distinct* emission. They also include resistance against KCI attacks because the rcvd∗ events are issued even if the receiver's static key has already been compromised: the receiver session is still clean in this case. Note that, for the responder, resistance against KCI attacks only starts after it receives the first data transport message. Indeed, the first protocol message is subject to a KCI attack: if the private static key of the responder (S_r^{priv}) is compromised, then the adversary can forge the first message and impersonate the initiator to the responder.

Secrecy and forward secrecy. We show secrecy of transport data messages in clean sessions by a left-or-right message indistinguishability game. In the initial setup, we randomly choose a secret bit. For each transport data message in a clean session, the adversary provides two padded plaintexts of the same length, and we encrypt one of them depending on the value of that bit. CryptoVerif proves the secrecy of that bit, in variants 1 and 2 of Section 5.1, showing that the adversary cannot determine which of the two plaintexts was encrypted.

The secrecy query includes forward secrecy, because we allow dynamic compromise of static keys after the session keys have been established, if the ephemeral key of the same party is not compromised. This assumes that the parties delete the sessions' ephemeral and chaining keys after key derivation.

In variant 2 of our model, the query also shows secrecy provided the pre-shared key is not compromised, even if all other keys (static and ephemeral) are compromised. Our models do not consider the dynamic compromise of the pre-shared key, due to a limitation of CryptoVerif. We can still obtain forward secrecy with respect to the compromise of the pre-shared key using the following manual argument. As mentioned above, variant 2 of our model shows authentication when the pre-shared key is not compromised (all other keys are compromised in this model). This authentication property is preserved when the pre-shared key is compromised after the rcvd∗ event, because the later compromise cannot alter the fact that the sent∗ event has been executed. Furthermore, authentication guarantees that the ephemeral public key received by the initiator was generated by the responder and conversely. Variant 1 of our model then guarantees secrecy in this case, because the session is clean when the ephemeral received by the initiator was generated by the responder and conversely. Hence, we get the desired forward secrecy property: we have message secrecy when the pre-shared key is compromised after the session, and neither both S_i^{priv} and E_i^{priv} nor both S_r^{priv} and E_r^{priv} are compromised.

We cannot prove key secrecy for the session keys in the full protocol, because the session keys are used for encrypting transport data messages, and this allows an adversary to distinguish them from fresh random keys. Instead, we prove key secrecy for a model in which all transport data messages, including key confirmation, are removed. To prove this result, we need to strengthen the session cleanliness condition. Indeed, the first message is subject to a KCI attack, as mentioned above. Therefore, when the private static key of the responder is compromised, we additionally require that the ephemeral received by the responder is equivalent to one generated by the initiator. With this stronger cleanliness condition, we show that the session keys are secret, that is, the keys for various clean sessions are indistinguishable from independent random keys. We do not need this stronger cleanliness condition when we study the full protocol, since the key

confirmation message protects the responder against KCI attacks.

Resistance against replay for the first message. We prove that the first message cannot be replayed but only if no static key is compromised when it is received. If S_i^{priv} were compromised, the adversary can impersonate the initiator as the sender of this message. If S_r^{priv} is compromised, we have a KCI attack, as described above. So we prove the following injective correspondence in a model where the static keys cannot be compromised but the ephemeral keys may be compromised, so we rely on the static-static Diffie-Hellman shared secret:

$$\begin{aligned} &\mathsf{inj\text{-}event}(\mathsf{rcvd1}(\mathit{true}, S_r^{pub}, E_i^{pub}, S_{i\,\text{🔒}}^{pub}, S_i^{pub}, ts_{\text{🔒}}, ts)) \\ \Rightarrow\ &\mathsf{inj\text{-}event}(\mathsf{sent1}(\quad S_r^{pub}, E_i^{pub}, S_{i\,\text{🔒}}^{pub}, S_i^{pub}, ts_{\text{🔒}}, ts))\,. \end{aligned}$$

The first parameter of rcvd1 is *true* if the public static key received by the responder with the first message is the public static key of the honest initiator: we prove this property only for sessions between honest peers. Replay protection is guaranteed by each timestamp being accepted only once. With this check removed, the first message can be replayed, but we still prove a non-injective correspondence between the two events, replacing inj-event by event in the query. This is a weaker property, meaning that, if an event rcvd1 has been executed, then at least one event sent1 with matching parameters has been executed before. Thus, even with the replay protection removed, we can prove that the origin of the first message cannot be forged in a model without static key compromise.

Correctness. Correctness means that, if the adversary does not modify the first two messages, then the initiator and responder share the same session keys and transcript hash H_7. Actually, it suffices that the adversary does not modify the ephemerals and ciphertexts of the first two messages. We prove it with the following query:

$$\begin{aligned} &\mathsf{event}(\mathsf{responder\_corr}(E_i^{pub}, S_{i\,\text{🔒}}^{pub}, ts_{\text{🔒}}, E_r^{pub}, \mathit{empty}_{\text{🔒}}, T_r^{\rightarrow}, T_r^{\leftarrow}, H_{r7})) \\ \wedge\ &\mathsf{event}(\mathsf{initiator\_corr}(E_i^{pub}, S_{i\,\text{🔒}}^{pub}, ts_{\text{🔒}}, E_r^{pub}, \mathit{empty}_{\text{🔒}}, T_i^{\rightarrow}, T_i^{\leftarrow}, H_{i7})) \\ \Rightarrow\ &T_i^{\rightarrow} = T_r^{\rightarrow} \wedge T_i^{\leftarrow} = T_r^{\leftarrow} \wedge H_{i7} = H_{r7}\,. \end{aligned}$$

The events initiator_$*$ and responder_$*$ used in this query and in the following ones are issued after key derivation, in the initiator and responder respectively. Here, the two events given as assumptions guarantee that the adversary did not modify the ephemerals and ciphertexts of the first two messages, and the query concludes that the session keys and transcript hash must be equal. However, in our main models, CryptoVerif is currently unable to prove that the ciphertexts have not been created by the adversary, although this is true in the sessions considered by the correctness query. Thus, we created a separate model to prove correctness, in which the assumption is hard-coded by interleaving the initiator and responder in a single sequential process. In this model, we prove correctness even if all keys are compromised.

Session Uniqueness. First, we prove that there is a single initiator and a single responder session with a given $T^{\rightarrow}$ or $T^{\leftarrow}$. The query below shows that there cannot be two distinct initiator sessions with the same $T^{\rightarrow}$:

$$\begin{aligned} &\mathsf{event}(\mathsf{initiator\_uniq\_T}^{\rightarrow}(i_i, T^{\rightarrow})) \\ \wedge\ &\mathsf{event}(\mathsf{initiator\_uniq\_T}^{\rightarrow}(i_i', T^{\rightarrow})) \Rightarrow i_i = i_i'\,, \end{aligned}$$

where i_i, i_i' are replication indices: CryptoVerif assigns each execution of the initiator (or responder) process a unique replication index, so the query means that if we execute two events initiator_uniq_$\mathsf{T}^{\rightarrow}$ with the same $T^{\rightarrow}$, then they have the same replication index $i_i = i_i'$, hence

they belong to the same session. This query is proved in variant 3 of Section 5.1, so the property holds even if all keys are compromised. (It relies on the choice of a fresh ephemeral at each session.) The queries for the other cases are similar.

Second, we show similarly that there is a single initiator and a single responder session for a given set of publicly transmitted protocol values.

Channel Binding. We prove channel binding with the query:

$$\begin{aligned}&\mathsf{event}(\mathsf{initiator\_H7}(params, H_7))\\ \wedge\ &\mathsf{event}(\mathsf{responder\_H7}(params', H_7)) \Rightarrow params = params'\end{aligned}$$

This query shows that if the initiator and responder have the same value of the session transcript H_7, then they share the same value of all session parameters *params* (static and ephemeral public keys, timestamp, pre-shared key, session keys). This query is also proved in variant 3 of Section 5.1, so the property holds even if all keys are compromised. (It relies on the collision resistance of hash.)

Identity Mis-Binding. For this property, we need to show that if an initiator and a responder session share the same session keys $T^{\rightarrow}$ and $T^{\leftarrow}$, then they share the same view on the ephemeral and static keys used in that session. This is formalised by the following query:

$$\begin{aligned}&\mathsf{event}(\mathsf{responder\_imb}(T^{\rightarrow}, T^{\leftarrow}, E^{pub}_{i,rcvd}, E^{pub}_{r}, S^{pub}_{i,rcvd}, S^{pub}_{r}))\\ \wedge\ &\mathsf{event}(\mathsf{initiator\_imb}(T^{\rightarrow}, T^{\leftarrow}, E^{pub}_{i}, E^{pub}_{r,rcvd}, S^{pub}_{i}, S^{pub}_{r,rcvd}))\\ \Rightarrow\ &E^{pub}_{i} = E^{pub}_{i,rcvd} \wedge E^{pub}_{r} = E^{pub}_{r,rcvd} \wedge S^{pub}_{i} = S^{pub}_{i,rcvd} \wedge S^{pub}_{r} = S^{pub}_{r,rcvd}\,.\end{aligned}$$

CryptoVerif proves it in variant 1 of our model, so it holds when neither both S^{priv}_i and E^{priv}_i nor both S^{priv}_r and E^{priv}_r are compromised. However, the proof fails when all static and ephemeral keys are compromised (variant 3 of our model): CryptoVerif can prove only the weaker property that $pow\_k\left(S^{pub}_i\right) = pow\_k\left(S^{pub}_{i,rcvd}\right)$ and $pow\_k\left(S^{pub}_r\right) = pow\_k\left(S^{pub}_{r,rcvd}\right)$. An adversary can indeed break the equality of public static keys in this case:

- The adversary instructs A to initiate a session to a public static key $S^{pub\prime}_r$ equivalent to our model's honest responder public static key: $pow\_k(S^{pub}_r) = pow\_k(S^{pub\prime}_r)$ but $S^{pub}_r \neq S^{pub\prime}_r$. This is possible because S^{priv}_r is compromised. In this session, the adversary acts as responder, and because the ephemeral is also compromised, gets A's E^{priv}_i.
- The adversary now acts as initiator to start a session with B using a public static key $S^{pub\prime}_i$ equivalent to the honest initiator public static key: $pow\_k(S^{pub}_i) = pow\_k(S^{pub\prime}_i)$ but $S^{pub}_i \neq S^{pub\prime}_i$. This is possible because S^{priv}_i is compromised. The adversary uses E^{priv}_i as ephemeral. The ephemeral of this session is also compromised, so the adversary gets E^{priv}_r.
- The adversary continues the session with A using the ephemeral E^{priv}_r.

If a pre-shared key is used, we assume that the adversary has the same pre-shared key with A (presenting itself with key $S^{pub\prime}_r$) and with B (presenting itself with $S^{pub\prime}_i$). The session keys $T^{\rightarrow}$ and $T^{\leftarrow}$ for these two sessions are computed as hashes of E^{pub}_i, $\mathsf{dh}(E^{priv}_i, S^{pub}_r)$, $\mathsf{dh}(S^{priv}_i, S^{pub}_r)$, E^{pub}_r, $\mathsf{dh}(E^{priv}_i, E^{pub}_r)$, $\mathsf{dh}(S^{priv}_i, E^{pub}_r)$, and psk. They are the same in both sessions, so the session keys are also the same.

This scenario, with a session between A and B' and one between B and A' that share the same session keys, is an instance of a bilateral unknown key-share attack [15] and of a key

synchronisation attack [8]. It appears only when all static and ephemeral Diffie-Hellman keys are compromised, and hence should be considered a corner-case. However, we note that this scenario does not require the *psk* shared by A and B to be compromised, since this *psk* does not get used in the execution above. We suggest a possible fix of this identity mis-binding issue in Section 7.

Resistance against DoS. As described in Section 2, WireGuard provides a cookie mechanism that a peer under load can use to enforce a round trip per sender address, and thus to bind a handshake message to a sender address; this permits per-address rate limiting. We model this mechanism in a separate model in which a responder generates R_r, replies with a cookie $\tau = \mathsf{mac}(R_r, A_i)$ upon receipt of messages 1 from A_i with zero mac_2, and verifies mac_2 upon receipt of messages 1 with non-zero mac_2. The rest of the protocol is run by the adversary, which has the long-term static keys. In particular, we do not model the encryption of the cookie τ, but send it in the clear, assuming that the adversary carries out the encryption and decryption, which depend only on values it knows.

In this model, we prove that, if a responder under load accepts a handshake message from a sender with address A_i, then this sender passed through a round trip, that is, the responder did indeed previously generate a cookie for the address A_i. This formalised by the following query:

$$\begin{aligned} &\mathsf{event}(\mathsf{accepted\_cookie}(A_i, i_r, \tau, msg_\beta, mac_2)) \\ \Rightarrow\ &\mathsf{event}(\mathsf{generated\_cookie}(A_i, i_r, \tau))\,, \end{aligned}$$

where i_r is an index that uniquely identifies the key R_r used for generating the cookie. This query is proved under the assumption that mac is a pseudo-random function (PRF).

Identity Hiding. When the adversary has a candidate public key S_Y^{pub}, it can determine whether this public key is involved in WireGuard sessions, as already mentioned in the WireGuard specification [18]. In the first message, it can test whether $mac_1 = \mathsf{mac}(\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_Y^{pub}), msg_\alpha)$ and that reveals whether $S_Y^{pub} = S_r^{pub}$. A similar test on message 2 reveals whether $S_Y^{pub} = S_i^{pub}$. When an entity with public key S_m^{pub} sends a cookie reply, the adversary can try to decrypt the encrypted cookie $\tau_{\text{🔒}}$ with the key $\mathsf{hash}(\mathsf{label}_{\mathsf{cookie}} \| S_Y^{pub})$, the nonce *nonce* (obtained from the cookie reply), and the associated data mac_1 (obtained from a previous message). If decryption succeeds, then the adversary knows that $S_Y^{pub} = S_m^{pub}$. In practice, the public keys of VPN endpoints may be easy to obtain: they are often published to subscribers on a web page. In such scenarios, WireGuard does not provide identity hiding.

If we consider the protocol without MACs and cookie reply, that is, basically the Noise protocol IKpsk2, we can obtain stronger identity protection guarantees, however with the additional assumption that the AEAD scheme also preserves the secrecy of the associated data. Indeed, if the AEAD scheme is only IND-CPA and INT-CTXT, then the adversary may obtain the associated data of the first ciphertext $S_i^{pub}{}_{\text{🔒}}$, that is, $\mathsf{hash}(\mathsf{hash}(H_0 \| S_r^{pub}) \| E_i^{pub})$. It can compare this value with $\mathsf{hash}(\mathsf{hash}(H_0 \| S_Y^{pub}) \| E_i^{pub})$ since E_i^{pub} is sent in the first message and H_0 is a constant. Thus, it can determine whether $S_r^{pub} = S_Y^{pub}$.

However, assuming that the AEAD scheme also preserves the secrecy of the associated data, we prove using CryptoVerif that the protocol without MACs and cookie reply satisfies the following identity hiding property: an adversary that has S_{A1}^{pub}, S_{A2}^{pub}, S_{B1}^{pub}, S_{B2}^{pub} cannot distinguish a configuration in which the entity with public key S_{A1}^{pub} initiates sessions with S_{B1}^{pub} from one in which the entity with public key S_{A2}^{pub} initiates sessions with S_{B2}^{pub}. ChaCha20Poly1305 indeed preserves the secrecy of the associated data, because it satisfies the stronger IND$-CPA property, which requires the ciphertext to be indistinguishable from random bits, as shown in [34].

We discuss possible solutions to strengthen the identity hiding for the protocol with MACs in Section 7.

Proof Guidance and Metrics. CryptoVerif needs to be manually guided to perform these proofs. We detail the instructions given to CryptoVerif for proving authentication and message secrecy in variant 1 of our model, with dynamic compromise of the private static keys. The guidance we give for other proofs follows similar ideas.

First, we set some options, in particular to speed up the proof and save memory.

```
set casesInCorresp = false;
```

reduces the number of cases that CryptoVerif considers in proofs of correspondences. This option does not affect the soundness; in complex cases, CryptoVerif might just not be able to prove a correspondence with this option set to false.

```
set mergeBranches = false;
```

prevents CryptoVerif from trying to automatically merge branches of tests when they execute the same code.

```
set forgetOldGames = true;
```

tells CryptoVerif to remove games generated by previous instructions from memory, in order to save memory. (However, that prevents undoing previous proof steps.)

```
set useKnownEqualitiesWithFunctionsInMatching = true;
```

tells CryptoVerif to apply known equalities that start with a function symbol when it tests whether a term matches another term. CryptoVerif would not do that by default because it is costly. However, when using Curve25519, we need to apply equalities of the form $pow\_k(\cdot) = pow\_k(\cdot)$, so we use this setting. We unset it later when it is no longer needed, to speed up the proof.

Next, we distinguish cases. In the initiator A, we add a test to distinguish whether the partner public key `S_X_pub` is equivalent to B's static public key `S_B_pub`. This test is added just after the input that receives `S_X_pub` from the adversary on channel `c_config_initiator`, by the instruction:

```
insert after "in(c_config_initiator\\["
  "if pow_k(S_X_pub) = pow_k(S_B_pub) then";
```

In the instruction above, the test `if pow_k(S_X_pub) = pow_k(S_B_pub) then` is inserted after the line that contains the regular expression `in(c_config_initiator\\[`. (`\\[` denotes the character `[` in regular expressions.) This is an improvement that we implemented in CryptoVerif. Before, CryptoVerif required indicating the program point at which a case distinction should be inserted by an integer number, and this number often varied with very minor changes in the protocol specification. We modified CryptoVerif to allow specifying program points as the beginning of a line that matches a certain regular expression, or as the line that follows a matching line. This is much more stable to small changes in the protocol model.

As a result, the initiator ephemeral is then chosen at two different places, in the `then` branch and in the `else` branch of the test that was just introduced. We rename the variable `E_i_pub_4` containing the initiator public ephemeral to two distinct names (these names are chosen by CryptoVerif and are here `E_i_pub_6` and `E_i_pub_7`) by the instruction:

```
SArename E_i_pub_4;
```

In the responder B, we distinguish whether the partner public key `S_i_pub_rcvd_4` is equivalent to A's static public key `S_A_pub`, by the instruction:

```
insert after "let injbot(G_to_bitstring(S_i_pub_rcvd_4: G_t))"
  "if pow_k(S_i_pub_rcvd_4) = pow_k(S_A_pub) then";
```

The test is inserted after the decryption of the ciphertext $S_{i\,\text{🔒}}^{pub}$. In the responder B, we also distinguish whether the received ephemeral `E_i_pub_rcvd_3` is equivalent to an ephemeral generated by A, `E_i_pub_6[i]` or `E_i_pub_7[i]` for any i. (`E_i_pub_6` and `E_i_pub_7` are arrays containing one public ephemeral for each execution of the A.) This test is inserted after the reception of the ephemeral by the responder B, by the following instruction:

```
insert after "in(c_init2resp_recv\\["
  "find i <= N_init_parties suchthat defined(E_i_pub_7[i]) &&
     pow_k(E_i_pub_rcvd_3) = pow_k(E_i_pub_7[i]) then
 orfind i <= N_init_parties suchthat defined(E_i_pub_6[i]) &&
     pow_k(E_i_pub_rcvd_3) = pow_k(E_i_pub_6[i]) then";
```

The construct `find` i `<=` N `suchthat defined(`$x[i]$`) &&` M `then` P `else` P' looks for an index i such that $x[i]$ is defined and the condition M holds. If it finds one, it runs P with that index; otherwise, it runs P'. It is extended to several branches by using `orfind`. Finally, in the initiator A, we distinguish whether the responder's ephemeral `E_r_pub_rcvd_2` received with the second protocol message is equivalent to an ephemeral generated by B, `E_r_pub[j]`, by the instructions:

```
insert after_nth 2 "in(c_resp2init_recv\\["
  "find j <= N_resp_parties suchthat defined(E_r_pub[j]) &&
     pow_k(E_r_pub_rcvd_2) = pow_k(E_r_pub[j]) then";
insert after_nth 1 "in(c_resp2init_recv\\["
  "find j <= N_resp_parties suchthat defined(E_r_pub[j]) &&
     pow_k(E_r_pub_rcvd_2) = pow_k(E_r_pub[j]) then";
```

We insert two tests because we need to insert one test in each branch of the initial case distinction made in the initiator. These case distinctions allow us to isolate Diffie-Hellman shared secrets that the adversary will be unable to compute because both shares come from honest participants. We simplify the obtained game by

```
simplify;
```

Then, we apply the random oracle assumption for the 3 random oracles chain'_6, chain'_2, chain'_1 (named `rom3_intermediate`, `rom2_intermediate`, and `rom1_intermediate` in the CryptoVerif file). For the arguments of these oracles that are Diffie-Hellman shared secrets in the protocol (and thus are in G_{sub}), we distinguish whether the argument received by the random oracle from the adversary is in G_{sub} before applying the random oracle assumption. (When it is not in G_{sub}, it cannot collide with a call coming from the protocol.) This is done by the following instructions:

```
insert after "in(ch1_rom3"
  "let rom3_input(x1_rom3, Gsub_to_G(x2_rom3), Gsub_to_G(x3_rom3), x4_rom3,
     Gsub_to_G(x5_rom3), Gsub_to_G(x6_rom3), v_psk) = x_rom3 in";
crypto rom(rom3_intermediate);
insert after "in(ch1_rom2"
  "let rom2_input(x1_rom2, Gsub_to_G(x2_rom2), Gsub_to_G(x3_rom2)) =
     x_rom2 in";
crypto rom(rom2_intermediate);
insert after "in(ch1_rom1"
  "let rom1_input(x1_rom1, Gsub_to_G(x2_rom1)) = x_rom1 in";
crypto rom(rom1_intermediate);
```

The first case distinction distinguishes whether the argument `x_rom3` of `rom3_intermediate` is of the form `rom3_input(x1_rom3, Gsub_to_G(x2_rom3), Gsub_to_G(x3_rom3), x4_rom3, Gsub_to_G(x5_rom3), Gsub_to_G(x6_rom3), v_psk)`, that is, a tuple in which the 2nd, 3rd, 5th, and 6th components are in G_{sub}. The next instruction applies the random oracle assumption to `rom3_intermediate`. The other two random oracles are handled similarly.

Next, we apply the gap Diffie-Hellman assumption to the function *exp_div_k*; the associated secret exponents are the secret static and ephemeral keys of the initiator and the responder:

```
crypto gdh(exp_div_k) S_B_priv E_i_priv_8 S_A_priv E_r_priv_4;
```

We modify settings to speed up the rest of the proof by the following instructions:

```
set useKnownEqualitiesWithFunctionsInMatching = false;
set elsefindFactsInSimplify = false;
```

The setting `elsefindFactsInSimplify`, when true, tells CryptoVerif to simplify games using the information obtained from being in an `else` branch of a `find`. It is the default, but it can be costly for large games.

We split the keys generated by chain'_6 into 4 keys by

```
crypto splitter(concat_four_keys) **;
```

The indication `**` means that we apply `splitter(concat_four_keys)` as many times as we can, without performing a full simplification between each application. Avoiding that simplification speeds up the proof a bit. `splitter(concat_four_keys)` means that a random bitstring of length 4 times the length of a key is indistinguishable from the concatenation of 4 random keys.

By default, when a cryptographic transformation fails, CryptoVerif tries to determine syntactic transformations that might make it succeed, applies those transformations, and retries the cryptographic transformation. For speed, we disable this behaviour by the following instruction:

```
set noAdviceCrypto = true;
```

We apply ciphertext integrity of the AEAD scheme:

```
crypto int_ctxt(enc) *;
```

The indication `*` means that we apply `int_ctxt(enc)` as many times as we can. Then we try to prove security properties:

```
success;
```

CryptoVerif shows the impossibility of nonce reuse in the AEAD scheme and the absence of identity mis-binding attacks. We simplify the game

```
simplify;
```

For keys that the adversary may have after compromising the static keys, we apply a variant of the ciphertext integrity transformation that allows corruption, as follows:

```
crypto int_ctxt_corrupt(enc) k_51;
```

The key `k_51` is generated by the initiator when the partner static key is equivalent to B's static public key but the received ephemeral is not equivalent to an ephemeral generated by B, and B's static key is not compromised. In this case, the adversary cannot produce a valid ciphertext in protocol message 2 (empty plaintext), thus the decryption will fail on the initiator's side and the protocol will not continue.

Then we try to prove security properties:

```
success;
```

CryptoVerif proves that the initiator can authenticate the second protocol message as well as transport data messages sent by the responder. We simplify the game

```
simplify;
```

and again apply the variant of the ciphertext integrity transformation that allows corruption:

```
crypto int_ctxt_corrupt(enc) "T_i_send_[0-9]*";
```

We apply this transformation to all keys of the form `T_i_send_`n for integers n. We want to apply this transformation to keys generated by the responder, in case the partner public key is equivalent to A's static public key, but the received ephemeral is not equivalent to an ephemeral generated by A, and A's static key is not compromised. In this case, the adversary cannot produce a valid ciphertext for a transport data message, thus the decryption will fail on the responder's side and the protocol will not continue. The keys in question are many variables of the form `T_i_send_`n; we apply the transformation to all variables of this form as it is easier and the proof still works. Then we try to prove security properties:

```
success;
```

CryptoVerif shows that the responder can authenticate transport data messages sent by the initiator. We again simplify the game

```
simplify;
```

That removes all events, which are no longer useful since all correspondence properties are proved. Then we apply the IND-CPA property of the AEAD scheme as many times as we can:

```
crypto ind_cpa(enc) **;
```

and finally prove message secrecy:

```
success
```

In total, we give 36 instructions to CryptoVerif to perform this proof (not counting the instruction to display the current game), and CryptoVerif generates a sequence of 168 games. This proof takes 14 min, the proof of key secrecy with dynamic compromise of private static keys takes 16 min, and the one for identity hiding 18 min on one core of an Intel Xeon 3.6 GHz; these are our longest proofs.

7 Discussion

WireGuard is a promising new VPN protocol that aims to replace IPsec and OpenVPN, and is being considered for adoption within the Linux kernel. We presented a mechanised cryptographic proof for a detailed model of WireGuard using the CryptoVerif prover. Our model accounts for the full Noise IKpsk2 secure channel protocol as well as WireGuard's extensions for stealthy operation and DoS resistance. We consider an arbitrary number of parallel sessions, with an arbitrary number of transport data messages. Furthermore, we base our proof on a precise model of the Curve25519 group.

We proved correctness, message and key secrecy, forward secrecy, mutual authentication, session uniqueness, channel binding, and resistance against replay, key compromise impersonation,

and denial of service attacks. In some cases, our analysis pointed out potential improvements in the protocol (which we did not prove secure using CryptoVerif):

Adding Public Keys to the Chaining Key Derivation. When analysing WireGuard for Identity Mis-Binding attacks, our analysis uncovered a corner case. Suppose all the Diffie-Hellman keys in a session between two hosts A and B were compromised, but the pre-shared key between them is still secret. Then the adversary can set up a man-in-the-middle attack where A thinks it is connected to B', B thinks it is connected to A', but in fact they are both connected to each other, in the sense that the two connections have the same traffic keys, even though they have different static keys.

In particular, once it has set up the session, the adversary can step away and let A and B directly communicate with each other, while retaining the ability to read and modify messages at will. Interestingly, this vulnerability only appears in our precise model of Curve25519; it cannot be detected under a classic Diffie-Hellman assumption.

Although this attack scenario may be quite unrealistic, it points to a theoretical weakness in the protocol that is easy to prevent with a simple modification. Noise IKpsk2 already adds ephemeral public keys to the chaining key derivation; we recommend that the static public keys be added as well. Alternatively, adding the full transcript hash to the traffic key derivation would also prevent this corner case.

Separately, it is also worth noting that adding public keys to the key derivation significantly helps with the cryptographic proof. For example, consider the Noise IK protocol, which is similar to IKpsk2 except that it does not use PSKs. IK does not mix the ephemeral keys into the chaining key, and it turns out that it is much harder for CryptoVerif to verify than IKpsk2, since we now have to reason about mis-matched ephemeral keys. In particular, even if we use a public PSK key of all-zeroes, the IKpsk2 protocol is easier to prove secure than IK. In fact, our recommendation is to add further contextual information to the key derivation. It would not only prevent theoretical attacks, but also make proofs easier.

Balancing Stealth and Identity Hiding. Our analysis also points out that the use of static public keys in mac_1 and mac_2 in WireGuard negatively affects the identity hiding guarantees provided by IKpsk2. This is a conscious trade-off that WireGuard makes to achieve stealthy operation [18]. However, in deployment scenarios where identity hiding is more important than stealth, we recommend that the protocol use a constant (say all-zeroes) instead of the static public keys to compute the MACs and cookies.

While it is difficult to preserve stealth while hiding the responder's identity, a modification to the protocol can still hide the initiator's identity. We recommend that the initiator should send a MAC key (along with the timestamp) in the first handshake message, and the responder should use this MAC key to compute mac_1 in the second handshake message. The initiator can verify this MAC to get DoS protection, but its static public key is kept hidden from a network adversary. Essentially, the MAC key acts as an in-session cookie.

Related Work. The use of formal verification tools to analyse real-world cryptographic protocols is now a well-established research area with hundreds of case studies (see e.g. [12]). CryptoVerif itself has been used to analyse modern protocols like Signal [24] and TLS 1.3 [7]. We conclude this paper by comparing our results with closely related work; Table 1 provides a condensed, high-level overview.

WireGuard itself has been formally analysed before. Donenfeld et al. [19] symbolically analyse the IKpsk2 key exchange protocol used by WireGuard for a number of security goals, including identity mis-binding and identity hiding. However, they do not model the MACs or the cookie mechanism, and hence they do not prove DoS resistance. Interestingly, their analysis concludes the absence of identity mis-binding attacks even if all keys are compromised, because their model

Table 1: Security models (upper part) and properties analysed (lower part) in different works on WireGuard or Noise IKpsk2.

	Noise Explorer [25]	Suter-Dörig [37]	Girol [22]	Donenfeld, Milner [19]	Dowling, Paterson [20]	this work
	Noise IKpsk2			WireGuard		
tool set	PV	T	T	T	m	CV
computational model	x	x	x	x	✓	✓
Curve25519 with equivalent keys	x	x	x	x	x	✓
compromise static keys	✓	✓	✓	✓	✓	✓
compromise ephemeral keys	x	x	✓	✓	✓	✓
dishonest ephemeral keys	x	x	✓	x	x	x
compromise pre-shared key	✓	✓	✓	✓	✓	✓
compromise all keys	x	x	✓	✓	x	✓
both roles per static key	x	✓	✓	✓	✓	✓
mutual authentication	✓	✓	✓	✓	✓	✓
key compromise impersonation	✓	✓	✓	✓	✓	✓
1st message replay	—	—	—	x	x	✓
transport data replay	x	✓	✓	x	x	✓
session uniqueness	x	✓	x	✓	✓	✓
channel binding	x	✓	x	x	x	✓
DoS resistance	—	—	—	x	x	✓
forward key secrecy	✓	✓	✓	✓	✓	✓
forward message secrecy	✓	✓	✓	x	x	✓
identity hiding	x	x	✓	✓[2]	x	✓
identity mis-binding	x	x	x	✓[1]	x	✓

Definitions differ between models.
T = Tamarin, PV = ProVerif, CV = CryptoVerif, m = manual.
✓= included, x = not included, — = not applicable.
1) the identity mis-binding issue we found was *not* found.
2) Weaker identity hiding property using a surrogate term.

does not include equivalent public keys. We disprove this property by considering a precise model of Curve25519.

Dowling et al. [20] present a manual cryptographic analysis of WireGuard. In particular, they prove key indistinguishability for the WireGuard handshake based on the PRF-ODH assumption in an extension of the eCK-PFS key exchange model. (Because of this difference in the used assumption, our mechanization cannot be used directly to find issues in proof steps; it is a different proof.) Key indistinguishability no longer holds once the key is used, so they prove security for a slightly modified variant of the IKpsk2 protocol that includes a key confirmation message independent of the session keys. In contrast, our proof requires no changes to the protocol, since we use an ACCE-style model. Furthermore, [20] focuses only on the key exchange, and does not consider other properties like identity hiding or DoS resistance. Their analysis also does not find the identity mis-binding issue since they do not consider a scenario where all Diffie-Hellman keys are compromised.

Finally, the Noise Explorer tool [25] has been used to perform a comprehensive symbolic analysis of numerous Noise protocols using the ProVerif analyser. Noise Explorer can be used to find violations of secrecy and authentication properties for any protocol expressed in the language defined by Noise, using per-message authentication and confidentiality grades. It includes a symbolic analysis of Noise IKpsk2. A similar work has been done in Tamarin [22, 37].

Acknowledgements

We thank Jason A. Donenfeld, the author of WireGuard, and Nadim Kobeissi for their helpful feedback on our work. This research was partly funded by the European Union's Horizon 2020 NEXTLEAP Project (grant agreement nº 688722), ERC CIRCUS (grant agreement nº 683032), ANR AnaStaSec (decision number ANR-14-CE28-0014-01), and ANR TECAP (decision number ANR-17-CE39-0004-03).

References

[1] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In *Applied Cryptography and Network Security*, volume 7954 of *LNCS*, pages 119–135. Springer, 2013.

[2] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT'00*, volume 1976 of *LNCS*, pages 531–545. Springer, Dec. 2000.

[3] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM CCS'93*, pages 62–73. ACM Press, 1993.

[4] D. J. Bernstein. The Poly1305-AES message-authentication code. In *FSE 2005*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005.

[5] D. J. Bernstein. Extending the Salsa20 nonce, 2011. `https://cr.yp.to/snuffle/xsalsa-20110204.pdf`.

[6] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE S&P (Oakland)*, pages 535–552, 2015.

[7] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE S&P (Oakland)*, pages 483–502, 2017.

[8] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland)*, pages 98–113, 2014.

[9] K. Bhargavan and G. Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM CCS'16*, pages 456–467, 2016.

[10] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *IEEE CSF'07*, pages 97–111, July 2007. Extended version available at `http://eprint.iacr.org/2007/128`.

[11] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008.

[12] B. Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust, POST'12*, volume 7215 of *LNCS*, pages 3–29. Springer, 2012.

[13] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, Oct. 2016.

[14] J. Brendel, M. Fischlin, F. Günther, and C. Janson. PRF-ODH: Relations, instantiations, and impossibility results. In *CRYPTO 2017*, volume 10403 of *LNCS*, pages 651–681. Springer, Aug. 2017.

[15] L. Chen and Q. Tang. Bilateral unknown key-share attacks in key agreement protocols. *Journal of Universal Computer Science*, 14(3):416–440, Feb. 2008.

[16] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, 2005.

[17] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro. To hash or not to hash again? (in)differentiability results for H^2 and HMAC. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 348–366. Springer, 2012. Full version at `https://eprint.iacr.org/2013/382`.

[18] J. A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *Network and Distributed System Security Symposium, NDSS*, 2017. We use the up-to-date whitepaper version for our analysis, which differs in how the MACs are defined: `https://www.wireguard.com/papers/wireguard.pdf`, Nov. 2nd, 2017, draft revision ceb3a49.

[19] J. A. Donenfeld and K. Milner. Formal verification of the WireGuard protocol, 2018. `https://www.wireguard.com/papers/wireguard-formal-verification.pdf`.

[20] B. Dowling and K. G. Paterson. A cryptographic analysis of the WireGuard protocol. In *Applied Cryptography and Network Security, ACNS 2018*, volume 10892 of *LNCS*, pages 3–21. Springer, 2018.

[21] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, 2018. IETF RFC 8446.

[22] G. Girol. Formalizing and verifying the security protocols from the Noise framework. Master's thesis, ETH Zürich, Mar. 2019. Available at `https://doi.org/10.3929/ethz-b-000332859`.

[23] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, 2012.

[24] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE EuroS&P'17*, pages 435–450, Apr. 2017.

[25] N. Kobeissi, G. Nicholas, and K. Bhargavan. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In *IEEE EuroS&P 2019*, June 2019. To appear. The tool is available at `https://noiseexplorer.com/`.

[26] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF), 2010. IETF RFC 5869.

[27] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security, Jan. 2016. IETF RFC 7748.

[28] A. Luykx, B. Mennink, and S. Neves. Security analysis of BLAKE2's modes of operation. *IACR Transactions on Symmetric Cryptology*, 2016(1):158–176, Dec. 2016.

[29] M. Marlinspike and T. Perrin. The X3DH key agreement protocol, Nov. 2016. Available at `https://signal.org/docs/specifications/x3dh/`.

[30] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification, CAV'13*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.

[31] Nir, Yoav and Langley, Adam. ChaCha20 and Poly1305 for IETF Protocols, June 2018. IETF RFC 8439.

[32] T. Okamoto and D. Pointcheval. The gap-problems: a new class of problems for the security of cryptographic schemes. In *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Feb. 2001.

[33] T. Perrin. The Noise protocol framework, July 2018. `https://noiseprotocol.org/noise.html`.

[34] G. Procter. A security analysis of the composition of ChaCha20 and Poly1305. Cryptology ePrint Archive, Report 2014/613, 2014. `https://eprint.iacr.org/2014/613`.

[35] S. Kent and K. Yao. Security Architecture for the Internet Protocol, 2005. IETF RFC 4301.

[36] M.-J. Saarinen and J.-P. Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC), 2015. IETF RFC 7693.

[37] A. Suter-Dörig. Formalizing and verifying the security protocols from the Noise framework. Bachelor's thesis, ETH Zürich, Nov. 2018. Available at `https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/noise_suter-doerig.pdf`.

A Indifferentiability Results

A.1 Basic Lemmas

Proof of Lemma 1. Consider

- the game G_0 in which H is a random oracle, and $H_i(x) = H(x)$ for each $x \in D_i$ and $i \leq n$, and
- the game G_1 in which $H_1, \ldots H_n$ are independent random oracles defined on $D_1, \ldots, D_n$ respectively, and $H(x) = H_i(x)$ if $x \in D_i$ for some $i \leq n$, and $H(x) = H_0(x)$ otherwise, where H_0 is a random oracle of domain $D \setminus (D_1 \cup \cdots \cup D_n)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. □

Proof of Lemma 2. Consider

- the game G_0 in which H_1 and H_2 are independent random oracles, and $H'(x) = H_1(x) \| H_2(x)$, and
- the game G_1 in which H' is a random oracle that returns bitstrings of length $l_1 + l_2$, $H_1(x)$ is the l_1 first bits of $H'(x)$ and $H_2(x)$ is the l_2 last bits of $H'(x)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. □

Proof of Lemma 3. Consider

- the game G_0 in which H is a random oracle, $H'_1(x)$ is the first l_1 bits of $H(x)$, and $H'_2(x)$ is the last $l - l_1$ bits of $H(x)$, and
- the game G_1 in which H'_1 and H'_2 are independent random oracles that return bitstrings of length l_1 and $l - l_1$ respectively, and $H(x) = H'_1(x) \| H'_2(x)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. (It is the same indistinguishability result as in Lemma 2, swapping G_0 and G_1.) □

Proof of Lemma 4. This is a consequence of Lemma 3, by not giving access to oracle H'_2 to the distinguisher (so $q_{H'_2} = 0$). H'_2 is then included in the simulator. We assume that random oracles answer in constant time. □

Proof of Lemma 6. Consider

- the game G_0 in which H_1 and H_2 are independent random oracles, $H'_1(x) = H_1(x)$, and $H'_2(x) = H_2(H_1(x), x)$, and
- the game G_1 in which H'_1 and H'_2 are independent random oracles; $H_1(x) = H'_1(x)$; $H_2(y, z)$ returns $H'_2(z)$ if $y = H'_1(z)$ and $H_3(y, z)$ otherwise, where H_3 is a random oracle (independent of H'_1 and H'_2).

CryptoVerif shows that these two games are indistinguishable, up to probability ϵ; the oracles H_1 and H'_1 are considered as a single oracle, which receives $q_{H_1} + q_{H'_1}$ queries in total). □

A.2 Indifferentiability of HKDF

Much like for HMAC in [17] and as mentioned in [24], hkdf_n is not indifferentiable from a random oracle in general. Intuitively, the problem comes from a confusion between the first and the second (or third) call to hmac, which makes it possible to generate *prk* by calling hkdf_2 rather than hmac. In more detail, let

$$\begin{aligned}
&prk\|_ = \mathsf{hkdf}_2(s, k, info)\\
&salt = \mathsf{hmac}(s, k)\\
&x = \mathsf{hmac}(prk, info'\|i_0)\\
&x'\|_ = \mathsf{hkdf}_2(salt, info\|i_0, info')
\end{aligned}$$

where the notation $x_1\|x_2 = \mathsf{hkdf}_2(s, k, info)$ denotes that x_1 consists of the first 256 bits of $\mathsf{hkdf}_2(s, k, info)$ and x_2 its last 256 bits.

When hkdf_2 is defined from hmac as above, we have $prk = \mathsf{hmac}(prk', info\|i_0)$ where $prk' = \mathsf{hmac}(s, k) = salt$, so $prk = \mathsf{hmac}(salt, info\|i_0)$. Hence, $x' = \mathsf{hmac}(prk, info'\|i_0) = x$. However, when hkdf_2 is a random oracle and hmac is defined from hkdf_2, the simulator that computes hmac sees what seems to be two unrelated calls to hmac. (It is unable to see that *prk* is in fact related to the previous call $salt = \mathsf{hmac}(s, k)$: we have $prk\|\_ = \mathsf{hkdf}_2(s, k, info)$ but the simulator does not know which value of *info* it should use.) Therefore, the simulator can only return fresh random values for *salt* and x, and $x \neq x'$ in general.

Proof of Lemma 7. In this proof, we write $S[H_1, \ldots, H_n]$ instead of $S^{H_1,\ldots,H_n}$ for a system S with oracle access to $H_1, \ldots, H_n$, because we need to write systems in which the oracles are themselves systems that access other oracles. Consider the game G_0 in which hmac is a random oracle and hkdf_n is defined as above.

By hypothesis, the different calls to hmac in the definition of hkdf_n use disjoint domains (the last byte differs among the last n calls to hmac), so by Lemma 1, there exists a simulator S_1 for hmac such that G_0 is perfectly indistinguishable from G_1 in which $\mathsf{hmac} = S_1[H_0, \ldots, H_n]$ and hkdf_n is defined by

$$\begin{aligned}
&\mathsf{hkdf}_n^1(salt, key, info) = k_1\| \ldots \|k_n \text{ where}\\
&\quad prk = H_0(salt, key)\\
&\quad k_1 = H_1(prk, info)\\
&\quad k_{i+1} = H_{i+1}(prk, k_i\|info) \text{ for } 1 \leq i < n
\end{aligned}$$

where $H_0, \ldots, H_n$ are independent random oracles and the simulator S_1 calls H_i at most q_{H_i} times, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

Slightly reorganising the arguments of H_i, there exists a simulator S_2 for hmac such that G_1 is perfectly indistinguishable from G_2 in which $\mathsf{hmac} = S_2[H_0, \ldots, H_n]$ and hkdf_n is defined by

$$\begin{aligned}
&\mathsf{hkdf}_n^2(salt, key, info) = k_1\| \ldots \|k_n \text{ where}\\
&\quad prk = H_0(salt, key)\\
&\quad k_1 = H_1(prk, info)\\
&\quad k_{i+1} = H_{i+1}(k_i, prk, info) \text{ for } 1 \leq i < n
\end{aligned}$$

where $H_0, \ldots, H_n$ are independent random oracles and the simulator S_2 calls H_i at most q_{H_i} times, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 6, there exists a simulator $S_{3,2}$ for H_2 (H_1 is simulated by $H'_1 = H_1$ itself) such that $G_2 = G_{3,1}$ is indistinguishable up to probability ϵ_2 from $G_{3,2}$ in which $\mathsf{hmac} = S_2[H_0, H'_1, S_{3,2}[H'_1, H'_2], H_3, \ldots, H_n]$ and hkdf_n is defined by

$$\begin{aligned}&\mathsf{hkdf}_n^{3,2}(salt, key, info) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad prk = H_0(salt, key)\\ &\quad k_1 = H'_1(prk, info)\\ &\quad k_2 = H'_2(prk, info)\\ &\quad k_{i+1} = H_{i+1}(k_i, prk, info) \text{ for } 2 \leq i < n\end{aligned}$$

where $H_0, H'_1, H'_2, H_3, \ldots, H_n$ are independent random oracles; the simulator for hmac calls H_0 at most q_{H_0} times, H'_1 at most $q_{H_1} + q_{H_2}$ times, H'_2 at most q_{H_2} times, H_i at most q_{H_i} times for $i \geq 3$, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_2})$; and $\epsilon_2 = q_{H_2}(2q_{H_1} + 3q_{\mathsf{hkdf}_n} + 1)/|\mathcal{M}|$.

Repeating the same reasoning inductively, there exists a simulator $S_{3,j}$ for H_j such that $G_{3,j-1}$ is indistinguishable up to probability ϵ_j from $G_{3,j}$ in which $\mathsf{hmac} = S_2[H_0, H'_1, S_{3,2}[H'_1, H'_2], \ldots, S_{3,j}[H'_{j-1}, H'_j], H_{j+1}, \ldots, H_n]$ and hkdf_n is defined by

$$\begin{aligned}&\mathsf{hkdf}_n^{3,j}(salt, key, info) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad prk = H_0(salt, key)\\ &\quad k_i = H'_i(prk, info) \text{ for } 1 \leq i \leq j\\ &\quad k_{i+1} = H_{i+1}(k_i, prk, info) \text{ for } j \leq i < n\end{aligned}$$

where $H_0, H'_1, \ldots, H'_j, H_{j+1}, \ldots, H_n$ are independent random oracles; the simulator for hmac calls H_0 at most q_{H_0} times, H'_i at most $q_{H_i} + q_{H_{i+1}}$ times for $1 \leq i < j$, H'_j at most q_{H_j} times, H_i at most q_{H_i} times for $i > j$, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_2} + \cdots + q_{H_j})$; and $\epsilon_j = q_{H_j}(2q_{H_{j-1}} + 3q_{\mathsf{hkdf}_n} + 1)/|\mathcal{M}|$.

For $j = n$, we obtain a game $G_{3,n}$ in which $\mathsf{hmac} = S_2[H_0, H'_1, S_{3,2}[H'_1, H'_2], \ldots, S_{3,n}[H'_{n-1}, H'_n]]$ and hkdf_n is defined by

$$\begin{aligned}&\mathsf{hkdf}_n^{3,n}(salt, key, info) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad prk = H_0(salt, key)\\ &\quad k_i = H'_i(prk, info) \text{ for } 1 \leq i \leq n\end{aligned}$$

where $H_0, H'_1, \ldots, H'_n$ are independent random oracles; the simulator for hmac calls H_0 at most q_{H_0} times, H'_i at most $q_{H_i} + q_{H_{i+1}}$ times for $1 \leq i < n$, H'_n at most q_{H_n} times, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_2} + \cdots + q_{H_n}) = \mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 2, there exist simulators $S_{4,j}$ ($1 \leq j \leq n$) for H'_j such that $G_{3,n}$ is perfectly indistinguishable from G_4 in which $\mathsf{hmac} = S_2[H_0, S_{4,1}[H], S_{3,2}[S_{4,1}[H], S_{4,2}[H]], \ldots, S_{3,n}[S_{4,n-1}[H], S_{4,n}[H]]]$ and hkdf_n is defined by

$$\begin{aligned}&\mathsf{hkdf}_n^{4}(salt, key, info) = H(prk, info) \text{ where}\\ &\quad prk = H_0(salt, key)\end{aligned}$$

where H_0 and H are independent random oracles; the simulator for hmac calls H_0 at most q_{H_0} times, H at most $q_H = \sum_{i=1}^{n-1}(q_{H_i} + q_{H_{i+1}}) + q_{H_n} \leq 2q_{\mathsf{hmac}}$ times, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 5, there exist simulators $S_{5,1}$ for H_0 and $S_{5,2}$ for H such that G_4 is indistinguishable up to probability ϵ' from G_5 in which $\mathsf{hmac} = S_2[H_0, S_{4,1}[H], S_{3,2}[S_{4,1}[H], S_{4,2}[H]]$,

$\ldots, S_{3,n}[S_{4,n-1}[H], S_{4,n}[H]]]$ with $H_0 = S_{5,1}[\mathsf{hkdf}_n^5]$ and $H = S_{5,2}[\mathsf{hkdf}_n^5]$ and $\mathsf{hkdf}_n = \mathsf{hkdf}_n^5$ is a random oracle, where the simulator for hmac calls hkdf_n at most $q_H \leq 2q_{\mathsf{hmac}}$ times[2], runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_0} q_H) = \mathcal{O}(q_{\mathsf{hmac}}^2)$, and

$$\begin{aligned}\epsilon' &= (2q_H q_{H_0} + q_{H_0}^2 + q_H q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2)/|\mathcal{M}| \\ &= (2q_{\mathsf{hmac}} q_{H_0} + q_{H_0}^2 + q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2)/|\mathcal{M}|\end{aligned}$$

The probability of distinguishing G_0 from G_5 is then at most

$$\begin{aligned}\epsilon &= \epsilon' + \sum_{j=2}^{n} \epsilon_j \\ &= \frac{2q_{\mathsf{hmac}} q_{H_0} + q_{H_0}^2 + q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2}{|\mathcal{M}|} + \sum_{j=2}^{n} \frac{q_{H_j}(2q_{H_{j-1}} + 3q_{\mathsf{hkdf}_n} + 1)}{|\mathcal{M}|} \\ &= \frac{1}{|\mathcal{M}|}(2q_{\mathsf{hmac}} q_{H_0} + q_{H_0}^2 + q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2) + (\sum_{j=2}^{n} q_{H_j})(2q_{\mathsf{hmac}} + 3q_{\mathsf{hkdf}_n} + 1) \\ &= \frac{1}{|\mathcal{M}|}((q_{H_0} + \sum_{j=2}^{n} q_{H_j}) \times 2q_{\mathsf{hmac}} + (\sum_{j=2}^{n} q_{H_j})(3q_{\mathsf{hkdf}_n} + 1) + q_{H_0}^2 + q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2) \\ &= \frac{1}{|\mathcal{M}|}(2q_{\mathsf{hmac}}^2 + 3q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hmac}} + q_{\mathsf{hmac}}^2 + q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2) \\ &= \frac{3q_{\mathsf{hmac}}^2 + 4q_{\mathsf{hmac}} q_{\mathsf{hkdf}_n} + q_{\mathsf{hmac}} + q_{\mathsf{hkdf}_n}^2}{|\mathcal{M}|} \\ &\leq \frac{(2q_{\mathsf{hmac}} + q_{\mathsf{hkdf}_n})^2}{|\mathcal{M}|}\end{aligned}$$

□

A.3 Chain of Random Oracle Calls

Proof of Lemma 8. We consider the following two games G_0 and G_1.

- The game G_0 in which H is a random oracle and the functions chain_n with $0 \leq n \leq m$ are defined from H by (11) and (12).
- The game G_1 in which the functions chain_n with $0 \leq n \leq m$ are independent random oracles and H is defined from them in Figure 2. In this figure, L is a list of triples $((C, v), (C_{j+1}, r_j), j)$ such that $C_{j+1} \| r_j$ is the result of a previous call to $\mathsf{H}(C, v)$ and j indicates the index of this H call in a chain of calls to H. If a call to H was not coming from a chain of calls, the index $j = -2$ is used.

  We shortly comment this simulator's four cases informally. In case 1, it returns a previous result because the same call has already been made before. In case 2, the call to H uses const as first argument and is thus the first call in a potential chain of calls to H. Therefore, the simulator uses chain_0 to get the result and writes it to the list L with index 0. In case 3, the simulator finds in L a previous call to H that returned the current call's C value as result. This means, with respect to the hypothesis we present just after this paragraph, that

[2]The factor 2 here is spurious: the same call to hkdf_n provides the results of H_j and H_{j+1} that are needed to simulate one call to hmac. So in fact, the simulator for hmac calls hkdf_n at most q_{hmac} times.

$\mathsf{H}(C, v) =$

1) **if** $((C, v), (C_{j+1}, r_j), j) \in L$ for some C_{j+1}, r_j, j **then**
   return $C_{j+1} \| r_j$
2) **elseif** $C = \mathsf{const}$ **then**
   $C_1 \leftarrow\!\!\$\ \{0,1\}^{l'}$
   $r_0 \leftarrow \mathsf{chain}_0(v)$
   add $((C, v), (C_1, r_0), 0)$ to L
   return $C_1 \| r_0$
3) **elseif** $((C_j, v_j), (C, r_j), j) \in L$ for some C_j, v_j, r_j and j with $0 \leq j < m$ **then**
   for $k = j - 1$ to 0 **do**
     find $((C_k, v_k), (C_{k+1}, r_k), k) \in L$ for some C_k, v_k, r_k
     set C_k and v_k to the found values
   endfor
   if $j + 1 = m$ **then**
     $C_{j+2} \| r_{j+1} = \mathsf{chain}_m(v_0, \ldots, v_j, v)$
   else
     $C_{j+2} \leftarrow\!\!\$\ \{0,1\}^{l'}$
     $r_{j+1} \leftarrow \mathsf{chain}_{j+1}(v_0, \ldots, v_j, v)$
   endif
   add $((C, v), (C_{j+2}, r_{j+1}), j + 1)$ to L
   return $C_{j+2} \| r_{j+1}$
4) **else**
   $C_{-1} \leftarrow\!\!\$\ \{0,1\}^{l'}$
   $r_{-2} \leftarrow\!\!\$\ \{0,1\}^{l-l'}$
   add $((C, v), (C_{-1}, r_{-2}), -2)$ to L
   return $C_{-1} \| r_{-2}$
  endif

Figure 2: Simulator for H

the current call belongs to a chain of previous calls that was started with a call responded to by case 2. The simulator collects the arguments v_k of those previous calls to be able to call the appropriate chain_n oracle. If the simulator reaches case 4, then the call did neither start a new chain nor belong to a previously started chain. Thus, it chooses fresh random values as a result and adds them with an index to L that makes sure that it will never be considered as part of a chain.

We name *direct* oracle calls to chain_n or H calls that are done directly by the distinguisher, and *indirect* oracle calls the calls to H done from inside chain_n in G_0 and the calls to chain_n done from inside H in G_1. Note for clarification that in G_0 there are no indirect calls to chain_n and in G_1 there are no indirect calls to H.

We show that the two games G_0 and G_1 are indistinguishable as long as the following hypotheses hold: In game G_0,

H1. Consider $C_{j+1}\|r_j = \mathsf{H}(C_j, v_j), j \leq n < m$ that gets called from inside a direct call to $\mathsf{chain}_n(v_1, \ldots, v_j, \ldots, v_n)$. If the distinguisher calls $\mathsf{H}(C_{j+1}, v_{j+1})$ before or after the call to chain_n, then $\mathsf{H}(C_j, v_j)$ has been called directly by the distinguisher before $\mathsf{H}(C_{j+1}, v_{j+1})$.

Stated informally, the distinguisher can only know C_{j+1} if it was received as result from H.

and in game G_1,

H2. no fresh C_j is equal to the first argument of a previous call to H (including the call that generates C_j).

Stated informally, the distinguisher cannot prepend to a chain of H calls.

H3. there are no collisions between the fresh C_j.

We have the following invariants:

P1. Given C, v, there is at most one pair $((C_{j+1}, r_j), j)$ such that $((C, v), (C_{j+1}, r_j), j) \in L$.

Indeed, when L contains such an element, calls to $\mathsf{H}(C, v)$ immediately return $C_{j+1}\|r_j$ in case 1, and never add another element $((C, v), (C_{j+1}, r_j), j)$ to L.

P2. Given $((C_j, v_j), (C, r_j), j) \in L$, for some C_j, v_j, r_j and j with $0 \leq j < m$ then there is, for each $k = j - 1$ to 0, a single matching element $((C_k, v_k), (C_{k+1}, r_k), k)$ in L.

More informally, at no time there are entries in L that belong to chains that are incomplete in the front, i. e. that did not start by a call to H with $C = \mathsf{const}$. And yet differently stated, the simulator can, in case 3, always reconstruct the whole chain of H calls and collect the arguments v_j.

We first show the existence of $((C_k, v_k), (C_{k+1}, r_k), k)$ in L for each $k = j - 1$ to 0. If there is an element in L with $j > 0$, then case 3 was executed before for a matching H call and its result was added to the list with $j' = j - 1$. This is because of H2 and the fact that only in case 3 elements with $j > 0$ are added to the list. This argument can be repeated recursively until reaching $j = 1$. For $j = 0$, the matching element that started the chain was added by case 2, once again because of H2 and because only in case 2 elements with $j = 0$ are added to the list.

Moreover, the uniqueness of $((C_k, v_k), (C_{k+1}, r_k), k)$ comes from H3: when an element $((C_k, v_k), (C_{k+1}, r_k), k)$ is added to L, C_{k+1} is always fresh C_j, so by H3, there is a single element in L with a given C_{k+1}.

We now treat all possible traces of calls in both games.

Case 1 Suppose the distinguisher makes a direct oracle call to H or chain_n with the same arguments as a previous direct call to the same oracle. Both G_0 and G_1 return the same result as in the previous call.

Case 2 Suppose the distinguisher makes a direct call to chain_n that has not been done before as a direct call.

Case 2. a) In G_0, the last $\mathsf{H}(C, v_n)$ in the chain that simulates $\mathsf{chain}_n(v_0, \ldots, v_n)$ *has* already been called directly. Then by H1 the distinguisher did all H calls in the chain that simulates $\mathsf{chain}_n(v_0, \ldots, v_n)$ directly.

The result in G_0 is

$$\_ \| \mathsf{chain}_n(v_0, \ldots, v_n) = \mathsf{H}(C, v_n)$$

which is the last part of the result of the previous call to H, or in the case of $n = m$

$$\mathsf{chain}_m(v_0, \ldots, v_m) = \mathsf{H}(C, v_m)\,.$$

In G_1, because the whole chain of H calls was made in the right order, $\mathsf{chain}_n(v_0, \ldots, v_n)$ has already been invoked indirectly by the call to $\mathsf{H}(C, v_n)$. Thus, this current call to chain_n returns a previously fixed value, fulfilling the following equation:

$$\mathsf{H}(C, v_n) = C_{j+1} \| \mathsf{chain}_n(v_0, \ldots, v_n)$$

or in the case of $n = m$

$$\mathsf{H}(C, v_m) = \mathsf{chain}_m(v_0, \ldots, v_m)\,.$$

This is the same result as in G_0.

Case 2. b) In G_0, the last $\mathsf{H}(C, v_n)$ in the chain that simulates $\mathsf{chain}_n(v_0, \ldots, v_n)$ *has not* already been called directly. Like in the previous case, the result is

$$\_ \| \mathsf{chain}_n(v_0, \ldots, v_n) = \mathsf{H}(C, v_n)$$

or in the case of $n = m$

$$\mathsf{chain}_m(v_0, \ldots, v_m) = \mathsf{H}(C, v_m)\,,$$

but as $\mathsf{H}(C, v_n)$ and $\mathsf{chain}_n(v_1, \ldots, v_n)$ have not been called before directly, the result is independent of previously returned values and thus looks like a fresh random value to the distinguisher.

In G_1, $\mathsf{chain}_n(v_0, \ldots, v_n)$ has not been invoked before and thus returns a fresh random value.

Case 3 Suppose the distinguisher makes a direct call to H that has not been done before as a direct call.

Case 3. a) In G_0, this call to $\mathsf{H}(C, v_i)$ *has* already been done from inside a $\mathsf{chain}_n(v_0, \ldots, v_n)$ call. Hence all other H calls belonging to this chain have also been done from inside said chain_n call, in particular the call $\mathsf{H}(C_{i-1}, v_{i-1})$ directly before the current call (except for $C = \mathsf{const}$, thus if the current call is the beginning of a chain). H1 implies that the distinguisher has then made a direct call to $\mathsf{H}(C_{i-1}, v_{i-1})$ before the current H call. By recursively applying H1, the distinguisher has then directly made all H calls in the chain up to the current one, in the right order.

Case 3. a) i) In G_0, the current direct call to $\mathsf{H}(C, v_n)$ has already been done as *the last one* of the chain of calls indirectly invoked from inside a $\mathsf{chain}_n(v_0, \ldots, v_n)$ call.

In G_0, the result fulfils the following equation:

$$C_{n+1} \| \mathsf{chain}_n(v_0, \ldots, v_n) = \mathsf{H}(C, v_n)\,,$$

and in the case of $n = m$:

$$\mathsf{chain}_m(v_0, \ldots, v_m) = \mathsf{H}(C, v_m)\,.$$

This is similar to case 2. a) just that the order of the calls is inverted and the following small difference: The parts of H's result coming from chain are already known by the distinguisher, while C_{n+1} looks like a fresh random value.

In G_1, because the whole chain of H calls was made in the right order, the current call will invoke case 3 of the simulator's algorithm and return

$$\mathsf{H}(C, v_n) = C_{n+1} \| \mathsf{chain}_n(v_0, \ldots, v_n)$$

or in the case of $n = m$

$$\mathsf{H}(C, v_m) = \mathsf{chain}_m(v_0, \ldots, v_m)\,.$$

The parts of H's result coming from chain are already known by the distinguisher, while C_{n+1} is a fresh random value. This is indistinguishable from the result in G_0.

Case 3. a) ii) In G_0, the current direct call to $\mathsf{H}(C, v_i)$ has already been done from inside a $\mathsf{chain}_n(v_0, \ldots, v_n)$ call, but *not as the last one*. This implies that said chain_n call was not chain_0 – this is covered by case 3. a) i).

In G_0, the result is thus a value fixed by a previous indirect call to H, but is independent of the results of previous direct calls, and thus looks like a fresh random value to the distinguisher.

In G_1, because the whole chain of H calls was made in the right order, the current call will invoke case 3 of the simulator's algorithm and return a result via a chain_n call. This chain_n call has not been made before by hypothesis and thus the result is a fresh random value.

Case 3. b) In G_0, this call to $\mathsf{H}(C, v_i)$ *has not* been done before, neither directly nor indirectly.[3] Hence, H returns a fresh random value.

In G_1, the simulator's case 1 is not relevant because this call has not been done before. Simulator's case 2: If $C = \mathsf{const}$, then H returns a fresh random C_1 and a fresh random r_0 via chain_0. This call to chain_0 has not been done before because this would have invoked the H call in G_0, which is excluded by the hypothesis. Simulator's case 3: If $((C_j, v_j), (C, r_j), j) \in L$ for some C_j, v_j, r_j and $0 \leq j < m$, then the current call to $\mathsf{H}(C, v_i)$ appends to a chain. Thus, a fresh random $C_{j+2} \| r_{j+1}$ is returned. The involved chain_{j+1} or chain_m has not been called before for the same reason as chain_0 above. Simulator's case 4: A fresh random $C_{-1} \| r_{-2}$ is returned. To conclude, a fresh random value is returned in every case in G_1.

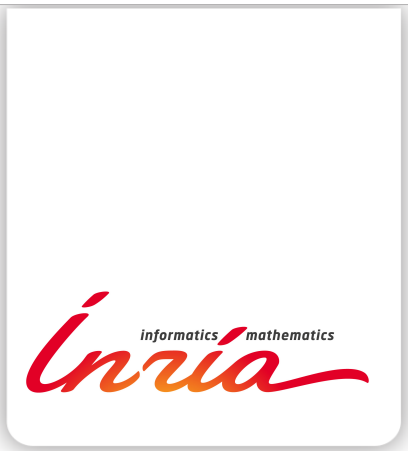
The previous proof shows that the games G_0 and G_1 are indistinguishable assuming the hypotheses H1, H2, and H3 hold. We will now bound the probability that they do not hold. Suppose that there are at most q_{H} direct queries to H and q_{chain_n} direct queries to chain_n.

[3] This means that there is no involvement of previous calls to chain_n, but the distinguisher can build an H chain with direct calls.

- When H1 does not hold, the distinguisher does an H call from a chain corresponding to an earlier or later chain_n call without having done the H calls starting from the beginning of the chain, by using the matching C value. There are at most $(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n})$ different C values from H, and the distinguisher has q_{H} attempts to hit a matching one, so the probability that H1 does not hold is at most $(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}) \cdot q_{\mathsf{H}}/2^{l'}$.
- The probability that H2 does not hold at the q-th call to H is at most the probability that a fresh random value in $\{0,1\}^{l'}$ collides with q values in $\{0,1\}^{l'}$, hence $q/2^{l'}$. So in total, the probability that H2 does not hold is $\sum_{q=1}^{q_{\mathsf{H}}} q/2^{l'} = q_{\mathsf{H}}(q_{\mathsf{H}}+1)/2^{l'+1}$.
- The probability that H3 does not hold is at most the probability that among q_{H} random values in $\{0,1\}^{l'}$, two of them collide, so it is at most $q_{\mathsf{H}}(q_{\mathsf{H}}-1)/2^{l'+1}$.

Hence, the probability that G_0 and G_1 are distinguished is at most

$$\frac{(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}) \cdot q_{\mathsf{H}} + q_{\mathsf{H}}^2}{2^{l'}} \,. \qquad \square$$



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399