A Mechanised Computational Analysis of the WireGuard Virtual Private Network Protocol

Master's Thesis of

Benjamin Lipp*

presented at

Institute of Theoretical Informatics (ITI)
Competence Center for Applied Security Technology (KASTEL)
Department of Informatics
Karlsruhe Institute of Technology

and prepared at

Prosecco Research Team
INRIA Paris

Reviewers:	Jörn Müller-Quade	(KIT)
	Dennis Hofheinz	(KIT)
Advisors:	Mario Strefler	(KIT)
	Karthikeyan Bhargavan	(INRIA)
	Bruno Blanchet	(INRIA)
	Harry Halpin	(INRIA)

November 24th, 2017 – May 23rd, 2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 23. Mai 2018

Benjamin Lipp*

Abstract

WireGuard is a new VPN software and protocol that is developed as free and open source project with academic support. It aims to replace IPsec and based VPNs like OpenVPN and will very soon be officially integrated into the Linux kernel. While this is desirable for performance reasons, flaws in the protocol or the implementation would be devastating. Ideally, this new implementation would be formally verified to have proved security guarantees from the start. This thesis lays the groundwork for that from the cryptographic point of view by contributing the first mechanised proof of the protocol under the computational model. In a future work, this proof can be formally linked with a formally verified implementation. We use the CryptoVerif proof assistant that produces proofs as sequence of games and calculates a probability negligible in the security parameter that bounds the attackers advantage to break the security properties (asymptotic security). Our attacker model is close to those of eCK and ACCE-like models and permits the attacker to compromise any key. Scenarios that would trivially break the protocol are excluded. We contribute proofs for message secrecy, forward secrecy, correctness, mutual authentication, as well as resistance against key compromise impersonation and unknown key-share attacks.

Zusammenfassung

In dieser Arbeit wird die erste maschinengestützte formale Analyse des WireGuard-Protokolls im Computational Modell vorgestellt. WireGuard ist ein neuer Virtual Private Network (VPN) Tunnel, der von Jason A. Donenfeld als Freie-Software-Projekt entwickelt und bald offiziell in den Linux-Kernel integriert wird. Unverifizierte Protokolle und unverifizierter Code im Kernel könnten schwerwiegende Sicherheitslücken hervorrufen, weswegen die hier vorgestellte Analyse sehr wichtig ist. WireGuard tritt an um die momentan verbreiteten VPN-Lösungen OpenVPN und IPsec zu ersetzen. Einige Analysen haben in der Vergangenheit Sicherheitsprobleme dieser Systeme herausgearbeitet. Diese resultieren unter anderem aus deren Zusammensetzung aus mehreren Schichten, sowie der Vielzahl an verfügbaren Protokollversionen und kryptographischen Primitiven.

WireGuard bricht mit dieser Flexibilität. Es nutzt ein festes Protokoll, das direkt auf UDP aufsetzt und dem Benutzer keine Freiheit in der Wahl der kryptographischen Primitive gibt. Es basiert auf dem kryptographischen Protokollframework Noise, das Diffie-Hellman-basierte Schlüsselaustauschprotokolle (informell) standardisiert. WireGuard ergänzt dies mit einem erweiterten Replayschutz, Identitätsschutz und Schutz gegen Distributed Denial of Service (DDoS) Angriffe. Der Identitätsschutz macht einerseits WireGuard-Paketströme verschiedener Nutzer auf kryptographischer Ebene ununterscheidbar. Andererseits wird garantiert, dass Endpunkte nur auf authentifizierte Pakete antworten. Ohne Kenntnis der kryptographischen Identität eines Servers kann dieser deswegen nicht zu einer Anwort gebracht werden. Damit wird auch ein Abscannen des gesamten IP-Adressraumes nach WireGuard-Endpunkten unmöglich. Unter Last können Endpunkte einen Cookie-Mechanismus verwenden, der den Kommunikationspartner zwingt, für jedes Paket einen Roundtrip zu investieren.

Die hier vorgestellte Analyse findet im Feld der Authenticated Key Exchange (AKE) Protokolle zwischen zwei Parteien statt, bei denen Langzeit- und Ephemeralkeys eingesetzt werden. WireGuard kann zusätzlich einen zuvor ausgetauschten symmetrischen Schlüssel in den Schlüsselaustausch einfließen lassen. Der in dieser Arbeit verwendete Beweisassistent CryptoVerif arbeitet im Computational Modell und erzeugt ausgehend von einem initialen Sicherheitsspiel einen Game-Hopping-Beweis unter Führung des Anwenders. Der Beweis ist für eine beliebige Anzahl paralleler Protokollsessions gültig, die polynomiell im Sicherheitsparameter ist. CryptoVerif berechnet eine asymptotische Schranke für das Brechen der Sicherheitseigenschaften. Die berechnete Formel ist detailliert genug, damit durch Einsetzen von konkreten Parametern und Laufzeiten eine exakte Sicherheitswahrscheinlichkeit berechnet werden kann. In dieser Arbeit wird ein Angreifermodell vergleichbar zu denen in bewährten Sicherheitsmodellen wie eCK und ACCE verwendet, bei dem der Angreifer alle möglichen im Protokoll verwendeten Schlüssel kompromittieren darf. Nur Szenarien, in denen das Protokoll trivialerweise gebrochen werden könnte, werden ausgeschlossen. WireGuard verwendet eine Kette von HKDF-Aufrufen um symmetrische

Schlüssel abzuleiten. In dieser Arbeit werden diese Ketten als Random Oracle modelliert und dieser Ansatz mit einem Indifferentiability-Beweis gerechtfertigt. CryptoVerif kann Secrecy und Korrespondenzen beweisen. Abfragen der ersten Art werden in dieser Arbeit verwendet um Message Secrecy, Forward Secrecy und eine weitere Secrecy-Eigenschaft für WireGuard zu beweisen. Korrespondenz-Abfragen werden verwendet um Korrektheit des Protokolls, gegenseitige Authentifizierung, sowie Widerstand gegen Key Comprimise und Unkown Key-Share Angriffe zu beweisen. Damit werden in dieser Arbeit alle Secrecy- und Korrespondenz-Eigenschaften im Computational Modell bewiesen, die bereits mit dem Beweisassistent Tamarin im symbolischen Modell bewiesen wurden. Einzig für den Fall des Kompromats beider Ephemeral Keys ist es dem Autor dieser Arbeit noch nicht gelungen, rechtzeitig einen Beweis zu finden. Für die bewiesenen Eigenschaften wird die asymptotische Schranke angegeben.

Contents

1. Introduction

This thesis is the first formally verification of WireGuard and a subset of the Noise Protocol Framework under the computational model [28, 29]. Formal verification is necessary for applications like Wireguard because they have kernel-level access to the user's computer.

In detail, this thesis is connected to a variety of currently undergoing research and development efforts: The WireGuard Virtual Private Network, the Noise Protocol Framework, and the larger effort to pursue the mechanisation of proofs as a means of precise formal analysis of protocols.

WireGuard [28, 29] is a recent development in the field of Virtual Private Networks (VPN), started by Jason A. Donenfeld to create a replacement for current VPN solutions like OpenVPN and L2TP VPNs using IPSec. Over the last years, these existing systems received several security analyses: The first version of IPSec and the Internet Key Exchange protocol IKEv1 were evaluated by Ferguson and Schneier in 1999 [32]. They mainly critizise the complexity of the protocol but also present several design flaws harming security. A more recent analysis of IKEv2 by Cremers [22] in 2011 shows that the updated protocol still doesn't meet all security properties it was designed for. OpenVPN, being based upon TLS, basically inherits all its security and complexity challenges as well as client certificate authentication. However with the SWEET32 attack [9] there is security research targeted directy at OpenVPN (and TLS). Besides VPN protocol security, there is research taking into account the whole system of a VPN user, analysing leakage of IPv6 packets and hijacking of DNS [50] – however, these kinds of attacks are out of scope of this work.

WireGuard's motivation is to provide a simpler, faster and more secure alternative to these existing systems, where simpler means at the same time simpler in protocol design and simpler to implement. The statements *simpler* and *faster* are backed by the fact that Donenfeld's implementation consists of less than 4000 lines of code (excluding cryptographic primitives), and by performance measurements comparing WireGuard, IPsec and OpenSSL. Work is already in progress to integrate WireGuard into the Linux kernel,[1] which creates a high incentive to verify WireGuard's claim for it to be *secure*, and thus the motivation to conduct thorough formal analysis of both the cryptographic design as well as the implementation. If WireGuard introduced a security flaw into kernel space, the impact would be devasting.

The cryptographic key exchange protocol used in WireGuard is built upon the IKpsk2 protocol from the **Noise Protocol Framework** which is an informal standardisation effort for cryptographic two-party protocols led by Trevor Perrin [49]. The Noise Protocol Framework defines in just 47 pages a concise and yet powerful language to describe protocols based on Diffie-Hellman key agreement that can use longterm and ephemeral

[1] `https://lkml.org/lkml/2017/11/10/666` – a mailing list post by Donenfeld summarising the state of the kernel patches

keys. This definition is precise in a way that besides the choice of the concrete Noise protocol and the employed cryptographic primitives – which both are indicated by a short Noise protocol name like `Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s` – there is no room left for ambiguous interpretation from implementors. WireGuard can so far be seen as the best use-case for a Noise protocol for different reasons: It is developed as a free and open source software project which makes it possible to study its use of Noise easily, and it will have a huge impact as soon as it is distributed within the Linux kernel. In terms of numbers of users there is another, larger adopter of a Noise protocol: WhatsApp uses Noise Pipes to secure client-server communication rather than TLS, which boosts the user count of the Noise protocol up to a billion [54, 53].

Existing Proofs. There are two conceptually different models in which security of a cryptographic protocol can be proved: The symbolic and the computational model. In the symbolic model, cryptographic primitives are considered perfect and the attacker is a Dolev-Yao attacker [27]. This means, as an example, that a ciphertext can only be decrypted if the right key is known. In this model, logical flaws and protocol traces can be found that lead to the violation of a security property. The computational model considers a more realistic attacker, described as a probabilistic Turing machine, and primitives that are not perfect, but are broken with a certain probability [33]. This permits to estimate a probability that security properties of the entire protocol can be broken, and, more constructively, to choose parameters like key sizes to keep this probability negligible in a real-world scenario.

While in [49], Perrin does define desirable cryptographic security properties for Noise protocols and lists for some of them which properties they achieve, no formal proofs are given to support these claims.

Currently, a comprehensive symbolic analysis by Nadim Kobeissi is underway verifying all claims stated by Perrin using the ProVerif protocol verifier, for arbitrary protocols expressed in the language defined by Noise. The models are generated entirely from the token-based language. A website presenting the results has recently been made available.[2] Donenfeld and Milner [30] did a symbolic analysis of WireGuard's protocol, that is Noise IKpsk2 with the additions made for DDoS protection. In their analysis they use Tamarin and are able to prove the properties of the underlying Noise protocol plus a notion of identity hiding and session uniqueness under the symbolic model.[3] The only computational analysis we are aware of is from Dowling and Paterson [31]. They provide a manual game hopping proof of WireGuard's protocol based on the PRF-ODH assumption. More precisely, their proof is for a slightly different but morally equivalent variant of the protocol: The authentication guarantee WireGuard gives only holds after the responder received the first transport data message from the initiator. This message serves as key confirmation and essentially interweaves key exchange and transport data phases.[4] This means the entire protocol cannot be proven in security models like CK [15], eCK [42] and eCK-PFS [23]: They are based on real-or-random key indistinguishability, which cannot be proven if

[2] `https://noiseexplorer.com/`

[3] In fact, Donenfeld maintains a page on WireGuard's project website collecting all formal results, which underlines the importance the project dedicates to formal methods.

[4] "Record layer" in TLS terms.

the key is used in the protocol.[5] Security models like ACCE [36], originally created to analyse TLS, permit to reason about protocols that internally use the key calculated by a key exchange. Instead of the keys, ACCE looks at the semantic security of the messages exchanged encrypted using the key: In the test session, a message indistinguishability game asks the attacker to decide which one of two provided plaintexts was encrypted. This model certainly is more involved and complex and seems to be generally avoided by cryptographers.[6] For the same reason, Dowling and Paterson decided to analyse a variant of WireGuard in an eCK model: They add a small key confirmation message to the protocol that permits to cleanly separate key exchange and the usage of the key. Because WireGuard includes the possibility to use a pre-shared key to strengthen the key exchange, they extend eCK-PFS to capture this notion and name their model eCK-PFS-PSK [31].

Our contribution. We contribute mainly a mechanised computational analysis of the WireGuard protocol. We say *mainly*, because logically our analysis starts with a manual proof of indifferentiability of a chain of HKDF calls from random oracles. In contrast to Dowling's and Paterson's analysis, we model the entire protocol and use the message indistinguishability approach. We use the CryptoVerif proof assistant [10] that permits to find game hopping proofs with a greater level of automation compared to EasyCrypt [3, 2], the only other major tool working in the computational model. EasyCrypt is more used for cryptographic primitives than protocols because of its more detailed ability to express transformations between games but at the same time the need for the user to formulate these games manually. CryptoVerif works on the level of assumptions on cryptographic primitives, and is for example currently not able to use the PRF-ODH assumption. It would require an extension of the code base to include it. This is one reason why we use the ROM-GapDH assumption. Another reason is that once the analysis of WireGuard is done, we can easily adapt the proof to other Noise protocols, and we believe that some of the Noise protocol variants might not be provable under PRF-ODH.

We want to motivate the use of proof assistants to avoid errors and improve readability of proofs. The problem of errors in proofs is real: A classic example is the Needham-Schroeder protocol that was published in 1978 [47]. It had a proof published in 1989 [13], but then an attack found using formal methods in 1995 [44, 45]. The Dual EC random bit generator had a serious security problem that was suspected to be a backdoor in 2014 [34]. The reasons for proof errors are thus manifold, including psychological bias when desiring to prove *security* of a system and not to find an attack, and overwhelmingly large state spaces of protocols that defy the capacity of a manual analysis. Security proofs are needed in order to understand the security reductions and assumptions of a system. Back to the technical aspects, game-based proofs are broadly viewed as a means to facilitate cryptographic security proofs of protocols [52, 6]. With more complex protocols like TLS and WireGuard however, even game-based proofs tend to get long, which makes it hard to proofread them. In 2004, Bellare and Rogaway use very clear words to describe how they perceive the situation: "In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor." [6]. Finally, agreeing with

[5] The problem here is that the attacker could simply test if it can decrypt the key confirmation message with the key provided by the test oracle. If yes, it received the real key, if not it received a random key.

[6] This model is also usually avoided by protocol designers. However, the wish for more efficient protocols seems to demand the interweaving of key exchange and transport data phase.

Halevi [35], they repeat his call for the creation of "automated tools, that can help write and verify game-based proofs" [35, 7].

This thesis aligns with the previous and ongoing projects done in the Prosecco research team at Inria. There has been work on the Signal protocol [38] and work on TLS [8], that both conduct symbolic and computational analysis of a real-world protocol that will eventually lead to the development of formally verified implementations using a language like F*. In the case of WireGuard, this approach is especially intruiging because of the possibility to integrate formally verified code into the Linux kernel. In general, establishing the formal link between a symbolic and computational analysis and a formally verified implementation is a larger ongoing research project.

Structure of the Thesis. In Chapter 2, we define the cryptographic primitives important for this work and give an overview of security models and properties for key exchange protocols. In Chapter 3, we describe the WireGuard VPN protocol and the Noise Protocol Framework. Chapter 4 contains our contribution, first giving an introduction into the proof technique and the tool used, then a description of our model of WireGuard in CryptoVerif and finally the results we obtained. In Chapter 5, we conclude and discuss future work.

2. Definitions of Cryptographic Primitives and Properties of Key Exchange Protocols

In this chapter we define the basic cryptographic notions needed for this work, and for some of them briefly mention where and why they are used in WireGuard. Unless stated otherwise, the definitions originate from [37].

When discussing security in the computational model, which we define more thouroughly later, we need a notion of negligible probabilities.

Definition 1 (Negligible Function)**.** A function f from the natural numbers to the non-negative real numbers is *negligible* if for every positive polynomial p there is an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.

Proposition 1. Let negl_1 and negl_2 be negligible functions. Then,

1. The function negl_3 defined by $\mathsf{negl}_3(n) = \mathsf{negl}_1(n) + \mathsf{negl}_2(n)$ is negligible.

2. For any positive polynomial p, the function negl_4 defined by $\mathsf{negl}_4(n) = p(n) \cdot \mathsf{negl}_1(n)$ is negligible.

The security parameter n, or 1^n in unary notation, captures the fact that cryptographic primitives are usually configurable in their key size and length of ciphertexts to name only two possibilities. The security parameter 1^n is, unless stated differently, always an implicit parameter to cryptographic primitives in the rest of this document. A cryptographic system is *asymptotically secure* if the probability of a successful attack is a negligible function in the security parameter 1^n. That is, there is a configuration with key lengths etc. such that the attack probability is negligible and so the primitive can be considered secure.

2.1. Cryptographic Primitives

Message Authentication Code (MAC) MACs serve to provide authenticity of messages, that is, an attacker cannot modify or create a message such that the recipient thinks it comes from a legitimate sender. For MACs, authenticity includes integrity. In WireGuard, MACs are used for protection against distributed denial of service (DDoS) attacks.

Definition 2 (Message Authentication Code)**.** A message authentication code (MAC) consists of three probabilistic polynomial-time algorithms (Gen, Mac, Vrfy) such that:

1. The key-generation algorithm Gen takes as input the security parameter 1^n and outputs a key k with $|k| \geq n$.

2. The tag-generation algorithm Mac takes as input a key k and a message $m \in \{0,1\}^*$, and outputs a tag t. We write this as $t \leftarrow \mathsf{Mac}_k(m)$.

3. The deterministic verification algorithm Vrfy takes as input a key k, a message m, and a tag t. It outputs a bit b, with $b = 1$ meaning *valid* and $b = 0$ meaning *invalid*. We write this as $b := \mathsf{Vrfy}_k(m, t)$.

It is required that for every n, every key k output by $\mathsf{Gen}(1^n)$, and every $m \in \{0,1\}^*$, it holds that $\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1$.

The security of a MAC can be defined in the following way:

Definition 3 (Existentially Unforgeable under Adaptive Chosen-Message Attacks)**.** A message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is *existentially unforgeable under an adaptive chosen-message attack*, or just *secure*, if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there is a negligible function negl such that:

$$\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathsf{MAC},\mathcal{A}}(n) = \Pr\left[\mathsf{Macforge}_{\mathcal{A},\Pi}(n) = 1\right] \leq \mathsf{negl}(n)\,,$$

where Macforge is given by the following game:

1. A key k is generated by running $\mathsf{Gen}(1^n)$.

2. The attacker $\mathcal{A}$ is given input 1^n and oracle access to $\mathsf{Mac}_k(\cdot)$. The attacker eventually outputs (m, t). Let Q denote the set of all queries that $\mathcal{A}$ asked its oracle.

3. $\mathcal{A}$ succeeds if and only if $\mathsf{Vrfy}_k(m, t) = 1$ and $m \notin Q$. In that case the output of the experiment is defined to be 1.

However, MACs are used in a non-standard way in WireGuard: The keys used depend entirely on public information, like the recipient's public key. The properties achieved by this construction will be briefly discussed in Section 3.2. In this work, we do not prove any security properties of the DDoS protection. Thus, we include a security definition of MACSs only for completeness.

Collision Resistant Hash Function Hash functions are commonly used to securely compress bitstrings of arbitrary length down to a fixed length. We define them below, and the definition of secure. WireGuard uses the hash function BLAKE2 in various places: A hash of the session's transcript is used as additional data to all ciphertexts, and BLAKE2 is used to instantiate HMAC and finally HKDF, which are defined later in this section.

Definition 4 (Hash Function)**.** A *hash function* (with output length l) is a pair of probabilistic polynomial-time algorithms (Gen, H) satisfying the following:

- Gen is a probabilistic algorithm which takes as input a security parameter 1^n and outputs a key s. We assume that 1^n is implicit in s.
- H takes as input a key s and a bitstring $x \in \{0,1\}^*$ and outputs a bitstring $H^s(x) \in \{0,1\}^{l(n)}$ (where n is the value of the security parameter implicit in s).

Definition 5 (Collision Resistant Hash Function)**.** A hash function $\Pi = (\mathsf{Gen}, H)$ is *collision resistant* if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there is a negligible function negl such that

$$\mathsf{Adv}^{\mathsf{coll-res}}_{\mathsf{HASH},\mathcal{A}}(n) = \Pr\left[\mathsf{Hashcoll}_{\mathcal{A},\Pi}(n) = 1\right] \leq \mathsf{negl}(n)$$

where $\mathsf{Hashcoll}_{\mathcal{A},\Pi}(n)$ is as follows:

1. A key s is generated by running $\mathsf{Gen}(1^n)$.
2. The attacker $\mathcal{A}$ is given s and outputs x, x'.
3. The output of the experiment is defined to be 1 if and only if $x \neq x'$ and $H^s(x) = H^s(x')$. In such a case we say that $\mathcal{A}$ has found a collision.

Note that hash functions are here defined as always having a key. However, in practice usually unkeyed hash functions are used, and this is not a problem because finding an actual collision is computationally hard.

Hash-Based Message Authentication Code HMAC is an industry standard for a hash-based MAC, that is very efficient and easy to implement. The interest of HMAC is the domain extension of MACs to arbitrary length bitstrings. HMAC is resistant against extension attacks, which is not the case for naive constructions using widely available hash functions that are mostly based on the Merkle-Damgård construction. We skip the details of the Merkle-Damgård construction, because BLAKE2 (the hash function used in WireGuard) is not constructed using Merkle-Damgård. The reason HMAC is used in WireGuard is because the Noise Protocol Framework is specified for different hash functions, and thus needs to be resistant against extension attacks in general.

Definition 6 (HMAC)**.** Let H be a hash function. Let `opad` and `ipad` be fixed constants of length n'. Define a MAC as follows:

- Gen: on input 1^n, uniformly choose and return $k \in \{0,1\}^{n'}$.
- Mac: on input a key k and a message $m \in \{0,1\}^*$, output

$$t := H((k \oplus \texttt{opad}) \| H(k \oplus \texttt{ipad} \| m)) .$$

- Vrfy: on input a key k, a message $m \in \{0,1\}^*$, and a tag t, output 1 if and only if $t = H((k \oplus \texttt{opad}) \| H(k \oplus \texttt{ipad} \| m))$.

Key Derivation Function Key derivation functions permit to create uniformly distributed bitstrings for use as shared keys, from data that is not necessarily uniformly distributed. Also, they permit to securely adapt the length of existing key material to the length required by a cryptographic primitive. HKDF is a particular key derivation function defined from HMAC in [40]:

Definition 7 (HKDF)**.** The HKDF key derivation function is defined as follows:

$$\begin{aligned}&\mathsf{HKDF}_n(salt, key, info) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad prk = \mathsf{HMAC}(salt, key)\\ &\quad k_1 = \mathsf{HMAC}(prk, info \| 0x00)\\ &\quad k_{i+1} = \mathsf{HMAC}(prk, k_i \| info \| i), \qquad \text{with } 1 \leq i < n\end{aligned}$$

salt is the extractor salt which may be null or constant, *key* is the source key material, and *info* is a "context information" string that can be used to bind key-related information to the produced key material. The variable i is a 1 byte value, and thus HKDF_n can return up to 256 blocks.

In WireGuard indeed, *info* is empty. HKDF is used in a chaining construction to continuously include key material into a finally extracted shared key.

Authenticated Encryption with Additional Data Authenticated Encryption with Additional Data (AEAD) is a shared key encryption scheme that guarantees secrecy and authentication. More precisely, it encrypts and authenticates messages, and it authenticates some additional data, usually called a *header*. Decryption in an AEAD scheme is defined in such a way that it returns only an error symbol in the case of failed authentication, and the plaintext in case of successful authentication.

Definition 8 (Authenticated Encryption with Additional Data [51])**.** We define an *authenticated encryption scheme with associated data* (an *AEAD scheme*) as a three-tuple of algorithms $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. Associated to Π are sets of bitstrings for nonces $\mathcal{N} = \{0,1\}^n$, messages $\mathcal{M} \subseteq \{0,1\}^*$, and headers $\mathcal{H} \subseteq \{0,1\}^*$.The key space $\mathcal{K}$ is a finite nonempty set of bitstrings. The key generation algorithm $\mathsf{KeyGen}(1^n)$ takes the security parameter as argument and returns $K \in \mathcal{K}$. The encryption algorithm Enc is a deterministic algorithm that takes bitstrings $K \in \mathcal{K}$ and $N \in \mathcal{N}$ and $H \in \mathcal{H}$ and $M \in \mathcal{M}$. It returns a bitstring $C = \mathsf{Enc}(K, N, H, M)$. The decryption algorithm Dec is a deterministic algorithm that takes bitstrings $K \in \mathcal{K}$, $N \in \mathcal{N}$, $H \in \mathcal{H}$, and $C \in \{0,1\}^*$. It returns $\mathsf{Dec}(K, N, H, C)$, which is either a bitstring in $\mathcal{M}$ or the distinguished symbol $\perp$. We require correctness by $\mathsf{Dec}(K, N, H, \mathsf{Enc}(K, N, H, M)) = M$ for all $K \in \mathcal{K}, N \in \mathcal{N}, H \in \mathcal{H}$ and $M \in \mathcal{M}$. $|\mathsf{Enc}(K, N, H, M)| = l(|M|)$ for some linear-time computable length function l.

In [51], a security definition is also given, based on IND$-CPA and INT-CTXT. CryptoVerif models AEAD using IND-CPA and INT-CTXT. We therefore define these two. Note that IND-CPA is a weaker security notion than IND$-CPA. In [4, 5], these are defined for authenticated encryption (without additional data). In this work, we adapt their definition for IND-CPA for AEAD as follows:

Definition 9 (Indistinguishability under Chosen Plaintext Attack for AEAD [4, 5])**.** An AEAD scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-CPA secure if for all probabilistic polynomial-time algorithms $\mathcal{A}$ the advantage of winning the following game is negligible.

- The challenger chooses a key $K \leftarrow \mathsf{KeyGen}$ and a random bit $b \leftarrow\!\!\$\ \{0,1\}$, and sets $L \leftarrow \emptyset$.
- $\mathcal{A}$ has access to a left-or-right encryption oracle $\mathsf{LR}(N, H, M_0, M_1)$ that, provided with a nonce $N \in \mathcal{N}$, a header $H \in \mathcal{H}$, and two equal-length messages $M_0, M_1 \in \mathcal{M}$, returns $\perp$ if $N \in L$ and thus N was already used in a query to this oracle, and $C \leftarrow \mathsf{Enc}(K, N, H, M_b)$ otherwise. In the latter case it adds N to L before returning.
- $\mathcal{A}$ finally outputs a bit d and wins the game if $d = b$.

We define $\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathsf{AEAD},\mathcal{A}}(n) = \Pr[d = b] - \frac{1}{2}$.

This definition clarifies that no guarantees are given in case of nonce reuse. Also, this definition assumes that messages of equal length are encrypted to ciphertexts of equal length. If the two provided messages do not have equal length, this security definition does not guarantee anything.

Definition 10 (Ciphertext Integrity for AEAD [4, 5])**.** An AEAD scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is INT-CTXT secure if for all probabilistic polynomial-time algorithms $\mathcal{A}$ the advantage of winning the following game is negligible.

- The challenger chooses a key $K \leftarrow \mathsf{KeyGen}$, and sets $S, L \leftarrow \emptyset$.
- $\mathcal{A}$ has access to an encryption oracle $\mathsf{Enc}(N, H, M)$ that, provided with a nonce $N \in \mathcal{N}$, a header $H \in \mathcal{H}$, and a message $M \in \mathcal{M}$, returns $\perp$ if N was already used in a query to this oracle, and $C \leftarrow \mathsf{Enc}(K, N, H, M_b)$ otherwise. In the latter case it adds N to L and (C, N, H) to S.
- $\mathcal{A}$ has access to a verification oracle $\mathsf{VF}(N, H, C)$ that, provided with a nonce $N \in \mathcal{N}$, a header $H \in \mathcal{H}$, and bitstring C, proceeds as follows: It sets $M \leftarrow \mathsf{Dec}(K, N, H, C)$. If $M \neq \perp$ and $(C, N, H) \notin S$, it sets $\mathsf{win} \leftarrow \mathsf{true}$. Finally, it returns $(M \neq \perp)$.
- $\mathcal{A}$ finally calls the oracle $\mathsf{Finalise}$ that returns the bit win.

We define $\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathsf{AEAD},\mathcal{A}}(n) = \Pr[\mathsf{win} = \mathsf{true}]$.

Note that the attacker also wins if it can get the verification oracle to accept a ciphertext produced by the encryption oracle with *different* nonce or header.

Random Oracle The random oracle is the idealisation of a hash function. We use the following definition:

Definition 11 (Random Oracle)**.** A random oracle H is a function with arbitrary length input, possibly separated into multiple input variables, and with a bitstring output of length l. It behaves as follows:

- If H is called with arguments it has never seen before, it returns a new uniformly random value from the output space.
- If H is called with arguments it was called with before, it returns the same result.

This implies that without calling H, its result for a tuple of arguments can only be guessed with probability $1/2^l$.

A random oracle is not actually implemented by a hash function as it would require exponential storage and exponential time to evaluate output, which is impractical for hash functions in real world usage [14]. We can however create proofs with a model of the world in which random oracles exist, the *random oracle model*. In contrast, the model that does not assume the existence of random oracles is the *standard model*. If a cryptographic system can be proved secure in the standard model, this is a stronger property than a proof in the random oracle model. However, there are cryptographic systems that cannot be proved secure in the standard model. The opinions of cryptographers are split about the question of the utility of a security proof in the random oracle model, but many agree that it is better than no proof at all, as a proof with the random oracle model still can prove attacks are not possible that are not dependent on hashing. Therefore, we will work with the random oracle model in this thesis.

Diffie-Hellman Key Exchange In 1976, Diffie and Hellman discovered the Diffie-Hellman key exchange protocol [24]. Its security is based on assumptions we introduce in the following. The protocol is described at this point to motivate these definitions.

We let $\mathcal{G}$ denote a generic, polynomial-time, group-generation algorithm. This is an algorithm that, on input 1^n, outputs a description of a cyclic group $\mathbb{G}$, its order q (with $|q| = n$), and a generator $g \in \mathbb{G}$.

Definition 12 (The Diffie-Hellman Key Exchange Protocol)**.** Two parties A and B have as common input the security parameter 1^n. The protocol runs as follows:

1. A runs $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$.
2. A chooses a uniform $x \in \mathbb{Z}_q$, and computes $u := g^x$.
3. Alice sends $(\mathbb{G}, q, g, u)$ to Bob.
4. B receives $(\mathbb{G}, q, g, u)$. He chooses a uniform $y \in \mathbb{Z}_q$, and computes $v := g^y$. B sends v to A and outputs the key $k_B := u^y = g^{xy}$.
5. A receives v and outputs the key $k_A := v^x = g^{xy}$.

This protocol is a key exchange protocol secure against a *passive* attacker, if the decisional Diffie-Hellman assumption holds. On the way to define this assumption, we first define the discrete-logarithm experiment, that is the base for all following assumptions.

Definition 13 (The Discrete-Logarithm Experiment $\mathsf{DLog}_{\mathcal{A},\mathcal{G}}(n)$)**.**

1. Run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$, where $\mathbb{G}$ is a cyclic group of order q (with $|q| = n$), and g is a generator of $\mathbb{G}$.
2. Choose a uniform $h \in \mathbb{G}$.
3. $\mathcal{A}$ is given $\mathbb{G}, q, g, h$, and outputs $x \in \mathbb{Z}_q$.
4. The output of the experiment is defined to be 1 if $g^x = h$, and 0 otherwise.

Definition 14 (The Discrete-Logarithm Assumption)**.** We say that the *discrete-logarithm assumption* holds relative to $\mathcal{G}$ if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl such that $\Pr\left[\mathsf{DLog}_{\mathcal{A},\mathcal{G}}(n) = 1\right] \leq \mathsf{negl}(n)$.

Based on this assumption, we define more involved assumptions that are used in security proofs.

Definition 15 (The Computational Diffie-Hellman Assumption)**.** We say that the *CDH assumption* holds relative to $\mathcal{G}$ if for all probabilistic polynomial-time algorithms $\mathcal{A}$ the probability to win the following game is negligible. The game runs as follows:

- The challenger uniformly chooses two random elements $x, y \in \mathbb{Z}_q$ and calculates $u = g^x, v = g^y, w = g^{xy}$ and gives the triple (g, u, v) to $\mathcal{A}$.
- $\mathcal{A}$ outputs some $w' \in \mathbb{G}$.

$\mathcal{A}$ wins the game if $w' = w$. We define $\mathsf{Adv}^{\mathrm{cdh}}_{\mathcal{G},\mathcal{A}}(n) = \Pr[w' = w]$.

The computational Diffie-Hellman assumption is stronger than the discrete logarithm assumption. The CDH assumption guarantees that a passive adversary cannot compute the shared key in the Diffie-Hellman protocol.

Definition 16 (The Decisional Diffie-Hellman Assumption)**.** We say that the *DDH assumption* holds relative to $\mathcal{G}$ if for all probabilistic polynomial-time algorithms A there is a negligible function negl such that

$$\mathsf{Adv}^{\mathrm{ddh}}_{\mathcal{G},\mathcal{A}}(n) = \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] \leq \mathsf{negl}(n)\,,$$

where in each case the probabilities are taken over the experiment in which $\mathcal{G}(1^n)$ outputs $(\mathbb{G}, q, g)$, and then uniform $x, y, z \in \mathbb{Z}_q$ are chosen. (Note that when z is uniform in Z_q, then g_z is uniformly distributed in $\mathcal{G}$.)

The decisional Diffie-Hellman assumption is stronger than the computational Diffie-Hellman assumption. For the Diffie-Hellman protocol, the DDH assumption guarantees that a passive adversary cannot distinguish the computed key from a randomly chosen key. This is needed to show its semantic security based on a real-or-random key indistinguishability game.

Gap problems have been defined in [48]. In some Diffie-Hellman based protocols, parties expose a Decisional Diffie-Hellman oracle: The attacker can test if a party responds

to a forged protocol message. Basing the security of these protocols only on the CDH assumption would thus be a too weak assumption.

Definition 17 (The Gap–Diffie-Hellman Problem)**.** We say that the *GDH problem* is hard relative to $\mathcal{G}$ if for all probabilistic polynomial-time algorithms $\mathcal{A}$ the probabability $\mathrm{Adv}^{\mathrm{cdh}}_{\mathcal{G},\mathcal{A}}(n)$ to solve the CDH problem for a given triple (g, g^x, g^y) is negligible even with access to a Decisional Diffie-Hellman Oracle (which answers whether a given quadruple (g, g^x, g^y, g^{xy}) is a Diffie-Hellman quadruple or not). We call the according probability and advantage $\mathrm{Adv}^{\mathrm{gdh}}_{\mathcal{G},\mathcal{A}}(n)$.

The authors of [48] argue in their paper why the GDH assumption is reasonable, and it has proven to be useful in practice to prove the security of primitives that cannot be proven under only decisional assumptions and protocols where access to such an oracle for forged messages is practical.

2.2. Authenticated Key Exchange Protocols

Authenticated Key Exchange (AKE) protocols permit parties to securely agree on a shared key. For instance, these protocols can be used to establish a *secure channel*.

Definition 18 (Party)**.** A party is identified by its long-term public key. We call a party *honest* if they follow the protocol. We call a party *dishonest* if they may not follow the protocol.

Establishing trust in long-term keys, that is linking them to identities, is out of scope of our analysis: We consider long-term keys to be known by all parties and we consider that public keys and identities are linked by some other protocol, like a certification authority.

Definition 19 (Session, Partner Session (informal))**.** A party may run the considered authenticated key exchange protocol with multiple parties sequentially and in parallel, and also sequentially and in parallel with a same party. Each execution of the protocol is called a *session*. The *partner session* to a session started by a party A is the session of the other *honest* party B, A intends to talk to.

A session does not necessarily have a partner session, that is, if the session is established with a dishonest party.

Definition 20 (Ephemeral Keys)**.** Ephemeral keys are keys that are freshly chosen per session by the parties.

Ephemeral keys are an essential ingredient for a protocol to satisfy the properties we define later.

We limit ourselves to the case of two parties, as that is the assumption used in WireGuard. There is also published research for the case of more than two parties, but for these the term “group key exchange” is usually used. Key exchange protocols are therefore bootstrapping the use of shared (symmetric) key cryptography. The definition of *secure*

has evolved over the last two decades and now includes a variety of interesting and important properties that can be summarised under the notions *secrecy* and *authentication*. Secrecy divides into key secrecy, message secrecy, forward secrecy, and post-compromise secrecy. Authentication divides further into mutual authentication, resistance against key impersonation attacks, and resistance against unknown key-share attacks. We will define these properties formally later in this section. Besides stating that a certain protocol has a subset of these properties, it is equally important to say in which attacker model these statements hold. An attacker model describes the capabilities of the attacker. The Dolev-Yao attacker (the symbolic model) vs. the probabilistic Turing machine (the computational model) was briefly mentioned in the introduction and we will elaborate on this now.

In the **symbolic model**, a Dolev-Yao attacker has access to the entire network, and can read, delay, reorder, block and replay all messages transmitted over it. It can also calculate and inject its own messages into the network. These messages can be calculated using everything transmitted over the network so far, and the cryptographic primitives available in the protocol. Calculation is limited: The primitives are modeled as perfect, that is, a calculation succeeds if and only if the attacker knows the exact arguments. In the case of encryption this means a message can only be decrypted if the attacker has the exact key the encrypted message was originally encrypted with. That is why this attacker model is called the symbolic model: Ciphertexts, keys, and variables in general are treated as symbols and no knowledge about any internal structure of them can be exploited by the attacker.

In the **computational model**, variables are bitstrings, which makes their internal structure exploitable by the attacker. It is allowed to do any probabilistic polynomial calculation. With this addition, the attacker model is much more realistic than the symbolic model: We allow the attacker to do any calculation current computers can do. We keep track of calculability by adding the notions of time and sizes to the model. Security is then based on the assumption that certain problems are hard to solve for a given problem size in a given time. The computational model therefore can model attacks not present in the symbolic model.

We have now described how the attacker can access the network and what it can calculate. It is left to describe what access the attacker has to the honest parties participating in a protocol. The most important aspect to model in key exchange protocols is the attacker's ability to **compromise keys of the parties**. This models in an abstract way what could happen in the real world in a lot of different concrete attacks: A private key could for example unintentionally be uploaded or stolen by malicious software. The more capabilities the attacker has, the stronger is the security model. The modelling has to be done carefully though: If the attacker has too much power, it can trivially break the protocol. We cannot prove security in such a case, and the model would be useless. If the attacker could for example compromise the long-term *and* ephemeral keys of a party's session, it would have all necessary information to calculate the shared secret. Such compromise scenarios need to be excluded in a meaningful attacker model.

This is usually done by defining the notion of a *test session*:

Definition 21 (Test Session)**.** The attacker can choose a session as the test session. For this session *and its partner session(s)*, it is not allowed to conduct trivial compromise scenarios. On all other sessions there are no limits; security models usually also expose a Reveal query to the attacker so it can get the calculated shared key of those other sessions. If the attacker cannot break the security property of the test session, the protocol is defined as secure.

Different attacker models have been proposed in the literature. Usually, a new model strengthens a previous one and captures the needs of new protocols. In the CK model [16], the attacker is not allowed to compromise ephemeral keys in the test session. Also, the model does not cover resistance against key impersonation attacks. This is because sessions that start after the parties long-term key was compromised cannot be test sessions. The authors of [42] include these possibilities in their model called extended CK, which is abbreviated as eCK. In [23], the model is extended to include *perfect forward-secrecy* (PFS), and thus is the eCK-PFS model. The extension of the model was possible by adapting the definition of partner sessions. In 2011, Cremers attempted a comparison of some of the above mentioned and other models [21]. He showed that the models are actually incomparable: Security in one model does not imply security in the other models. These underlines that the space of possible models for authenticated key exchange is not yet settled. In Section 4.3, we will elaborate on how the security notions are defined in our model and briefly outline differences to the other models mentioned here.

2.2.1. Security Properties of Authenticated Key Exchange

Definition 22 (Key Secrecy)**.** The shared key established by an authenticated key exchange protocol is secret, if it is only known by the eligible parties.

This is usually modeled by a real-or-random indistinguishability game: For a test session, the attacker receives either the key calculated by the protocol, or a random key from the same key space. If the attacker can distinguish the two cases only with negligible probability, the protocol ensures key secrecy.

The terms *secrecy* and *confidentiality* can be used synonymously. Key secrecy is proven in CK and eCK-like models for protocols that do not use the shared key.

Definition 23 (Message Secrecy)**.** The plaintext sent encrypted over a secure channel established by an authenticated key exchange protocol is only known by the eligible parties.

This is usually modeled by a left-or-right ciphertext indistinguishability game. For a test session, the attacker provides two equal-length plaintexts to the challenger. One of them gets encrypted with the shared key of the session. If the attacker can distinguish the two cases only with negligible probability, the protocol ensures message secrecy.

Message Secrecy is proven in ACCE-like models for protocols that *use* the shared key. This is also the case for the WireGuard protocol. If we play a real-or-random key

indistinguishability game when the key is used by the protocol, the attacker can simply attempt decryption of the appropriate ciphertext using the key he received. If decryption succeeds, it received the real key, if it failed, the attacker received a random key.

Definition 24 (Forward Secrecy)**.** Plaintexts sent encrypted over a secure channel that was established by an authenticated key exchange protocol *before* the compromise of one or both party's longterm keys cannot be decrypted, thus the plaintexts stay secret "forward in time".

This is usually modeled like key secrecy or message secrecey, by allowing compromise of the party's longterm keys *after* a test session has agreed on a key or sent a message.

Definition 25 (Correctness)**.** If two parties successfully complete a key exchange in sessions that are partnered, they calculate the same key.

If a protocol satisfies this definition, this simply means that the protocol at least works.

Definition 26 (Key Authentication)**.** If a party A believes to have established a key with a party B, then B also believes to have established this key with A.

If this property holds in both directions, this is called mutual key authentication [18].

Definition 27 (Message Authentication, Mutual Message Authentication)**.** If a party A believes a message received was sent from B, then B has indeed sent this message to A.

If an authenticated protocol guarantees this for both directions, this property is called *mutual message authentication.*

The following property was first formalised by Krawczyk for the HMQV protocol [41].

Definition 28 (Resistance Against Key Compromise Impersonation Attacks)**.** Suppose the longterm key of a party A is compromised. If an attacker can execute a session with A, successfully pretending to be any other party, this is called a *key compromise impersonation attack* (KCI).

If such attacks are not possible in a protocol, it is said to be resistant against KCI.

This can be illustrated with a Diffie-Hellman key exchange based only on two longterm keys. We have

$$\mathsf{DH}(S_A^{priv}, S_B^{pub}) = \mathsf{DH}(S_B^{priv}, S_A^{pub})\,.$$

If the attacker knows S_A^{priv}, it can calculate the Diffie-Hellman function for any other party by just using their public key.

Definition 29 (Resistance Against Unknown Key-Share Attacks [18])**.** If two honest parties A and B calculate a same shared key, and A believes to have established this key with B, but B believes to have established the key with a third party E, this is called an *unilateral unknown key-share attack* (UUKS).

A *bilateral unknown key-share attack* (BUKS) is an attack whereby two honest parties A and B end up sharing a key between them but A believes it shares the key with another

party C, and B believes it shares the key with another entity D, where C is not equal to B and D is not equal to A.

If such attacks are not possible in a protocol, it is said to be resistant against unknown key-share attacks (UKS).

Note that mutual key authentication implies resistance against unilateral UKS. In bilateral UKS, the parties C and D may or may not be the same entity, and they may or may not be honest. Because they may be dishonest, bilateral UKS is not excluded if the protocol guarantees key authentication: It makes no sense to make a statement about what a dishonest party believes to have agreed upon.

3. The WireGuard Virtual Private Network Protocol

The WireGuard VPN permits two parties to establish a secure channel using a Diffie-Hellman based key exchange protocol. It works on layer 3 and uses UDP as transport layer. WireGuard employs a new "cryptokey routing" technique to configure routes to endpoints, and provides new identity hiding and DDoS resistance features, the latter based on a cookie reply system. The handshake is based on a longterm and an ephemeral elliptic curve key. Security can be strengthened by an optional pre-shared symmetric key. If the handshake fails, then the protocol needs to restart, as UDP does not permit to detect package loss. After the handshake, the protocol running over the secure WireGuard channel, for example TCP, can perform all its internal package loss detection techniques. WireGuard itself does not detect package loss but only employs a sliding window technique to reorder packages. A new handshake is performed every two minutes or after a number of sent packages predefined in the specification, to prevent the possibility of collision attacks on the stream cipher.

3.1. The Noise Protocol Framework

WireGuard uses a protocol from the Noise Protocol Framework as the basis for its key exchange protocol. The Noise Protocol Framework standardises a variety of cryptographic two-party key exchange protocols based on Diffie-Hellman key exchange, agnostic of the transport used. It defines a concise language to define protocols, based on tokens like *e* for ephemeral key, *s* for static longterm key, *psk* for preshared key, and two-letter combinations of *e* and *s* that stand for Diffie-Hellman operations between keys of the two parties: *ee*, *es*, *se*, *ss*, where the first letter denotes the participating key from the initiator, and the second letter the participating key for the responder. The protocol used by WireGuard, Noise IKpsk2, looks like follows, and we will explain the notation step by step:

```
IKpsk2(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se, psk
```

The first part are the two protocol messages at the end:

```
  -> e, es, s, ss
  <- e, ee, se, psk
```

The first one, denoted by `->` is sent from initiator to responder, the second one, denoted by `<-` in the other direction. Noise defines precise processing rules for the tokens that make up a message. These rules affect the `CipherState`, `SymmetricState`, and `HandshakeState`, each party holds locally. Single-letter tokens like *e* and *s*, for the according public keys, are sent over the network, and also contribute to a session transcript hash. Two-letter tokens and `psk`, do not get sent over the network but get immediately mixed into a key derivation function to derive a new symmetric key. This key is then used to encrypt the next token that gets sent over the network. In our example, *e* means that the initiators ephemeral public key gets sent in the clear to the responder. The initiators public key *s* however is sent encrypted with a symmetric key derived from the *es* Diffie-Hellman operation. A possible payload of the first protocol message is then again encrypted with a new key, that is derived from the key material before *and* the *ss* Diffie-Hellman operation. Indeed, in WireGuard, a timestamp is sent as payload of the first protocol message, encrypted with this key. The second protocol message is handled in the same manner, although in this example we do not elaborate on the details how the pre-shared key is mixed into the key derivation.

The first two lines of the protocol description are the so-called pre-messages:

```
<- s
...
```

To be able to calculate the *ss* Diffie-Hellman, the initiator needs to know the responder's static public key. This is denoted by a message `<- s` that is sent from the responder to the initiator. It is out of scope for Noise (as for WireGuard), how this message is transmitted. It is assumed that the parties use some other protocol to establish the knowledge of the static public key. The last part is the protocol name and the parameters:

```
IKpsk2(s, rs):
```

The protocol name, IKpsk2, can be divided into the two parts `IK` and `psk2`.

The two letters `IK` mean, that the initiator immediately sends its static key to the responder in the first protocol message (`I` for immediately), and the the initiator knows the responder's static key beforehand (`K` for known). The `2` at the end of `psk2` defines that the pre-shared key is used at the end of the second protocol message. The parameters `(s, rs)` state which values need to be given to an implementation to execute the protocol. In this case, it is the two static keys, *s* for the one of the initiator, and *rs* the remote static key, from the responder.

The Motivation Behind Noise. It can be argued that the TLS protocol suite provides already all building blocks for secure channels. However, the TLS standard is huge, and TLS libraries are large codebases. Noise provides a lightweight and simple alternative, that can be attractive for projects that have a specific need and explicitely do not need or do not want to inherit the whole TLS stack.

Why WireGuard Chose IKpsk2. The author of WireGuard wanted to use a 1-RTT key exchange. The XK Noise pattern would provide forward-secret identity hiding, but needs

an addional message. To be more precise, WireGuard uses a 1.5-RTT key exchange protocol, because the third protocol message is needed to establish mutual authentication (it proves that the initiator controls its ephemeral key). Noise specifies different possible locations for the pre-shared key. WireGuard chose to place it at the end, because VPN servers can then look up the needed pre-shared key in some database after they already authenticated the initiator based on its static long-term key.

3.2. Protocol Messages and Key Derivation

The *initiator* is the party that starts a protocol session by sending the first protocol message to another party, which is called the *responder*. These roles do not in general correspond to the typical roles of *client* and *server* in a VPN scenario, where the client is the customer and the server is a machine of a VPN provider. While the client typically is the protocol's initiator for the first handshake, the roles might change during a longer VPN "session."

The WireGuard protocol [29] consists of 3 protocol messages that are needed such that mutual authentication between initiator and responder is guaranteed. It can therefore be considered as a 1.5-RTT key exchange protocol. The third message, which is sent from initiator to responder, is already a transport data message and can contain real data. The authenticated encryption in this third message permits the responder to authenticate the initiator. After these three messages, transport data messages can be exchanged in any order between the parties. The following description of the protocol is largely based on the original paper [29]. We leave out some technical details that are not important for the cryptographic proof. Thus, our description must not serve as a basis for implementation.

Notation. Variables belonging to the initiator are indicated with an index i, those belonging to the responder with r. The subscript asterisk $*$ means both of them. In (parts of) messages that can be sent by both parties, if the initiator creates it, let $(m, m') = (i, r)$, and if the responder creates it, let $(m, m') = (r, i)$. The operator $\|$ means concatenation of bitstrings. Assignments of values to variables from deterministic algorithms are denoted with $\leftarrow$, from probabilistic algorithms with $\leftarrow_{\$}$. Given an integer value n, $\hat{n}$ has the value $n + 16$, where 16 is the length of the authentication tag. The empty bitstring is denoted by ϵ. 0^n represents the all zero bitstring of length n bytes, and ρ^n a random bitstring of length n bytes.

Cryptographic Primitives, Functions, and Constants. The WireGuard protocol uses the following functions [29]:

- DH($privatekey, publickey$): WireGuard uses the elliptic curve Curve25519, and this function is the point multiplication of private key and public key, returning 32 bytes of output.

- DH–Generate(): Generates a random Curve25519 private-public key pair, returning a pair of 32 bytes values, ($privatekey, publickey$).

- AEAD($key, counter, plaintext, authtext$): The ChaCha20Poly1305 AEAD [43], its nonce being composed of 32 bits of zeros followed by the 64-bit little-endian value of $counter$.
- HASH($input$): BLAKE2s($input$, 32), returning 32 bytes of output.
- MAC($key, input$): Keyed–BLAKE2s($key, input$, 16), the keyed MAC variant of the BLAKE2s hash function, returning 16 bytes of output.
- HMAC($key, input$): HMAC–BLAKE2s($key, input$, 32): the ordinary BLAKE2s hash function used in an HMAC construction, as defined in Section 2.1, returning 32 bytes of output.
- $\text{HKDF}_n(key, input)$: The HKDF function as defined in Section 2.1, with an empty *info* bitstring.
- TIMESTAMP(): Returns the TAI64N timestamp of the current time, which is 12 bytes of output.
- Construction: The UTF-8 string literal "`Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s`", spaces denoted by ␣.
- Identifier: The UTF-8 string literal "`WireGuard␣v1␣zx2c4␣Jason@zx2c4.com`".
- Label–MAC1: The UTF-8 string literal "`mac1----`".
- Label–Cookie: The UTF-8 string literal "`cookie--`".

Local State. The parties calculate or receive during a protocol run, and keep as local state (at least temporarily) the following variabes:

- $I_m, I_{m'}$: 32-bit indices that locally represent the current session (I_m), and the other peer of this session ($I_{m'}$).
- $S_m^{priv}, S_m^{pub}, S_{m'}^{pub}$: The own longterm private and public key, and the other peers longterm public key. In Noise and WireGuard, they are called *static* keys and are therefore denoted with S.
- $E_m^{priv}, E_m^{pub}, E_{m'}^{pub}$: The own ephemeral private and public key, and the other peers ephemeral public key.
- Q: The optional pre-shared key. If pre-shared key mode is not used, Q is set to a 32 byte bitstring of zeros.
- H_m, C_m: A hash result and a chaining key.
- T_m^{send}, T_m^{recv}: Transport data symmetric keys for sending and receiving with AEAD.
- N_m^{send}, N_m^{recv}: Transport data nonce counters for sending and receiving with AEAD.

First Protocol Message. The first protocol message `msg` is sent from initiator to responder:

`type` $\leftarrow 0x1$ (1 byte)	`reserved` $\leftarrow 0^3$ (3 bytes)
`sender` $\leftarrow I_i$ (4 bytes)	
`ephemeral` $\leftarrow E_i^{pub}$ (32 bytes)	
`static` ($\hat{32}$ bytes)	
`timestamp` ($\hat{12}$ bytes)	
`mac1` (16 bytes)	`mac2` (16 bytes)

The value I_i is randomly chosen as ρ^4 when preparing the message.

$$
\begin{aligned}
C_i &\leftarrow \mathsf{HASH}(\text{Construction}) \\
H_i &\leftarrow \mathsf{HASH}(C_i \| \text{Identifier}) \\
H_i &\leftarrow \mathsf{HASH}(H_i \| S_r^{pub}) \\
(E_i^{priv}, E_i^{pub}) &\leftarrow_{\$} \mathsf{DH\text{-}Generate}() \\
C_i &\leftarrow \mathsf{HKDF}_1(C_i, E_i^{pub}) \\
H_i &\leftarrow \mathsf{HASH}(H_i \| E_i^{pub}) \\
C_i \| k &\leftarrow \mathsf{HKDF}_2(C_i, DH(E_i^{priv}, S_r^{pub})) \\
\texttt{msg.static} &\leftarrow \mathsf{AEAD}(k, 0, S_i^{pub}, H_i) \\
H_i &\leftarrow \mathsf{HASH}(H_i \| \texttt{msg.static}) \\
C_i \| k &\leftarrow \mathsf{HKDF}_2(C_i, DH(S_i^{priv}, S_r^{pub})) \\
\texttt{msg.timestamp} &\leftarrow \mathsf{AEAD}(k, 0, \mathsf{TIMESTAMP}(), H_i) \\
H_i &\leftarrow \mathsf{HASH}(H_i \| \texttt{msg.timestamp}) \\
\texttt{msg.mac1} &\leftarrow \mathsf{MAC}(\mathsf{HASH}(\text{Label-MAC1} \| S_r^{pub}), \texttt{msg}_\alpha)
\end{aligned}
$$

In the computation of `mac1`, the value $\texttt{msg}_\alpha$ represents all bytes of the message prior to `mac1`. We do not describe the computation of `mac2`, because we did not include it into our model.

The responder can, by first verifying `mac1`, make sure that the sender knows his identity. Note that the MAC key, besides the constant, depends on S_r^{pub}, which is also used in the chained hash, which could also be used to verify this fact. However, the MAC permits to verify that this is no randomly sent message *before* computing expensive DH functions. Also, `mac1` prevents a WireGuard endpoint from being detected by someone scanning the internet.

After verifying the MAC, the responder can perform the same computations than the initiator, just by replacing the parameters of the DH function accordingly. This will result in the same values for $C_r = C_i$ and $H_r = H_i$. Note how a chaining hash of parts of the protocol transcript is used as additional data in the AEAD. This ties each ciphertext to a message and prevents injecting the ciphertext into other sessions. Also, with Construction and

Identifier, the message is tied to a specific version of Noise and WireGuard, prohibiting attacks exploiting different protocol versions (where, at the moment, there no other protocol versions yet).

In terms of the Noise protocol, the timestamp is the payload of the first protocol message. The MACs are not specified in Noise at all.

Second Protocol Message. This message `msg` is sent from the responder to the initiator as response to the first protocol message:

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

The value I_r is chosen randomly as ρ^4 when preparing this message. The responder calculates the message as follows:

$$
\begin{aligned}
(E_r^{priv}, E_r^{pub}) &\longleftarrow_{\$} \mathsf{DH\text{–}Generate}() \\
C_r &\leftarrow \mathsf{HKDF}_1(C_r, E_r^{pub}) \\
H_r &\leftarrow \mathsf{HASH}(H_r \| E_r^{pub}) \\
C_r &\leftarrow \mathsf{HKDF}_1(C_r, DH(E_r^{priv}, E_i^{pub})) \\
C_r &\leftarrow \mathsf{HKDF}_1(C_r, DH(E_r^{priv}, S_i^{pub})) \\
C_r \| \tau \| k &\leftarrow \mathsf{HKDF}_3(C_r, Q) \\
H_r &\leftarrow \mathsf{HASH}(H_r \| \tau) \\
\texttt{msg.empty} &\leftarrow \mathsf{AEAD}(k, 0, \epsilon, H_r) \\
H_r &\leftarrow \mathsf{HASH}(H_r \| \texttt{msg.empty}) \\
\texttt{msg.mac1} &\leftarrow \mathsf{MAC}(\mathsf{HASH}(\mathsf{Label\text{–}MAC1} \| S_i^{pub}), \texttt{msg}_\alpha)
\end{aligned}
$$

The MAC equally permits the initiator to protect itself from denial of service attacks. Additionally, the second protocol message is smaller than the first. This way, a responder cannot be misused as an amplificator for denial of service attacks. After receiving this message, the initiator can perform the same computations, replacing the parameters to DH accordingly.

In terms of Noise, the second protocol message has no payload, but an empty payload is obligatory for authentication of the transcript.

Derivation of Transport Data Keys. After the second protocol message, both parties can compute the shared key by the following computation:

$$(T_i^{send} = T_r^{recv}, T_i^{recv} = T_r^{send}) \leftarrow \mathsf{HKDF}_2(C_i = C_r, \epsilon)\,.$$

Afterwards, they set the counters for the upcoming transport data messages to zero, and securely erase the ephemeral keys and the chaining key. This erasure is necessary to guarantee forward secrecy, because using these values, the session key could be recovered.

Transport Data Messages. The transport data messages in WireGuard serve to exchange encrypted encapsulated packets. The responder is only allowed to send a transport data message *after* the initiator has sent its first transport data message. This first transport data message serves as key confirmation, and proves that the initiator has control over its ephemeral key; Intuitively, only then the initiator has sent a message that depends also on the responder's ephemeral key.

The inner plaintext packet is denoted by P and has length $|P|$ bytes. Both peers can send the following message `msg`:

`type` $\leftarrow 0x4$ (1 byte)	`reserved` $\leftarrow 0^3$ (3 bytes)
`receiver` $\leftarrow I_i$ (4 bytes)	
`counter` $\leftarrow N_m^{send}$ (8 bytes)	
`packet` ($\lvert\hat{P}\rvert$ bytes)	

The inner packet is first padded with zeros to a length that is a multiple of 16 bytes:

$$P \leftarrow P \| 0^{16 \cdot \lceil |P|/16 \rceil - |P|}$$

The encryption is as follows:

$$\texttt{msg.packet} \leftarrow \mathsf{AEAD}(T_m^{send}, N_m^{send}, P, \epsilon)$$

And finally, the counter gets incremented preparing for the next packet.

$$N_m^{send} \leftarrow N_m^{send} + 1$$

The recipient of the message decrypts using $T_{m'}^{recv}$ and $N_{m'}^{recv}$, and increments the counter after successful decryption. Besides being the nonce for the AEAD scheme, the counter also is needed to avoid replay attacks. As messages might arrive out of order over UDP, WireGuard employs a sliding window technique.

4. Proofs of Cryptographic Properties with CryptoVerif

4.1. Introduction to Proofs Based on Sequences of Games

In Chapter 2 we already defined both some assumptions (like the DDH assumption) and security properties (like IND-CPA) on the basis of games played between a *challenger* and an *adversary*. We further describe this approach based on a tutorial paper by Shoup [52]. He begins by saying that because the challenger and the adversary are probabilistic processes, the game can be modeled as a probability space. Games can expose oracles to the adversary, and that is how the challenger and the adversary communicate: Oracles receive an input, perform a possibly probabilistic computation, and return an output. These outputs are the random variables that the adversary sees from the probability space. An example are the encryption and verification oracles in the INT-CTXT game.

The definition of a security property is usually based on the probability with which certain events take place in the game. The difference between this probability and some, as Shoup calls it, target probability is called the *advantage* of the attacker to break the security property. Just to give one common example, the target property would be $1/2$ for guessing a bit. If the advantage is *negligible*, we say that the security property holds.

Security properties for "complex" protocols can also be expressed by a game. In Section 2.2 for example, we say that message secrecy is usually modeled as a left-or-right message indistinguishability game. We say this is more "complex" than the games for security properties like IND-CPA, because protocols might use more than one cryptographic primitive in a nested way. Thus, a proof of a protocol will be based on more than one cryptographic assumption, and generally the game considered will be larger.

A widely used technique to formulate security proofs is a sequence of games [52], which is sometimes called game hopping. Starting from the initial game, transformations are gradually made to the game, producing a sequence of games. For each transformation between two games, a bound needs to be specified for the probability that the adversary can distinguish the distribution of the two game's outputs. The sequence stops at a final game, where the probability that the adversary breaks the security property can be directly bounded (for example by 0 because it can be shown that it is impossible in this game). Starting from there, an overall probability can be calculated with which the adversary can distinguish the initial game and the final game. This probability is then a bound for the probability that the security property can be broken in the initial game.

Shoup describes three types of game transformations. **Transitions based on indistinguishability.** If an adversary could distinguish two games that differ by such a transformation, this would mean that the adversary could also distinguish two distributions that

are *indistinguishable* according to an assumption the proof is based on. The probability that the adversary distinguishes the two games has then a bound by the probability with which the assumption holds.

For example with IND-CPA, we know that an adversary cannot distinguish the oracle-provided ciphertexts of two chosen plaintexts. Supposing part of the game is an encryption of a plaintext possibly known by the attacker. We can then make a game transformation and replace this encryption by the encryption of a constant value of the same length (this is a plaintext definitely known by the attacker). We now have a bound for the probability that the adversary distinguishes the distribution of ciphertexts by the probability that the IND-CPA property does not hold for the encryption scheme: The adversary needs to distinguish the encryption of two equal length ciphertexts; and this is exactly how we defined IND-CPA.

Transitions based on failure events. Such a transition transforms a game into a game that has the same output distribution except if a failure event occurs. The probability that an adversary can distinguish the two games is the difference in probability with which the failure event occurs on both sides. This is backed by the following lemma.

Definition 30 (Difference Lemma [52])**.** Let A, B, F be events defined in some probability distribution, and suppose that $A \wedge \neg F \Leftrightarrow B \wedge \neg F$. Then $|\Pr[A] - \Pr[B]| \leq \Pr[F]$.

For an example of this type of transition, consider a game where a nonce-using AEAD scheme is used. We remind that according to the security definition we gave in Section 2.1, when a nonce would be reused, no guarantee is given about the security of the scheme by the IND-CPA or INT-CTXT property. Consider a second game that is the same as the previous but does not encrypt but abort with a failure event if a nonce is reused. The probability that an adversary can distinguish the games is thus the probability that a nonce is reused.

Bridging steps. These transformations do not change the output distributions, but serve only to prepare for transitions of the other two types. They do so by changing computations in an equivalent way, for example by reordering independent computations, or deleting unnecessary computations.

4.2. Introduction to CryptoVerif

CryptoVerif is a proof assistant that helps to write game-based proofs. More precisely, given the initial game and a sequence of game transformations, it automatically applies them and transforms the game accordingly, thus creating the series of games itself. A transformation is only done if it is applicable to the game in a cryptographically sound way. CryptoVerif also has an automatic mode where it can find which transformations to apply based on a built-in proof strategy. This way, simple protocols can be proven completely automatically. This is fundamentally different from EasyCrypt [3] where the user has to write all games manually and indicate why they are indistinguishable, and then verifies if the games are indeed indistinguishable using an SMT solver.

To prove concrete security properties, queries can be asked to CryptoVerif at each stage in the sequence of games. CryptoVerif supports secrecy and correspondence queries.

The former correspond directly to the secrecy properties introduced in Section 2.2.1, and the latter permit to prove correspondences between events, which serves to prove authentication-like properties.

For each transformation, CryptoVerif keeps track of the probability with which the adversary can distinguish the two games, and computes the final probability based on them. A successful proof and a final probability are reached when in a game, all queries can be answered positively. This is then the final game. As all transformations have a negligible probability in the security parameter of being detected, the same holds for the final probability. This means that when CryptoVerif concludes with a proof, an asymptotic proof for the queries has been found. The calculation of an exact probability for certain parameters and instantiations of the cryptographic primitives is left to the user. The probability formula given is precise enough to allow for this: It includes distinct probability functions for each primitive to be broken and also depends on the execution time.

Games in CryptoVerif are expressed in an applied pi calculus, and the tool comes with a comprehensive library of cryptographic primitives. Since version 2.00, CryptoVerif and ProVerif, a tool working in the symbolic model also by Bruno Blanchet, have become very close in syntax. It is now possible to write models in the common language, and run them in both tools. This makes the transition from one tool to the other much easier. The usual approach is to analyse a protocol in the symbolic model to find logical protocol issues. Tools working in the symbolic mode can find attack traces to understand these protocol flaws. Once logical problems are fixed, a cryptographic proof can be attempted. When using ProVerif 2.00 and CryptoVerif 2.00, this process becomes much more streamlined. We did not attempt a symbolic analysis in ProVerif first, because WireGuard already has a symbolic analysis done in Tamarin.

In the following, we describe CryptoVerif's syntax and semantics, and finally dive into the modelling of certain cryptographic primitives and game transformations.

4.2.1. Syntax and Semantics

Games in CryptoVerif are represented in a process calculus with probabilistic semantics. It is inspired by the pi-calculus and other calculi that were developed for the purpose of cryptographic proofs [12]. Process calculi are fitted for the use case of protocols because they permit to model their concurrent nature with parallel executions. Parallel execution is modeled via parallel composition of possibly different processes, for example $P_1|P_2$, and replication of one process, that is $!^N P$, which intuitively corresponds to an n times parallel composition of P. Interaction with the attacker is modeled via input and output channels. Messages are bitstrings in CryptoVerif's calculus, and functions are from bitstrings to bitstrings. Security assumptions that express the indistinguishability of two processes Q, Q' up to probability p are notated as $Q \approx_p Q'$.

A CryptoVerif input file consists of a list of declarations followed by a process:

$$\langle \text{declaration} \rangle^* \texttt{ process } \langle \text{iprocess} \rangle$$

The process describes the considered security protocol possibly embedded into a security game; the declarations contain type and function definitions and specify in particular

hypotheses on the cryptographic primitives and the security queries to prove. We give an overview of the syntax using Figure 4.1, and describe some language elements in more detail in the following. Most of them will be described only as soon as they are used within the code we describe later.

Types and the `new` keyword. The language is strongly typed, type conversions need to be done explicitely. Arbitrary type conversion functions can be defined. Costum types are simply defined with the `type` keyword and by giving them a name. Types correspond to sets of bitstrings or a special symbol $\perp$ (used for failed decryptions, for instance). Of cryptographic interest are the options that can be defined between brackets: fixed, large, and bounded are possible annotations. They determine from what probability distribution new random values are created with the `new` keyword. New variables are either created with the `new` keyword which corresponds to generation of random elements in CryptoVerif's semantics. Or they are created with the `let` keyword, which permits to assign the result of another term (variable, function, ...) to the variable.

Arrays. An extension of CryptoVerif's calculus over other calculi of particular interest is the automatic accessibility of variable values via arrays: All the values of a variable v in all replications of a process can be accessed and searched via array indices $v[i]$, where i is the replication index of the process in which the variable v has the value $v[i]$. This facilitates the formulation of games with lists that is common in cryptography, like when a list needs to be populated with all previously used nonces, and it needs to be checked if a new value collides with an old value.

Channel and Channel Names. The top-level process starts with an input process. This means the adversary starts the game by calling it. This can be used to give parameters to the game directly at the start. Input and output channels have names.

`out(`⟨channel⟩`,` ⟨term⟩`)[;` ⟨iprocess⟩`]`
`in(`⟨channel⟩`,` ⟨pattern⟩`)[;` ⟨oprocess⟩`]`

If input channels have the same name and the adversary calls an input channel of this name, each of the channels has uniform probability of being chosen. For our model this is not desirable, because we need deterministic communication. Note that input and output processes always alternate, and only input processes can be run in parallel.

4.2.2. How CryptoVerif Checks if Queries are Satisfied

Secrecy Queries. In standard key exchange protocols, we would query the secrecy of a key k with `query secret k`. CryptoVerif would then attempt to prove that the values of k in various sessions are indistinguishable from independent random numbers. We already discussed why key secrecy is not meaningful in the WireGuard protocol, because the key is used in the protocol. In our model instead, the only secrecy query is the one about a global bit b that is used in left-or-right message indistinguishability games. That means, b is used in the game, and outputs of oracles depend on it. For such situations,

$M, N ::=$	terms
i	replication index
$x[M_1, \ldots, M_m]$	variable access
$f(M_1, \ldots, M_m)$	function application
new $x[\widetilde{i}] : T; N$	random number
let $p = M$ in N else N'	assignment (pattern-matching)
let $x[\widetilde{i}] : T = M$ in N	assignment
if defined$(M_1, \ldots, M_l) \wedge M$ then N else N'	conditional
find$[unique?]$ $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M'_j$ then $N_j)$ else N'	array lookup
insert $Tbl(M_1, \ldots, M_l); N$	insert in table
get $Tbl(p_1, \ldots, p_l)$ suchthat M in N else N'	get from table
event $e(M_1, \ldots, M_l); N$	event
event$_a$bort e	event e and abort
$p ::=$	pattern
$x[\widetilde{i}] : T$	variable
$f(p_1, \ldots, p_m)$	function application
$=M$	comparison with a term
$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq n} Q$	replication n times
newChannel $c; Q$	channel restriction
$c[M_1, \ldots, M_l](p); P$	input
$P ::=$	output process
$\overline{c[M_1, \ldots, M_l]}\langle N \rangle; Q$	output
new $x[\widetilde{i}] : T; P$	random number
let $p = M$ in P else P'	assignment
if defined$(M_1, \ldots, M_l) \wedge M$ then P else P'	conditional
find$[unique?]$ $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] = i_{jm_j} \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P	array lookup
insert $Tbl(M_1, \ldots, M_l); P$	insert in table
get $Tbl(p_1, \ldots, p_l)$ suchthat M in P else P'	get from table
event $e(M_1, \ldots, M_l); P$	event
event$_a$bort e	event e and abort
yield	end

Figure 4.1.: Syntax of the process calculus [12]

CryptoVerif's approach is to eliminate the usage of b. In automatic mode, it would use the built-in proof strategy to do so, but this is not possible in our model (because of key compromises that are not handled automatically). In interactive mode, CryptoVerif would point out the occurrence in the game, where an output depends on b. It is then up to the user to make cryptographic transformations to make b disappear. Thus, to make it short: Checking the secrecy query in our model means checking that no oracle output depends on b.

Correspondence Queries. Queries of this type permit to prove correspondence properties between events. We use three slightly different variants in our model of WireGuard. First, queries like "if an event A has been executed, then an event B has be executed before". These are non-injective correspondences. We use such a query to prove that the first protocol message cannot be forged if no key was compromised. However it can be replayed. Second, queries like "if an event A has been executed, then for each occurrence of the event A, there is at least one occurrence of an event B". These are called injective correspondences and permit to prove authentication properties (and in comparison with the previous example, that replay is not possible). Third, queries like "if an event A and an event B have been executed, then some equation holds". We use these to prove correctness of the protocol and resistance against UKS: If two parties have the same view on a protocol transcript, then they calculate the same key; if two parties calculate the same key, then they have the same view on the protocol transcript (the protocol transcript of course defined according to the definition of partnering).

These three are only motivational examples. In fact, a non-injective correspondence is an implication between two logical formulae $\psi \Rightarrow \phi$, and these formulae can contain events. The grammar for logical formulae ϕ is the following [11]:

ϕ ::=		formula
	M	term
	event($e(M_1, \ldots, M_m)$)	event
	$\phi_1 \vee \phi_2$	conjunction
	$\phi_1 \wedge \phi_2$	disjunction .

The formula M holds if the term M evaluates to true, the formula event($e(M_1, \ldots, M_m)$) holds if the event has been executed, and conjunction and disjunction are defined as usual. For injective correspondences, the grammar for logical formula is extended with inj-event($e(M_1, \ldots, M_m)$). The left-hand formula ψ is then a conjunction of injective or non-injective events.

The algorithm to check correspondences is much more involved than the one that checks secrecy. We only present the rough idea and refer to Blanchet's publication on correspondence assertions in CryptoVerif for more details [11]. For all programme points in a model, CryptoVerif collects true facts. That is, the value a variable is set to, the fact that a variable is defined, and the fact that an event has been executed. For injective events, CryptoVerif also collects information on the replication indices of the events. To prove a non-injective correspondence $\psi \Rightarrow \phi$, CryptoVerif collects all facts that hold at programme

points of events in ψ and shows that these facts imply ϕ using an equational prover. For injective correspondences, it is shown that if the replication indices of two executions of injective events in ψ are different, then the replication indices of the corresponding executions of the considered injective event of ϕ are also different [11].

Events for which all correspondence queries have been proved will be deleted from the model with the next run of the `simplify` command. This is important because in some cases, the arguments of events can are considered as "usage" of a variable (i.e. a key that is used, but needed for a real-or-random indistinguishability game) and can thus prevent the proof of a secrecy query. As a result, before attempting to prove a secrecy query, it is advised to prove all correspondence queries.

4.3. A Security Model for WireGuard in CryptoVerif

We create a model for the WireGuard protocol in CryptoVerif and prove secrecy and correspondence properties for it. We 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.

For secrecy properties, we integrate left-or-right message indistinguishability games into the protocol. We do so at each transport data message, including the key confirmation message. For correspondence properties, we include events at important places in the protocol and issue queries on them.

4.3.1. Modelling the Cryptographic Primitives

CryptoVerif provides a macro system to instantiate definitions from the included cryptographic library. This permits one to adapt type, variable, function, and probability names, in case one definition (of a cryptographic primitive) needs to be used multiple times. A macro for a cryptograpic primitive defines types, functions, equations, and equivalences. The equations permit to model properties and relations of the primitive, like calculation with group elements, or relations like correctness between encryption and decryption functions. The equivalences define how CryptoVerif can transform the game if this primitive is used. In the following we describe what primitives we instantiate for our model and elaborate on some of the functions and equivalences.

The AEAD Encryption Scheme. We define types, constants, type conversion functions and probabilities for use with the `AEAD_nonce` macro:

```
1  type key_t [large,fixed].
2  const dummy_key: key_t.
3  fun key_to_bitstring(key_t): bitstring [data].
4  type psk_t [large,fixed].         (* 32 byte pre-shared symmetric key *)
5  const psk_0: psk_t.                (* pre-shared key with all zeros,  *)
6                                     (* used in case the WireGuard user *)
7                                     (* did not provide a psk.          *)
```

```
type nonce_t [large,fixed].        (* 12 byte counter nonce for AEAD. *)
const nonce_0: nonce_t.            (* const value for the zero nonce  *)
const empty_bitstring : bitstring. (* const value for the empty
                                      bitstring that will be encrypted *)
const dummy_bitstring: bitstring.
proba P_enc.
proba P_encctxt.
expand AEAD_nonce(
  (* types *)
  key_t,     (* keys *)
  bitstring, (* plaintext *)
  bitstring, (* ciphertext *)
  bitstring, (* additional data *)
  nonce_t,   (* nonces *)
  (* functions *)
  enc,     (* encryption:
              (* enc(plaintext, additional data, key, nonce): ciphertext *)
  dec,  (* decryption:
           (* dec(ciphertext, additional data, key, nonce): bitstringbot *)
  injbot,    (* injection from plaintext to bitstringbot:
                (* injbot(plaintext): bitstringbot *)
  Zero,      (* returns a plaintext of same length, consisting of zeros:
                (* Zero(plaintext): plaintext *)
  (* probabilities *)
  P_enc,     (* breaking IND-CPA *)
  P_encctxt  (* breaking INT-CTXT *)
).
```

The macro defines the following parameters, functions, equation between functions, and events:

```
def AEAD_nonce(key, cleartext, ciphertext, add_data, nonce, enc, dec, injbot,↩
     Z, Penc, Pencctxt) {

param N, N2, N3.

fun enc(cleartext, add_data, key, nonce): ciphertext.
fun dec(ciphertext, add_data, key, nonce): bitstringbot.

fun enc'(cleartext, add_data, key, nonce): ciphertext.

fun injbot(cleartext):bitstringbot [compos].
equation forall x:cleartext; injbot(x) <> bottom.

(* The function Z returns for each bitstring, a bitstring
   of the same length, consisting only of zeroes. *)
fun Z(cleartext):cleartext.

```

```
17 equation forall m:cleartext, d: add_data, k:key, n: nonce;
18     dec(enc(m, d, k, n), d, k, n) = injbot(m).
19
20 (* Event raised when some nonce is used several times
21    with the same key, which breaks security. *)
22 event repeated_nonce.
```

The function `enc'` which has the exact same signature as `enc` permits to detect which occurrences of the encryption function have already been treated by the cryptographic equivalences. In lines 10 and 11, the error symbol `bottom` is defined. It is returned by the decryption function is case of failed authentication. The function `injbot` converts between the space of bitstrings to the space of bitstrings including the error symbol. It can be used in a pattern matching expression on a decryption result. If the pattern matching fails, this means that the decryption's result was `bottom`. The function `Z` models the leakage of the length of the plaintext according to the definition of IND-CPA; `Z(x)` is a bitstring of the same length as `x`, containing only zeros. The equation starting in line 17 defines the correctness of the encryption scheme. The event `repeated_nonce` is added automatically by CryptoVerif and will be successfully proved to not occur (in our case). We will shortly comment the definitions of the IND-CPA and INT-CTXT security properties.

```
1  (* IND-CPA *)
2  equiv(ind_cpa(enc))
3          foreach i2 <= N2 do k <-R key;
4                  foreach i <= N do Oenc(x:cleartext, d: add_data, n: nonce) :=
5                      return(enc(x, d, k, n))
6        <=(N2 * Penc(time + (N2-1)*(N*time(enc, maxlength(x), maxlength(d), ↩
             maxlength(n)) + N*time(Z, maxlength(x))), N, maxlength(x)))=>
7          foreach i2 <= N2 do k <-R key;
8                  foreach i <= N do Oenc(x:cleartext, d: add_data, n: nonce) :=
9                  find u <= N suchthat defined(x[u],d[u],n[u],r[u])
10                                 && n = n[u] && (x <> x[u] || d <> d[u]) then
11                         event_abort repeated_nonce
12                     else
13                     let r: ciphertext = enc'(Z(x), d, k, n) in
14                 return(r).
```

Line 6 separates two processes, that are indistinguishable with the probability defined between `<=(` and `)=>`. If the equivalence is applicable, the process above `<=(...)=>` can be replaced by the process below `<=(...)=>`, and the probability will be used as the probability that an adversary distinguishes the two games. CryptoVerif will instantiate the equivalence with the according parameters and variables. The outer loop models encryption with different keys, the inner loop encryption under the same key. The parameters `N2` und `N` appear in the probability formula. The equivalence replaces an encryption by the `find` condition starting in line 10: If another encryption under the same key, with the same nonce is found, the failure event `repeated_nonce` is called and the game aborted. Otherwise, the encryption is done, but the plaintext is not encrypted but the function `Z` of the plaintext. This models the leakage of the length of the plaintext. More formally, the `find` condition looks for a replication index `u`, for which the variables `x`, `d`, `n`, and `r` are defined, the current

nonce is the same, but plaintext or ciphertext are different. Note that the encryption function is replaced by `enc'` to prevent the equivalence from being applied multiple times.

The INT-CTXT assumption is modeled as follows:

```
1  (* INT-CTXT *)
2  equiv(int_ctxt(enc))
3    foreach i2 <= N2 do k <-R key; (
4      foreach i <= N do Oenc(x:cleartext, d: add_data, n: nonce) :=
5        return(enc(x, d, k, n)) |
6      foreach i3 <= N3 do Odec(y:ciphertext, c_d: add_data, c_n: nonce) :=
7        return(dec(y,c_d,k,c_n)))
8    <=(N2 * Pencctxt(time + (N2-1)*(N*time(enc, maxlength(x), maxlength(d), ↩
         maxlength(n)) + N3*time(dec,maxlength(y),maxlength(c_d),maxlength(c_n))↩
         ), N, N3, maxlength(x), maxlength(y), maxlength(d), maxlength(c_d)))=> ↩
         [computational]
9    foreach i2 <= N2 do k <-R key [unchanged]; (
10     foreach i <= N do Oenc(x:cleartext, d: add_data, n: nonce) :=
11       find u <= N suchthat defined(x[u],d[u],n[u],r[u])
12                && n = n[u] && (x <> x[u] || d <> d[u]) then
13         event_abort repeated_nonce
14       else
15         let r: ciphertext = enc(x, d, k, n) in
16     return(r) |
17     foreach i3 <= N3 do Odec(y:ciphertext, c_d: add_data, c_n: nonce) :=
18       find j <= N suchthat defined(x[j],d[j],n[j],r[j]) &&
19              r[j] = y && d[j] = c_d && n[j] = c_n then
20         return(injbot(x[j]))
21       else
22         return(bottom)).
```

This time, two oracles are defined on both sides: An encryption and a decryption oracle. The transformation of the encryption is not the interesting point, just the failure event in case of nonce reuse is added. The transformation of the decryption is important here: It is replaced by a `find` condition that looks for the replication index of an encryption oracle that returned the ciphertext in question while called with the given nonce, key and additional data. If such an encryption call is found, the plaintext used there is returned. Otherwise, the error symbol `bottom` is returned. This models that the adversary cannot forge a ciphertext: Decryption must fail if the ciphertext was not produced by the encryption function.

The key `k` is modeled as chosen randomly in line 3, and it is used in the encryption and decryption function modeled below. If the key is used anywhere else in the game, the equivalence does not match and cannot be applied. This is a problem in our model, because we allow dynamic compromise of keys. We show later how this is modeled exactly, for now it is enough to know that the key `k` appears in an `out` channel somewhere else in the game, which the attacker may have called or not. Of course, in the case the key *is* compromised, it is useless to apply any security assumption on the encryption scheme. But in case the attacker did not call the corruption oracle, we want to be able to apply the

game transformations. Therefore, there is another definition of INT-CTXT that can be applied in such a scenario:

```
equiv(int_ctxt_corrupt(enc))
  foreach i2 <= N2 do k <-R key; (
    foreach i <= N do Oenc(x:cleartext, d: add_data, n: nonce) :=
      return(enc(x, d, k, n)) |
    foreach i3 <= N3 do Odec(y:ciphertext, c_d: add_data, c_n: nonce) [↩
        useful_change] :=
      return(dec(y,c_d,k,c_n)) |
    Ocorrupt() [10] := return(k))
  <=(N2 * Pencctxt(time + (N2-1)*(N*time(enc, maxlength(x), maxlength(d), ↩
      maxlength(n)) + N3*time(dec,maxlength(y),maxlength(c_d),maxlength(c_n))↩
      ), N, N3, maxlength(x), maxlength(y), maxlength(d), maxlength(c_d)))=> ↩
      [manual,computational]
  foreach i2 <= N2 do k <-R key [unchanged]; (
    foreach i <= N do Oenc(x:cleartext, d: add_data, n: nonce) :=
      find u <= N suchthat defined(x[u],d[u],n[u],r[u])
             && n = n[u] && (x <> x[u] || d <> d[u]) then
        event_abort repeated_nonce
      else
        let r: ciphertext = enc(x, d, k, n) in
        return(r) |
    foreach i3 <= N3 do Odec(y:ciphertext, c_d: add_data, c_n: nonce) :=
      if defined(corrupt) then return(dec(y,c_d,k,c_n)) else
      find j <= N suchthat defined(x[j],d[j],n[j],r[j]) &&
           r[j] = y && d[j] = c_d && n[j] = c_n then
        return(injbot(x[j]))
      else
        return(bottom) |
     Ocorrupt() := let corrupt: bool = true in return(k)).
}
```

In this equivalence, besides encryption and decryption oracles, there is a third oracle that simply publishes the key. This models that the attacker can have knowledge of the key in some way. In the transformed game, we change the corruption oracle to define a boolean variable `corrupt`. The decryption oracle can then proceed as in the standard INT-CTXT, if this boolean is not defined. If it is defined, this means the key has been compromised and then the decryption is not transformed.

We define several convenience wrapper functions around encryption and decryption to make the code of the protocol messages easier to read:

```
letfun enc_G(group_element: G_t, current_hash: hashoutput_t, k: key_t, n: ↩
    nonce_t) =
  enc(G_to_bitstring(group_element), hashoutput_to_bitstring(current_hash), k↩
      , n).

```

```
letfun dec_ad_hash(ciphertext: bitstring, current_hash: hashoutput_t, k: key_t, n: nonce_t) =
  dec(ciphertext, hashoutput_to_bitstring(current_hash), k, n).

letfun enc_timestamp(timestamp: timestamp_t, current_hash: hashoutput_t, k: key_t, n: nonce_t) =
  enc(timestamp_to_bitstring(timestamp), hashoutput_to_bitstring(current_hash), k, n).

letfun enc_bitstring(plaintext: bitstring, current_hash: hashoutput_t, k: key_t, n: nonce_t) =
  enc(plaintext, hashoutput_to_bitstring(current_hash), k, n).
```

In WireGuard transport data messages, a counter value is transmitted, that is prepended with zeros to build the nonce, as described in Section 3.2. We therefore define an according type, a type conversion function and a relationship between the zeros in both types.

```
type counter_t [fixed].     (* 8 byte counter in the data message *)
const counter_0: counter_t. (* constant for counter with value 0  *)
fun nonce_to_counter(nonce_t) : counter_t [data].
  (* This is [data] because WireGuard enforces a new handshake before *)
  (* the counter would overflow. So basically we have a bijection
  (* between counter and nonce. *)

equation nonce_to_counter(nonce_0) = counter_0.
```

Gap Diffie-Hellman. Following the Noise specification, the Gap Diffie-Hellman assumption must hold for the group used for the Diffie-Hellman operations. Thus we instantiate Gap Diffie-Hellman from the library as follows:

```
type G_t [bounded,large].  (* type of group elements (must be "bounded"
                              and "large", of cardinal a prime q) *)
const dummy_g: G_t.      (* return value in letfuns in case of errors *)
fun G_to_bitstring(G_t): bitstring [data].
type Z_t [bounded,large].   (* type of exponents (must be "bounded" and
                               "large", supposed to be {1, ..., q-1}) *)
const dummy_z: Z_t.      (* return value in letfuns in case of errors *)
proba P_GDH.            (* probability of breaking the GDH assumption *)
(* Page 7 in the Noise paper, rev 33:
   The public_key either encodes some value in a large prime-order group
   (which may have multiple equivalent encodings), or is an invalid
   value. *)
expand GDH_prime_order_all_args(
  (* types *)
  G_t,  (* Group elements *)
  Z_t,  (* Exponents *)
  (* variables *)
  g,    (* a generator of the group *)
```

```
19    exp,  (* exponentiation function *)
20    exp', (* a symbol that replaces exp after game transformation *)
21    mult, (* multiplication function for exponents *)
22    (* probabilities *)
23    P_GDH (* probability of breaking the GDH assumption *)
24  ).
25  letfun DH(group_element: G_t, exponent: Z_t) =
26    exp(group_element, exponent).
```

We define types for group elements and exponents. For both types, we define constants that serve as dummy return type. The probability `P_GDH` will appear in the final probability formula as the probability that the Gap Diffie-Hellman assumption is broken by any adversary. The macro in CryptoVerif's library defines for example the following functions, constants, and equations for Gap Diffie-Hellman (this is just an excerpt):

```
1  fun exp(G,Z): G.
2  fun exp'(G,Z): G.
3  const g:G.
4  fun mult(Z,Z): Z.
5  equation builtin commut(mult).
6  equation forall a:G, x:Z, y:Z;
7    exp(exp(a,x), y) = exp(a, mult(x,y)).
8  equation forall a:G, x:Z, y:Z;
9    exp'(exp'(a,x), y) = exp'(a, mult(x,y)).
```

This defines what calculations can be made in the group. The macro will replace `Z` by `Z_t` and `G` by `G_t`. The equivalences defined for Gap Diffie-Hellman, which we do not reproduce here, serve CryptoVerif to match DDH oracles and possible corruptions of private keys, and to transform the game accordingly.

Message Authentication Code. WireGuard does not define which security property is required for the MAC. We chose to model it as a SUF-CMA secure, deterministic MAC. However, our proof does not rely on the security of the MAC, that is, does not use any security assumption on the MAC. Thus we only use the definition of the two functions `mac` and `verify` from the macro and the correctness equations:

```
1  fun mac(bitstring, hashoutput_t): mac_t.
2  fun check(bitstring, hashoutput_t, mac_t): bool.
3
4  equation forall m: bitstring, k: hashoutput_t;
5      check(m, k, mac(m, k)).
6
7  equation forall m: bitstring, k: hashoutput_t, m': mac_t;
8      (mac(m,k) = m') = check(m, k, m').
```

The first MAC in WireGuard is calculated over all bytes of the message prior to `mac1`. Therefore we define functions that map all prior protocol fields to a bitstring, one for the first protocol message and one for the second protocol message:

```
1  fun concat_msg_alpha_1(msg_type_t, reserved_t,
```

```
2                       session_index_t, G_t,
3                       (* and the two ciphertexts *)
4                       bitstring, bitstring): bitstring [data].
5   fun concat_msg_alpha_2(msg_type_t, reserved_t,
6                       session_index_t, session_index_t,
7                       G_t, bitstring): bitstring [data].
```

Key Derivation Function. WireGuard uses HKDF in a chain of calls to derive symmetric keys at different stages of the protocol:

$$
\begin{aligned}
C_i &= \text{const} \\
C_i &= \mathsf{HKDF}_1(C_i, v_0) \\
C_i \| \kappa_1 &= \mathsf{HKDF}_2(C_i, v_1) \\
C_i \| \kappa_2 &= \mathsf{HKDF}_2(C_i, v_2) \\
C_i &= \mathsf{HKDF}_1(C_i, v_3) \\
C_i &= \mathsf{HKDF}_1(C_i, v_4) \\
C_i &= \mathsf{HKDF}_1(C_i, v_5) \\
C_i \| \tau \| \kappa_3 &= \mathsf{HKDF}_3(C_i, v_6) \\
T_i^{\text{send}} \| T_i^{\text{recv}} &= \mathsf{HKDF}_2(C_i, v_7)
\end{aligned}
$$

We want to model HKDF as a random oracle, as there are existing indifferentiability results we can built upon to justify this. However, using 8 calls to a random oracle, chained by the chaining key C_i, will vastly increase the size of the games CryptoVerif produces. To understand this, consider the definition of a random oracle hash in CryptoVerif's library:

```
1   def ROM_hash(key, hashinput, hashoutput, hash, hashoracle, qH) {
2
3   param Nh, N, Neq.
4
5   fun hash(key, hashinput):hashoutput.
6
7   equiv(rom(hash))
8         foreach ih <= Nh do k <-R key;
9           (foreach i <= N do OH(x:hashinput) := return(hash(k,x)) |
10           foreach ieq <= Neq do Oeq(x':hashinput, r':hashoutput) := return(r' ↩
                 = hash(k,x')))
11         <=(#Oeq / |hashoutput|)=> [computational]
12        foreach ih <= Nh do
13          (foreach i <= N do OH(x:hashinput) :=
14         find[unique] u <= N suchthat defined(x[u],r[u]) && x= x[u] then return↩
               (r[u]) else r <-R hashoutput; return(r) |
15           foreach ieq <= Neq do Oeq(x':hashinput, r':hashoutput) :=
16         find[unique] u <= N suchthat defined(x[u],r[u]) && x' = x[u] then ↩
               return(r' = r[u]) else
17         return(false)).
```

On top, we have two oracles, one that calculates a hash, and one that compares a variable `r'` to a hash result `hash(k, x')`. On the bottom, both oracles are replaced by a `find` construction. The first in line 14 returns a previous result if the oracle was called with the same arguments, and otherwise returns a fresh random value drawn from the result type. The second one in line 16 compares the variable `r'` to the previous hash result, if the hash function was called with the arguments `k, x'` before. If the hash function was not called with the arguments before, it returns false. This models that an attacker cannot know the result of a random oracle hash function if the random oracle was not called for these arguments before. This game transformation models exactly the definition of a random oracle that we gave in Section 2.1. The probability to distinguish the two games is the number of calls the attacker makes to the hash oracle divided by the output space. This is the case because in the above game, the comparison could return true if the attacker found a collision, while collisions are excluded in the bottom game.

If a random oracle is called in a game, then the transformation will replace all random oracle function calls by a `find` construction comparing the arguments with the arguments of *all other* calls of the same random oracle, leading to 8 branches in the game at each call of a random oracle call. The calls will thus become nested, and the number of `find` conditions could increase to 8^8, which exceeds by far the capabilities of CryptoVerif on current computers. Thus, we need to simplify the protocol. We observe, that keys are only extracted in 4 of the HKDF calls. Additionally, the last HKDF call does not take any new input, meaning that the transport data keys depend only on the same inputs as the intermediate symmetric key before. Thus, we replace the chain of HKDF calls by three calls to independent random oracles. We keep track of the dependency of the extracted keys of all inputs by giving all those inputs as arguments to the random oracles:

$$\begin{aligned}
\kappa_1 &= \mathsf{Chain}'_1(v_0, v_1)\\
\kappa_2 &= \mathsf{Chain}'_2(v_0, v_1, v_2)\\
\tau \| \kappa_3 \| T_i^{\text{send}} \| T_i^{\text{recv}} &= \mathsf{Chain}'_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)
\end{aligned}$$

We give justification of this approach with an indifferentiability proof. Readers who want to skip this proof can continue reading on page 55.

4.3.1.1. Indifferentiability Proof for the Key Derivation Chain

Indifferentiability is a more general notion of indistinguishability. In the way we use indifferentiability for the following proofs, it has been defined in [20]. Like [39], we instantiate it for random oracles as the ideal primitive. However, because we need to show indifferentiability for more than one function, we extend the definition.

Definition 31 (Indifferentiability)**.** Functions $F_{i,1\le i\le n}$ with oracle access to a random oracle H are $(t_D, t_S, (q_H, q_{F_i,1\le i\le n}), \epsilon)$-indifferentiable from independent random oracles $H'_{i,1\le i\le n}$, if there exists a simulator S such that for any distinguisher D

$$|\Pr[D^{F_{i,1\le i\le n},H} = 1] - \Pr[D^{H'_{i,1\le i\le n},S} = 1]| \le \epsilon\,.$$

The simulator S has oracle access to $H'_{i,1\le i\le n}$, and runs in time t_S. The distinguisher D runs in time t_D and makes at most q_H queries to H and at most q_i queries to $F_{i,1\le i\le n}$.

In the game $G_0 = D^{F_{i,1\leq i\leq n},H}$, the distinguisher interacts with the real functions F_i and the random oracle H from which F_i are defined. In the game $G_1 = D^{H'_{i,1\leq i\leq n},S}$, the distinguisher interacts with a random oracle H'_i instead of F_i, and with a simulator S, which simulates the behavior of the random oracle H using calls to H'_i. Indifferentiability means that these two games are indistinguishable.

Intuitively, the indifferentiability proofs we do in the following work as follows: For all traces in G_0 we need to find the corresponding trace in G_1 and show that they produce the same result. Interesting traces are those where the distinguisher already knows the result from another related call.

This is done by proving the following steps:

- HKDF_n is indifferentiable from a random oracle. A proof for $n = 2$ has been done in [39], and we generalise it here.

- In a chain of calls to HKDF_n, every call is indifferentiable from a call to an independent random oracle that, additionally to the arguments of the corresponding call to HKDF_n, receives all prior arguments to HKDF_n. This also holds for the last call of HKDF_n in the chain.

- The last two calls to HKDF_n respectively Chain are indifferentiable from one call to a random oracle that has a longer output.

Before diving into the proofs themselves, we describe how the lemmas will be instantiated for use in WireGuard. For this, we introduce a notation for the concatenation of bitstrings used in the whole section. The $\|$ operator is used to concatenate blocks of 256 bit and multiples of it. We use the placeholder _ for one 256 bit block and $\ldots$ for multiple 256 bit blocks. In $x\|y = z$ and $x\|\_ = z$ and $x\|\ldots = z$, x consists of the first 256 bit block of z. In $x\|y = z$ and $\_\|y = z$ and $\ldots\|y = z$, y consists of the last 256 bit block of z. In $x_1\|\ldots\|x_i\|\ldots\|x_n = z$, x_1 consists of the first, x_i of i-th and x_n of the n-th 256 bit block of z.

In the original protocol, the chain of HKDF calls is the following. The constant value that initialises C_i is calculated by hashing the Noise protocol name.

$$
\begin{aligned}
C_i &= \mathsf{const} \\
C_i &= \mathsf{HKDF}_1(C_i, v_0) \\
C_i\|\kappa_1 &= \mathsf{HKDF}_2(C_i, v_1) \\
C_i\|\kappa_2 &= \mathsf{HKDF}_2(C_i, v_2) \\
C_i &= \mathsf{HKDF}_1(C_i, v_3) \\
C_i &= \mathsf{HKDF}_1(C_i, v_4) \\
C_i &= \mathsf{HKDF}_1(C_i, v_5) \\
C_i\|\tau\|\kappa_3 &= \mathsf{HKDF}_3(C_i, v_6) \\
T_i^{\mathrm{send}}\|T_i^{\mathrm{recv}} &= \mathsf{HKDF}_2(C_i, v_7)
\end{aligned}
$$

We prove Lemma 2 for a chain of calls to the same HKDF_n function, thus we rewrite the chain to use only calls to HKDF_3, as 3 is the maximum number of outputs needed.

$$
\begin{aligned}
C_i &= \mathsf{const} \\
C_i \| \_ \| \_ &= \mathsf{HKDF}_3(C_i, v_0) \\
C_i \| \kappa_1 \| \_ &= \mathsf{HKDF}_3(C_i, v_1) \\
C_i \| \kappa_2 \| \_ &= \mathsf{HKDF}_3(C_i, v_2) \\
C_i \| \_ \| \_ &= \mathsf{HKDF}_3(C_i, v_3) \\
C_i \| \_ \| \_ &= \mathsf{HKDF}_3(C_i, v_4) \\
C_i \| \_ \| \_ &= \mathsf{HKDF}_3(C_i, v_5) \\
C_i \| \tau \| \kappa_3 &= \mathsf{HKDF}_3(C_i, v_6) \\
T_i^{\mathrm{send}} \| T_i^{\mathrm{recv}} \| \_ &= \mathsf{HKDF}_3(C_i, v_7)
\end{aligned}
$$

Because of the way HKDF is constructed, this is actually the same computation. We just throw away parts of HKDF's result at places denoted with an underscore.

At this stage we can apply Lemma 2 which adds a negligible probability for an attacker to distinguish the two games. Note how every Chain call depends on all previous arguments.

$$
\begin{aligned}
\_ \| \_ &= \mathsf{Chain}_0(v_0) \\
\kappa_1 \| \_ &= \mathsf{Chain}_1(v_0, v_1) \\
\kappa_2 \| \_ &= \mathsf{Chain}_2(v_0, v_1, v_2) \\
\_ \| \_ &= \mathsf{Chain}_3(v_0, v_1, v_2, v_3) \\
\_ \| \_ &= \mathsf{Chain}_4(v_0, v_1, v_2, v_3, v_4) \\
\_ \| \_ &= \mathsf{Chain}_5(v_0, v_1, v_2, v_3, v_4, v_5) \\
\tau \| \kappa_3 &= \mathsf{Chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \\
T_i^{\mathrm{send}} \| T_i^{\mathrm{recv}} \| \_ &= \mathsf{Chain}_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{aligned}
$$

The last Chain call has a third output block that is not used – this will become clear when we formally define the HKDF chain later.

We now have established that all these random oracles are independent. It is therefore sufficient to use the following four lines to calculate the relevant parts of the HKDF chain:

$$
\begin{aligned}
\kappa_1 \| \_ &= \mathsf{Chain}_1(v_0, v_1) \\
\kappa_2 \| \_ &= \mathsf{Chain}_2(v_0, v_1, v_2) \\
\tau \| \kappa_3 &= \mathsf{Chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \\
T_i^{\mathrm{send}} \| T_i^{\mathrm{recv}} \| \_ &= \mathsf{Chain}_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{aligned}
$$

The output of the random oracles can be truncated by Lemma 3 to avoid having to throw away parts of the output:

$$
\begin{aligned}
\kappa_1 &= \mathsf{Chain}'_1(v_0, v_1) \\
\kappa_2 &= \mathsf{Chain}'_2(v_0, v_1, v_2) \\
\tau \| \kappa_3 &= \mathsf{Chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \\
T_i^{\mathrm{send}} \| T_i^{\mathrm{recv}} &= \mathsf{Chain}'_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{aligned}
$$

In WireGuard, $v_7 = \epsilon$. This means that T_i^{send} and T_i^{recv} only depend on $v_0, \ldots, v_6$, as do τ and κ_3 on the previous line. By Lemma 4 we can replace the last two lines by only one call to a random oracle with a longer output:

$$\begin{aligned} \kappa_1 &= \mathsf{Chain}'_1(v_0, v_1) \\ \kappa_2 &= \mathsf{Chain}'_2(v_0, v_1, v_2) \\ \tau \| \kappa_3 \| T_i^{\text{send}} \| T_i^{\text{recv}} &= \mathsf{Chain}'_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \end{aligned}$$

HKDF_n is indifferentiable from a random oracle WireGuard employs BLAKE2s [1] as hash function in HMAC and thus also in HKDF. BLAKE2 has a keyed mode of operation, which could be used directly instead of using the standard hash operation mode inside of the HMAC construction. As WireGuard is built upon the Noise Protocol Framework, it sticks to the approach chosen there, and this is to use HMAC for all hash functions, even for those which provide a more specialised function. The reasons are explained in great detail in [49] – mainly it is to have a common interface, and to be able to keep results from cryptographic analyses applicable that are based on the HMAC and HKDF construction.

BLAKE2's authors claim in their publication that an indifferentiability result [17] for BLAKE2's ancestor BLAKE can be inherited by BLAKE2. This is not the case, as stated by the authors of [46], who give a proof of indifferentiability from a random oracle of the entire BLAKE2 construction. With Theorem 4.3 from [26, 25, this theorem is only in the full version], we have indifferentiability of HMAC-BLAKE2s from a random oracle. Note that we cannot apply Theorem 4.4 from this paper because BLAKE2 is not a Merkle-Damgård construction.

The HKDF key derivation function is defined as follows in [40]:

$$\begin{aligned} &\mathsf{HKDF}_n(salt, key, info) = k_1 \| \ldots \| k_n \text{ where} \\ &\quad prk = \mathsf{HMAC}(salt, key) \\ &\quad k_1 = \mathsf{HMAC}(prk, info \| 0x00) \\ &\quad k_{i+1} = \mathsf{HMAC}(prk, k_i \| info \| i), \qquad \text{with } 1 \le i < n \end{aligned} \tag{4.1}$$

The variable i is a 1 byte value, and thus HKDF_n can return up to 256 blocks. This function 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_n rather than HMAC. In more detail, let

$$\begin{aligned} prk \| \ldots &= \mathsf{HKDF}_n(s, k, info) \\ salt &= \mathsf{HMAC}(s, k) \\ x &= \mathsf{HMAC}(prk, info' \| 0x00) \\ x' \| \ldots &= \mathsf{HKDF}_n(salt, info \| 0x00, info') \,. \end{aligned}$$

When HKDF_n is defined from HMAC as above, we have

$$\begin{aligned} prk' &= \mathsf{HMAC}(s, k) \\ prk = k'_1 &= \mathsf{HMAC}(prk', info \| 0x00) \end{aligned}$$

but with

$$prk' = salt\ ,$$

we have

$$prk = \mathsf{HMAC}(salt, info \| 0x00) \tag{4.2}$$

The second call to HKDF_n will calculate

$$\begin{aligned} prk'' &= \mathsf{HMAC}(salt, info \| 0x00) = prk \\ k_1'' &= \mathsf{HMAC}(prk'', info' \| 0x00) \end{aligned}$$

and thus,

$$x' = \mathsf{HMAC}(prk, info' \| 0x00) = x\ .$$

However, when HKDF_n is a random oracle and HMAC is defined from HKDF_n, the simulator that computes HMAC sees two calls to HMAC that seem to be unrelated. It is unable to see that *prk* is in fact related to the previous call $salt = \mathsf{HMAC}(s, k)$ by equation (4.2), because it doesn't 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.

We can however recover the indifferentiability of HKDF_n under the additional assumption that the $n + 1$ calls to HMAC use disjoint domains, and we will later justify that this assumption holds in our case. Let $\mathcal{S}$, $\mathcal{K}$, and $\mathcal{I}$ be the sets of possible values of *salt*, *key*, and *info* respectively, and $\mathcal{M}$ the set of 256 bit bitstrings, output of HMAC.

Lemma 1. Let $n \geq 2$ be a fixed integer. If $\mathcal{K} \cap (\mathcal{I} \| 0x00 \cup (\bigcup_{i=1}^{n-1} \mathcal{M} \| \mathcal{I} \| i)) = \emptyset$ and $\mathcal{S}$ consists of 256 bit bitstrings, then HKDF_n with domain $\mathcal{S} \times \mathcal{K} \times \mathcal{I}$ is (t_D, t_S, q, ϵ)-indifferentiable from a random oracle, where $\epsilon = O(nq^2/|\mathcal{M}|)$ and $t_S = O(q^2)$, and O just hides small constants.

Proof. Consider

- the game G_0 in which HMAC is a random oracle and HKDF_n is defined from HMAC by (4.1), and
- the game G_1 in which HKDF_n is a random oracle and HMAC is defined as follows.

  Let L be a list of pairs $((k, m), r)$ such that r is the result of a previous call to $\mathsf{HMAC}(k, m)$. The list L is initially empty.

  $\mathsf{HMAC}(k, m) =$

  1. if $((k, m), r) \in L$ for some r, then return r, else
  2. if $((k_0, m_0), k) \in L$ for some $k_0 \in \mathcal{S}$ and $m_0 \in \mathcal{K}$, and $m = info \| 0x00$ for some $info \in \mathcal{I}$, then let $r \| \ldots = \mathsf{HKDF}_n(k_0, m_0, info)$, else
  3. if $((k_0, m_0), k) \in L$ for some $k_0 \in \mathcal{S}$ and $m_0 \in \mathcal{K}$, and $m = k_i \| info \| i$ for some $k_i \in \mathcal{M}$, $info \in \mathcal{I}$ and $1 \leq i < n$, then let $k_1' \| \ldots \| k_i' \| \ldots \| k_n' = \mathsf{HKDF}_n(k_0, m_0, info)$; if $k_i' = k_i$, then $r = k_{i+1}'$;

4. otherwise, let r be a fresh random element of $\mathcal{M}$;
5. add $((k, m), r)$ to L;
6. return r.

We name *direct* oracle calls to HKDF_n or HMAC calls that are done directly by the distinguisher, and *indirect* oracle calls the calls to HMAC done from inside HKDF_n (in G_0) and the calls to HKDF_n done from inside HMAC (in G_1).

Let us show that these two games are indistinguishable as long as, in G_0,

H1. HMAC never returns the same result for different arguments,

H2. no fresh result of HMAC is equal to the first argument of a previous call to HMAC,

H3. the distinguisher never calls $\mathsf{HMAC}(k, m)$ where $k = \mathsf{HMAC}(salt, key)$ has been called from inside HKDF_n but not directly by the distinguisher,

H4.
- $\mathsf{HMAC}(prk, info\|0x00)$ never returns a fresh k_1 such that $\mathsf{HMAC}(prk, k_1\|info\|0x01)$ has been called (directly or indirectly) before, and
- for all $1 \leq i < n$, $\mathsf{HMAC}(prk, k_i\|info\|i)$ never returns a fresh k_{i+1} such that $\mathsf{HMAC}(prk, k_{i+1}\|info\|i+1)$ has been called (directly or indirectly) before,

and in G_1,

H5. there are no two elements $((k, m), r)$ and $((k', m'), r)$ in L with $(k, m) \neq (k', m')$,

H6. if the distinguisher calls $\mathsf{HMAC}(prk, k_i\|info\|i)$ with $1 \leq i < n$, $((salt, key), prk) \in L$, and $\_\|\ldots\|k_i\|\ldots\|\_ = \mathsf{HKDF}_n(salt, key, info)$, then
- $\mathsf{HKDF}_n(salt, key, info)$ has been called *directly* before the call to $\mathsf{HMAC}(prk, k_i\|info\|i)$, *or*
- if $i = 1$ then $\mathsf{HMAC}(prk, info\|0x00)$, or if $i > 1$ then $\mathsf{HMAC}(prk, k_{i-1}\|info\|i-1)$ has been called before directly and returned k_i, and thus $\mathsf{HKDF}_n(salt, key, info)$ has been called *indirectly* before the call to $\mathsf{HMAC}(prk, k_i\|info\|i)$.

We have the following invariant:

P1. Given $salt$, key, there is at most one prk such that $((salt, key), prk) \in L$.

Indeed, when L contains such an element, calls to $\mathsf{HMAC}(salt, key)$ immediately return prk at step 1, and never add another element $((salt, key), prk')$ to L.

Case 1. Suppose the distinguisher makes a direct oracle call to HKDF_n or HMAC 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 $\mathsf{HMAC}(k, m)$ that has not been done before as a direct call.

Case 2. a) In G_0, this HMAC call has already been done as HMAC($salt$, key) from inside HKDF_n. In G_0, the result is $prk = \mathsf{HMAC}(salt, key)$, which is independent from previously returned values, so it looks like a fresh random value to the distinguisher. In G_1, we cannot have $m = info\|0x00$ nor $m = k_i\|info\|i$ for some $1 \leq i \leq n$ because $m = key \in \mathcal{K}$ which is disjoint from $\mathcal{I}\|0x00$ and from $\mathcal{M}\|\mathcal{I}\|i$, so HMAC returns a fresh random value.

Case 2. b) In G_0, this HMAC call has already been done as HMAC(prk, $info\|0x00$) from inside $\mathsf{HKDF}_n(salt, key, info)$. Hence $\mathsf{HMAC}(k, m) = \mathsf{HMAC}(prk, info\|0x00)$ is the first 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$ and $prk = \mathsf{HMAC}(salt, key)$. Since by H3, the distinguisher never calls $\mathsf{HMAC}(k, m)$ where $k = \mathsf{HMAC}(salt, key)$ has been called from inside HKDF_n but not directly by the distinguisher, HMAC($salt$, key) has been called directly by the distinguisher. In G_1, since HMAC($salt$, key) has been called, $((salt, key), prk) \in L$, so $\mathsf{HMAC}(k, m) = \mathsf{HMAC}(prk, info\|0x00)$ returns the first 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$ (step 2), as in G_0.

Case 2. c) In G_0, this HMAC call has already been done as $\mathsf{HMAC}(prk, k_i\|info\|i)$, with $1 \leq i < n$ from inside $\mathsf{HKDF}_n(salt, key, info)$. Hence $\mathsf{HMAC}(k, m) = \mathsf{HMAC}(prk, k_i\|info\|i)$ is the $(i+1)$-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$, $prk = \mathsf{HMAC}(salt, key)$, and $k_i = \mathsf{HMAC}(prk, k_{i-1}\|info\|i-1)$ is the i-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$. As above, HMAC($salt$, key) has been called directly by the distinguisher. In G_1, since HMAC($salt$, key) has been called, $((salt, key), prk) \in L$, so, since k_i is the i-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$, $\mathsf{HMAC}(k, m) = \mathsf{HMAC}(prk, k_i\|info\|i)$ returns the $(i+1)$-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$ (step 3), as in G_0.

Case 2. d) In G_0, this HMAC call has never been done, directly or indirectly. Hence, HMAC returns a fresh random value. In G_1, if $((salt, key), k) \in L$, then HMAC may return one of the 256 bit blocks of $\mathsf{HKDF}_n(salt, key, info)$. However, since $\mathsf{HMAC}(k, m)$ has not been called from HKDF_n in G_0, $\mathsf{HKDF}_n(salt, key, info)$ has not been called directly by the distinguisher, so the result of HMAC always looks like a fresh random value to the distinguisher.

Case 3. Suppose the distinguisher makes a direct call to $\mathsf{HKDF}_n(salt, key, info)$ that has not been done before as a direct call.

Case 3. a) In G_1, this call to HKDF_n has already been done from HMAC. This means the algorithm entered case 2 or 3. Hence $((salt, key), prk) \in L$ and either $\mathsf{HMAC}(prk, info\|0x00)$ or $\mathsf{HMAC}(prk, k_i\|info\|i)$ for an i with $1 \leq i < n$ has been called. Since $((salt, key), prk) \in L$, HMAC($salt$, key) has been called before the call to $\mathsf{HMAC}(prk, info\|0x00)$ or $\mathsf{HMAC}(prk, k_i\|info\|i)$, and it has returned prk.

Case 3. a) i) Suppose that $\mathsf{HMAC}(prk, info\|0x00)$ has been called and it returned k'_1. Furthermore, suppose that $\mathsf{HMAC}(prk, k'_{n'}\|info\|n')$, $1 \leq n' < n$ has been called and returned $k'_{n'+1}$, where $n' = \max\{i|\mathsf{HMAC}(prk, k_i\|info\text{ß})\}$. This means $\mathsf{HMAC}(prk, k'_j\|info\|j)$ with

$n' < j < n$ have not been called. Then by H6, all $\mathsf{HMAC}(prk, k'_i \| info \| i), 1 \leq i \leq n'$ have been called, and returned k'_{i+1}.

To begin with, we treat the first n' 256 bit blocks of HKDF_n's output, because these are already determined by previous calls. By step 2 of the definition of HMAC in G_1, since by H5, the only element of L of the form $(\_, prk)$ is $((salt, key), prk)$, $\mathsf{HMAC}(prk, info \| 0x00)$ is the first 256 bit block of a previous call to $\mathsf{HKDF}_n(salt, key, info)$. In the same way, by step 3 of the definition of HMAC in G_1, $\mathsf{HMAC}(prk, k'_i \| info \| i)$ is the $(i+1)$-th 256 bit block of the same call to $\mathsf{HKDF}_n(salt, key, info)$. The current call to HKDF_n returns the same results for its first n' 256 bit blocks and thus $\mathsf{HKDF}_n(salt, key, info) = k'_1 \| \ldots \| k'_{n'} \| \ldots \| k'_n$, where $k'_1 = \mathsf{HMAC}(prk, info \| 0x00)$ and $k'_{i+1} = \mathsf{HMAC}(prk, k'_i \| info \| i)$. In G_0 we have the same property by definition of HKDF_n.

Now we treat the last $n - n'$ 256 bit blocks. In G_1 they are independent from returned random values. Indeed, if a call to HMAC returns the $(n' + 1)$-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$, then this call occurs in step 3 of HMAC, and it is $\mathsf{HMAC}(prk', m)$ with $((salt, key), prk') \in L$, $m = k''_{n'} \| info \| n'$, and $k''_{n'}$ is the n'-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$. By P1, $prk' = prk$. We have $k'_{n'} = k''_{n'}$, so $\mathsf{HMAC}(prk', m)$ is $\mathsf{HMAC}(prk, k'_{n'} \| info \| n')$. But $\mathsf{HMAC}(prk, k'_{n'} \| info \| n')$ has not been called by the hypothesis of this case, so no previous call to HMAC returns the $(n' + 1)$-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$. Thus the $(n' + 1)$-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$ look like a fresh random value. For calls to HMAC that would return the $(n' + t)$-th 256 bit block, $t > 1$, of HKDF_n, we start with the same reasoning. However, they fail at the point where $k'_{n'+t} = k''_{n'+t}$ would be necessary to continue: This is excluded by H6.

In G_0, the last $n - n'$ 256 bit blocks of $\mathsf{HKDF}_n(salt, key, info)$ are independent of previously returned values. Indeed, $\mathsf{HMAC}(prk, k'_j \| info \| j), n' \leq j < n$ has not been called directly. Furthermore, they have not been called indirectly from previous calls to HKDF_n, because, if $\mathsf{HMAC}(prk, k'_j \| info \| j)$ had been called from some $\mathsf{HKDF}_n(salt', key', info')$, then by $\mathcal{K} \cap (\mathcal{I} \| 0x00 \cup (\bigcup_{i=1}^{n-1} \mathcal{M} \| \mathcal{I} \| i)) = \emptyset$, this call would be the $(j+1)$-th call to HMAC in $\mathsf{HKDF}_n(salt', key', info')$, $prk = \mathsf{HMAC}(salt', key')$, and $info' = info$. Since by H1, HMAC never returns the same result for different arguments, this would imply $salt' = salt$ and $key' = key$, contradicting that $\mathsf{HKDF}_n(salt, key, info)$ has not been called before. Therefore, the last $n - n'$ 256 bit blocks of $\mathsf{HKDF}_n(salt, key, info)$ look like a fresh random value.

Case 3. a) ii) Otherwise, $\mathsf{HMAC}(prk, info \| 0x00)$ has not been called.

If $\mathsf{HKDF}_n(salt, key, info)$ had been called indirectly from step 2 of HMAC, then we would have called $\mathsf{HMAC}(prk', m)$ with $((salt, key), prk') \in L$ and $m = info \| 0x00$. Furthermore, by P1, $prk' = prk$, so we would have called $\mathsf{HMAC}(prk, info \| 0x00)$. Contradiction. So $\mathsf{HKDF}_n(salt, key, info)$ has not been called from step 2 of HMAC.

Therefore, $\mathsf{HKDF}_n(salt, key, info)$ has been called from step 3 of HMAC. If $\mathsf{HKDF}_n(salt, key, info)$ had been called at step 3 of HMAC and the next 256 bit block was returned, then the distinguisher would have called $\mathsf{HMAC}(prk', k'_i \| info \| i, 1 \leq i < n)$ with $((salt, key), prk') \in L$ and $\ldots \| k'_i \| \ldots = \mathsf{HKDF}_n(salt, key, info)$. By H6, $\mathsf{HKDF}_n(salt, key, info)$ would have been called before, either directly (excluded by hypothesis) or indirectly. We continue this argument recursively using H6 until the conclusion that ultimately, $\mathsf{HKDF}_n(salt, key, info)$ would have been called indirectly at step 2. Then the distinguisher would have called

$\mathsf{HMAC}(prk'', info\|0x00)$ with $((salt, key), prk') \in L$, so by P1, $prk'' = prk$, so this is excluded by hypothesis. Therefore, no 256 bit blocks of $\mathsf{HKDF}_n(salt, key, info)$ were returned at step 3. So far, this paragraph can be summarised as: Because $\mathsf{HMAC}(prk, info\|0x00)$ has not been called by hypothesis, the first 256 bit block of HKDF_n was not returned. Because of the above argument, neither were the other bits of HKDF_n returned. We can conclude that in G_1, the value of $\mathsf{HKDF}_n(salt, key, info)$ is independent from previously returned values, so it looks like a fresh random value.

In G_0, $\mathsf{HMAC}(salt, key)$ has been called directly and returned prk, $\mathsf{HMAC}(prk, info\|0x00)$ has not been called directly. If a previous call to $\mathsf{HKDF}_n(salt', key', info')$ called $\mathsf{HMAC}(prk, info\|0x00)$, then we would have $info' = info$ and $prk = \mathsf{HMAC}(salt', key')$. By H1, this would imply $salt' = salt$ and $key' = key$, so $\mathsf{HKDF}_n(salt, key, info)$ would have been called before, which is excluded by hypothesis. Therefore, $\mathsf{HMAC}(prk, info\|0x00)$ has not been called before, neither directly nor indirectly. By H4, $\mathsf{HMAC}(prk, k_1\|info\|1)$ has not been called before, with $k_1 = \mathsf{HMAC}(prk, info\|0x00)$, which is a call invoked by the new call to $\mathsf{HKDF}_n(salt, key, info)$. Also by H4 the same reasoning concludes that $\mathsf{HMAC}(prk, k_i\|info\|i)$, $1 \leq i < n$ have not been called. Therefore, $\mathsf{HMAC}(prk, info\|0x00)$ and $\mathsf{HMAC}(prk, k_i\|info\|i)$, $1 \leq i < n$ have not been called before, so their result is independent from previously returned values. Hence $\mathsf{HKDF}_n(salt, key, info)$ is independent from previously returned values, as in G_1.

Case 3. b) In G_1, this HKDF_n call has never been done, directly or indirectly. Hence HKDF_n returns a fresh random value. In G_0, the result is obtained from calls to HMAC. The distinguisher has not made these calls to HMAC directly calling $\mathsf{HMAC}(salt, key)$ first, because otherwise the simulator for HMAC in G_1 would have called $\mathsf{HKDF}_n(salt, key, info)$ in step 2 or 3. Furthermore, it cannot call $\mathsf{HMAC}(salt, key)$ with result prk after calling $\mathsf{HMAC}(prk, info\|0x00)$ or $\mathsf{HMAC}(prk, k_i\|info\|i)$, $1 \leq i < n$, by H2. So the result of HKDF_n is independent of the result of direct HMAC calls made by the distinguisher. Moreover, other calls to HKDF_n did not generate the same last n calls to HMAC, because by H1, the first call to HMAC, $\mathsf{HMAC}(salt, key)$, never returns the same result for different arguments. So the result looks like a fresh random value to the distinguisher.

The previous proof shows that the games G_0 and G_1 are indistinguishable assuming the hypotheses H1–H6 hold. Let us bound the probability that they do not hold. Suppose that there are at most q (direct or indirect) queries to HMAC.

- The probability that H1 does not hold is at most the probability that among q random values in $\mathcal{M}$, two of them collide, so it is at most $q^2/|\mathcal{M}|$.
- The probability that H2 does not hold is at most the probability that among q random values in $\mathcal{M}$, one of them is equal to one among the q first arguments of HMAC queries, so it is also at most $q^2/|\mathcal{M}|$.
- When H3 does not hold, the distinguisher calls $\mathsf{HMAC}(k, m)$ for a value k that happens to be equal to $\mathsf{HMAC}(salt, key)$, which is independent of the values the distinguisher has seen, since $\mathsf{HMAC}(salt, key)$ has not been called directly by the

distinguisher. There are at most q values $\mathsf{HMAC}(salt, key)$, and the distinguisher has q attempts, so the probability that H3 does not hold is at most $q^2/|\mathcal{M}|$.

- H4 means that $\mathsf{HMAC}(prk, \cdot)$ never returns a fresh result k that was already used *before* in a direct call to $\mathsf{HMAC}(prk, k\|info\|\cdot)$ which is the directly subsequent call to $\mathsf{HMAC}(prk, \cdot)$ in HKDF_n.

  Thus, when H4 does not hold, the fresh random value from HMAC collides with a previously fixed k. There are at most q values $\mathsf{HMAC}(prk, \cdot)$ and at most q values k, so the probability that H4 does not hold is at most $q^2/|\mathcal{M}|$.

- Let us show that, if the random values r chosen at step 4 are all distinct and distinct from all 256 bit blocks of HKDF_n results used in HMAC, then H5 holds. The proof is by induction on the sequence of calls of HMAC. If $((k, m), r)$ is added to L and r comes from a result of HKDF_n at step 2 or 3, then k determines k_0, m_0 uniquely by induction hypothesis, and m determines *info* as well as which 256 bit block of the result of HKDF_n is r, hence r is uniquely determined from k, m, and distinct from elements chosen at step 4 by hypothesis. If $((k, m), r)$ is added to L and r is chosen at step 4, then r is always distinct from elements already in L by hypothesis. This concludes the proof of our claim.

  From this claim, we can easily see that the probability that H5 does not hold is at most $q^2/|\mathcal{M}|$.

- When H6 does not hold, the distinguisher calls $\mathsf{HMAC}(prk, k_i\|info\|i)$ and k_i happens to be equal to the i-th 256 bit block of $\mathsf{HKDF}_n(salt, key, info)$ which is independent from values returned to the distinguisher. There are at most q calls to HMAC, and at most q values k_i they could collide with, thus the probability that H6 does not hold is at most $q^2/|\mathcal{M}|$.

Hence, the probability that the distinguisher distinguishes G_0 from G_1 is at most $6q^2/|\mathcal{M}|$. This is the same probability that was found for $n = 2$ in [39], thus we do not loose probability by generalising the proof to HKDF_n. □

We now want to justify that the hypothesis of Lemma 1 is satisfied in our case. First of all, *info* is empty in WireGuard. Then, $\mathcal{K}$ consists of bitstrings of length 256 bit = 32 byte, $\mathcal{I}\|0x00$ consists of bitstrings of length 1 byte, and $\mathcal{M}\|\mathcal{I}\|i$ consists of bitstrings of length 33 byte.

Indifferentiability of an HKDF_n chain For the next step of the proof we assume that HKDF_n is a random oracle that outputs bitstrings of length l. Because of this, we will just denote it by H, leaving out the n, because it does not matter anymore how many calls to HMAC are used to build the output. What allows us to abstract this is the indifferentiability proof in the previous section (and the lemma about indifferentiability of truncation of random oracles, to be able to have values of l that are not an integer multiple of the length of HMAC's output). Also, this change in notation makes it possible to use n unambiguously as an index for the Chain functions.

Definition 32 (HKDF-like 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 $(l - l')$. We define the functions $\mathsf{Chain}_n, 0 \leq n < m$ and the function Chain_m in the following way:

$$\begin{aligned}
&\mathsf{Chain}_n(v_0, \ldots, v_n) = \\
&\quad C_0 = \mathsf{const} \\
&\quad \text{for } j = 0 \text{ to } n: \\
&\qquad C_{j+1} \| r_j = \mathsf{H}(C_j, v_j) \\
&\quad \text{return } r_n
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&\mathsf{Chain}_m(v_0, \ldots, v_m) = \\
&\quad C_0 = \mathsf{const} \\
&\quad \text{for } j = 0 \text{ to } m: \\
&\qquad C_{j+1} \| r_j = \mathsf{H}(C_j, v_j) \\
&\quad \text{return } C_{m+1} \| r_m
\end{aligned} \tag{4.4}$$

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

In WireGuard, $l' = 256$. Also, we define Chain_m specially because in WireGuard, the first 256 bit block of the last HKDF call's output is used as symmetric encryption key in the continuation of the protocol, and not as a chaining value for a next HKDF call.

Lemma 2. Chain_n, $n \leq m$, are (t_D, t_S, q, ϵ)-indifferentiable from independent random oracles, where $\epsilon = O(q^2/2^{l'})$ and $t_S = O(q^2)$, and O just hides small constants.

Proof. 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 (4.3) and (4.4).

- 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 as follows.

  Let L be a list of triples $((C, v), (C_{j+1}, r_j), j)$ suchthat $C_{j+1} \| r_j$ 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.

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

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
   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

We shortly comment this simulator's four cases in an informal way. 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 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 this means that 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.

Similar to the proof of Lemma 1, 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 get's 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 is equal to the first argument of a previous call to H.

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

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 matching element $((C_k, v_k), (C_{k+1}, r_k), k)$ in L.

More informally, this means that 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 this means that the simulator can, in case 3, always reconstruct the whole chain of H calls and collect the arguments v_j.

If there is an element in L with $j > 0$ then this means that 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.

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 from 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. This means 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 $C = \text{const}$, 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 fulfills 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 from 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.[1] 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. If $C = \text{const}$ 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. If $((C_j, v_j), (C, r_j), j) \in L$ for some C_j, v_j, r_j and $0 \leq j < m$, this means that 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

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

for the same reason as Chain_0 above. 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 and H2 hold. We will now bound the probability that they do not hold. Suppose that there are at most q queries, direct or indirect, to H.

- 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 $\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{Chain}_n}\right)$ 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 $\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{Chain}_n}\right) \cdot q_{\mathsf{H}}/2^{l'}$.
- The probability that H2 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 $q_{\mathsf{H}}^2/2^{l'}$.

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

$$\frac{\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{Chain}_n}\right) \cdot q_{\mathsf{H}} + q_{\mathsf{H}}^2}{2^{l'}}.$$

□

For completeness, we reproduce Lemma 3 and its proof from [39]. It states that the truncation of a random oracle is indifferentiable from a random oracle.

Lemma 3. If H is a random oracle that returns bitstrings of length l, then the truncation of H to length $l' < l$ is $(t_D, t_S, q, 0)$-indifferentiable from a random oracle, where $t_S = O(q)$.

Proof. Consider

- the game G_0 in which H is a random oracle, and $H'(x)$ is $H(x)$ truncated to length l', and
- the game G_1 in which H' is a random oracle that returns bitstrings of length l' and $H(x) = H'(x)||H''(x)$ where H'' is a random oracle that returns bitstrings of length $l - l'$.

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

As a last step, we prove the following lemma.

Lemma 4. If H_1 and H_2 are random oracles defined on the same domain D that return bitstrings of length l_1 and l_2 respectively, then a random oracle H defined on the same domain that returns bitstrings of length $(l_1 + l_2)$ is $(t_D, t_S, q, 0)$-indifferentiable from this construction.

Proof. Consider

- the game G_0 in which H_1 and H_2 are random oracles and H is the concatenation of their outputs: $H(x) = H_1(x)||H_2(x)$.

- the game G_1 in which H is a random oracle, $H_1(x)$ is the first l_1 bit of H's output and $H_2(x)$ is the last l_2 bit of H's output.

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

In WireGuard, $l_1 = l_2 = 512$.

By combining Lemmas 1 to 4, we conclude that our modelling of WireGuard's HKDF chains is indifferentiable from the original, with $n = 3$.

Coming back to the modelling of the HKDF chain in CryptoVerif. For the first two random oracles, we can instantiate macros built-in to CryptoVerif's library: one for two arguments and one for three arguments. This gives us the following definitions:

```
type hashkey_t [fixed].
type hashoutput_t [large,fixed].
const dummy_hashoutput: hashoutput_t.
fun hashoutput_to_bitstring(hashoutput_t): bitstring [data].

fun rom1(hashkey_t, G_t, G_t): key_t.
fun rom2(hashkey_t, G_t, G_t, G_t): key_t.
```

For the last one, we create our own macro for seven arguments. Also, because the output of the third macro needs to be split into four variables, we define a type `four_keys_t` and functions to split it into individual keys.

```
type four_keys_t [large,fixed].

fun concat(key_t, key_t, key_t, key_t): four_keys_t [data].
fun get_part1(four_keys_t): key_t [projection].
fun get_part2(four_keys_t): key_t [projection].
fun get_part3(four_keys_t): key_t [projection].
fun get_part4(four_keys_t): key_t [projection].
```

The function `rom3_intermediate` is a random oracle with an equivalence as the one defined above, and `rom3` is a convenience wrapper around it, directly returning the four individual keys.

```
fun rom3_intermediate(hashkey_t, G_t, G_t, G_t, G_t, G_t, G_t, psk_t): ↩
    four_keys_t.

letfun rom3(key_rom: hashkey_t, arg1: G_t, arg2: G_t, arg3: G_t, arg4: G_t, ↩
    arg5: G_t, arg6: G_t, arg7: psk_t) =
  let parts: four_keys_t = rom3_intermediate(key_rom, arg1, arg2, arg3, arg4,↩
      arg5, arg6, arg7) in
  let part1: key_t = get_part1(parts) in
  let part2: key_t = get_part2(parts) in
  let part3: key_t = get_part3(parts) in
  let part4: key_t = get_part4(parts) in
  (part1, part2, part3, part4).
```

We need to formally establish for CryptoVerif to work with it, that the individual keys are independent random values. We do this by defining the following equivalence:

```
equiv(split_hashoutput)
    foreach ik <= Nk do r <-R hashoutput_t; (O1() := return(get_part1(r)) |
                          O2() := return(get_part2(r)) |
                          O3() := return(get_part3(r)) |
                          O4() := return(get_part4(r)) |
                          O5() := return(r))
  <=(0)=>
    foreach ik <= Nk do part1 <-R part1_t;
            part2 <-R part2_t;
            part3 <-R part3_t;
            part4 <-R part4_t;
            (O1() := return(part1) |
             O2() := return(part2) |
             O3() := return(part3) |
             O4() := return(part4) |
             O5() := return(concat(part1, part2, part3, part4))).
```

In our model we can then just use the three random oracles `rom1`, `rom2`, and `rom3`. After they are converted to `find` constructs, we need to apply the equivalence `split_hashoutput` so CryptoVerif can continue working with the individual keys.

Hash Function. The hash function employed by WireGuard needs to be a collision resistant hash function following the Noise specification. The CryptoVerif cryptographic library contains a macro for a collision resistant hash function. We adapt this to a hash function taking two inputs, because the protocol always hashes the concatenation of a previous hash output with a new input.

```
1  def CollisionResistant_hash_pair(key_t, hashinput1_t, hashinput2_t, ↩
       hashoutput_t, hash, hashoracle, P_hash) {
2
3  fun hash(key_t, hashinput1_t, hashinput2_t):hashoutput_t.
4
5  collision k <-R key_t; forall x1:hashinput1_t, x2:hashinput2_t, y1:↩
       hashinput1_t, y2:hashinput2_t;
6      return(hash(k, x1, x2) = hash(k, y1, y2)) <=(P_hash(time))=> return(x1 = ↩
           y1 && x2 = y2).
7
8  channel ch1, ch2.
9  let hashoracle(k: key_t) =
10     in(ch1, ());
11     out(ch2, k).
12 }
```

The equivalence in line 5 is a special notation for collisions. It replaces the comparison of the results of two calls to the same hash function by a comparison of the arguments of the calls. The probability to distinguish is then the probability that the attacker finds a

collision in a given time. Note that the hash function takes a key as first input. This just serves to model different hash functions with the same macro. We could thus name the key we use in our model `blake2s`. Because the attacker needs to be able to use the hash function, the macro makes an oracle available that the attacker can call to retrieve the key. We use the following code to instantiate the macro:

```
1  proba P_hash.    (* probability of breaking collision resistance *)
2  expand CollisionResistant_hash_pair(
3    (* types *)
4    hashkey_t,    (* key of the hash function, models the choice of *)
5                  (* the hash function *)
6    hashoutput_t, (* first argument that gets hashed. See the comment *)
7                  (* just above this macro for an explanation. *)
8    bitstring,    (* second argument that gets hashed. *)
9    hashoutput_t, (* output type of the hash function *)
10   (* functions *)
11   hash,     (* name of the hash function: *)
12         (* hash(hashkey_t, hashoutput_t, bitstring): hashoutput_t *)
13   (* processes *)
14   hash_oracle,  (* name of the oracle that will make available the *)
15                 (* hash key to the attacker *)
16   (* parameters *)
17   P_hash        (* probability of breaking collision resistance *)
18 ).
19 (* constants used in the transcript hashing *)
20 const hash_construction_identifier : hashoutput_t.
21   (* This is hash( hash("Noise_IK...") || "WireGuard v1 ..." ), and it's *)
22   (* the same for all parties, so no need to calculate it with hash() *)
23 const label_mac1: hashoutput_t.  (* This is "mac1----" *)
```

The comment in line 12 describes how the hash function is called. It takes the key, then a value of type `hashoutput_t` and a second value of type `bitstring`. The function returns a value of type `hashoutput_t`, equally. This chain of `hashoutput_t` values needs to be started somewhere, and that is why we define a constant in line 20. It stands for the result of the hash computation in the second line of the first protocol message. This result is a protocol constant, so in our model it is not necessary to calculate it dynamically.

We define several wrapper functions around `hash`, taking care of type conversion and thus making the code for the protocol messages easier to read:

```
1  letfun mix_hash_G(key_hash: hashkey_t, prev_hash: hashoutput_t, value: G_t) =
2    hash(key_hash, prev_hash, G_to_bitstring(value)).
3
4  letfun mix_hash_bitstring(key_hash: hashkey_t, prev_hash: hashoutput_t, value↩
       : bitstring) =
5    hash(key_hash, prev_hash, value).
6
7  letfun mix_hash_key(key_hash: hashkey_t, prev_hash: hashoutput_t, value: ↩
       key_t) =
```

```
8    hash(key_hash, prev_hash, key_to_bitstring(value)).
```

4.3.2. Modelling the Protocol Messages, Timestamps and Nonces

Before showing how we calculate the protocol messages in CryptoVerif, we define some more types specific to the WireGuard protocol:

```
1  type msg_type_t [fixed]. (* 1 byte msg type field *)
2  const msg_type_init2resp:    msg_type_t.
3  const msg_type_resp2init:    msg_type_t.
4  const msg_type_data:         msg_type_t.
5  const msg_type_cookie_reply: msg_type_t.
6
7  type reserved_t [fixed]. (* 3 byte reserved field *)
8  const reserved: reserved_t.
9
10 type session_index_t [fixed]. (* 4 byte session identifier field *)
11 const dummy_session_index: session_index_t.
12 type timestamp_t [fixed].     (* 12 byte timestamps *)
13 const dummy_timestamp: timestamp_t.
14 fun timestamp_to_bitstring(timestamp_t): bitstring [data].
```

Timestamps. WireGuard specifies that the responder keeps the latest timestamp it received *per peer*. In CryptoVerif's process calculus, variables cannot be overwritten. Also, it is not possible to express order relations, which we would need to handle strictly increasing timestamps. Instead, we store all received timestamps in a table and do not accept a message with an already used timestamp.

```
1  table rcvd_timestamps(G_t, G_t, timestamp_t).
```

This is not an exact model of the real protocol, but it effectively prevents the attacker from replaying a first protocol message. A timestamp is not a uniform random value, thus choosing it randomly within the type `timestamp_t` would be a too strong assumption. It is also not a constant, because it is different for each first protocol message. We decided to let the attacker provide the timestamp to be used.

Nonces. Nonces are incremented by one with each message, which we also cannot model in CryptoVerif. Just as with the timestamps, we use tables to keep track of already used nonces. Also, we let the attacker choose which nonce should be used in a transport data message and abort if it provides an already used nonce.

```
1  type side.
2  const is_initiator: side.
3  const is_responder: side.
4  (* the bitstring is used as tuple (side, replication_index) *)
5  table sent_nonces(bitstring, nonce_t).
6  table recv_nonces(bitstring, nonce_t).
```

We use one table for nonces used to send (encrypt) messages, and another table for nonces used to receive (decrypt) messages. We use a tuple of a variable indicating initiator or responder, and the replication index as index for the table, to separate the nonces used by each participant, in each session.

The Protocol Messages. The computation of the protocol messages has been separated into functions `prepare` and `process` for each protocol message. These functions are called by initiator and responder respectively. In the following, variable names ending in `_i` denote a variable from the initiator, and the ending `_r` a variable from the responder. Two-letter combinations of `e` and `s` denote a Diffie-Hellman computation between an ephemeral (`e`) or long-term (`s`) key, where the first letter indicates the used key from the initiator and the second key the used key from the responder. The ending `_i` or `_r` in this case denotes the initiators or the responders *view* on the variable. In a successful protocol execution between initiator and responder, `es_i = es_r` etc. The suffix `_recv` denotes a variable received or derived from a variable received from the other peer. Again, in a successful protocol run, `timestamp_i = timestamp_i_recv`.

The goal of the separation into functions for preparing and processing protocol messages is to make comparison with both the specification and implementations easier.

First Protocol Message. The first protocol message from the initiator to the responder is calculated as follows. The function receives the necessary hash and random oracle keys as argument, as well as the long-term public key S_X^{pub}, the initiator's keypair (S_i^{priv}, S_i^{pub}), and the timestamp. The peer's long-term public key is called `S_X_pub_foo`, because for technical reasons we could not re-use `S_X_pub`, which is already used in the top-level process (the top-level process will be described later).

```
letfun prepare1(
          key_hash: hashkey_t,
          key_rom1: hashkey_t,
          key_rom2: hashkey_t,
          S_X_pub_foo: G_t,
          S_i_priv: Z_t,
          S_i_pub: G_t,
          timestamp_i: timestamp_t) =

  new I_i: session_index_t;
  new E_i_priv: Z_t;
  let E_i_pub: G_t = exp(g, E_i_priv) in

  let H_i1: hashoutput_t = mix_hash_G(key_hash, hash_construction_identifier,↩
       S_X_pub_foo) in
  let H_i2: hashoutput_t = mix_hash_G(key_hash, H_i1, E_i_pub) in

  let es_i: G_t = DH(S_X_pub_foo, E_i_priv) in
  let k_i2: key_t = rom1(key_rom1, E_i_pub, es_i) in

```

```
20    let static_i_enc: bitstring = enc_G(S_i_pub, H_i2, k_i2, nonce_0) in
21    let H_i3: hashoutput_t = mix_hash_bitstring(key_hash, H_i2, static_i_enc) ↩
          in
22
23    let ss_i: G_t = DH(S_X_pub_foo, S_i_priv) in
24    let k_i3: key_t = rom2(key_rom2, E_i_pub, es_i, ss_i) in
25
26    let timestamp_i_enc: bitstring = enc_timestamp(timestamp_i, H_i3, k_i3, ↩
          nonce_0) in
27
28    let H_i4: hashoutput_t = mix_hash_bitstring(key_hash, H_i3, timestamp_i_enc↩
          ) in
29
30    let msg_alpha: bitstring = concat_msg_alpha_1(msg_type_init2resp, reserved,↩
           I_i, E_i_pub, static_i_enc, timestamp_i_enc) in
31    let mac1_i: mac_t = mac(msg_alpha, hash(key_hash, label_mac1, ↩
          G_to_bitstring(S_X_pub_foo))) in
32    (* Dummy mac2 for the moment *)
33    new mac2_i: mac_t;
34    (I_i, E_i_priv, E_i_pub, static_i_enc, timestamp_i_enc, mac1_i, mac2_i, ↩
          es_i, ss_i, H_i4).
```

This calculation is a direct translation of the WireGuard protocol as it is described in Section 3.2, except for the different modelling of the HKDF chain. The first symmetric key is derived with the `rom1` function; it depends on E_i^{pub} and the Diffie-Hellman function between S_X^{pub} and E_i^{priv}, just as in the original protocol. The second symmetric key is derived with the `rom2` function; it depends on E_i^{pub}, the Diffie-Hellman function between S_X^{pub} and E_i^{priv}, and the Diffie-Hellman function between S_X^{pub} and S_i^{priv}, just as in the original protocol. We do not model the second MAC. It is part of the cookie reply system, that a party can use in case of load, to force the sending party to do another roundtrip. Also, the second MAC only depends on public values. This means that in our model (we will elaborate on that later), the attacker knows with whom the parties are talking and thus, could just calculate the second MAC itself. The last line of the function returns all values needed to send the first protocol message (`I_i, E_i_pub, static_i_enc, timestamp_i_enc, mac1_i, mac2_i`) and to continue the protocol (`es_i, ss_i, H_i4`). The ephemeral private key is returned for its optional compromise.

The responder, upon receiving a first protocol message, calls the following function to process it. The function takes the usual hash keys as arguments, all variables received in the protocol message, and the responder's long-term key pair (E_r^{priv}, E_r^{pub}).

```
1  letfun process1(
2            key_hash: hashkey_t,
3            key_rom1: hashkey_t,
4            key_rom2: hashkey_t,
5            S_r_priv: Z_t,
6            S_r_pub: G_t,
7            I_i_recv: session_index_t,
```

```
 8           E_i_pub_recv: G_t,
 9           static_i_enc_recv: bitstring,
10           timestamp_i_enc_recv: bitstring,
11           mac1_i_recv: mac_t, mac2_i_recv: mac_t
12         ) =
13
14   let msg_alpha: bitstring = concat_msg_alpha_1(msg_type_init2resp, reserved,↩
         I_i_recv, E_i_pub_recv, static_i_enc_recv, timestamp_i_enc_recv) in
15   if check(msg_alpha, hash(key_hash, label_mac1, G_to_bitstring(S_r_pub)), ↩
        mac1_i_recv) then
16   (
17     (* We don't verify mac2. *)
18
19     let H_r1: hashoutput_t = mix_hash_G(key_hash, ↩
           hash_construction_identifier, S_r_pub) in
20     let H_r2: hashoutput_t = mix_hash_G(key_hash, H_r1, E_i_pub_recv) in
21
22     let es_r: G_t = DH(E_i_pub_recv, S_r_priv) in
23     let k_r2: key_t = rom1(key_rom1, E_i_pub_recv, es_r) in
24
25     let injbot(G_to_bitstring(S_i_pub_recv: G_t)) = dec_ad_hash(↩
           static_i_enc_recv, H_r2, k_r2, nonce_0) in
26     (
27       let H_r3: hashoutput_t = mix_hash_bitstring(key_hash, H_r2, ↩
             static_i_enc_recv) in
28
29       let ss_r: G_t = DH(S_i_pub_recv, S_r_priv) in
30       let k_r3: key_t = rom2(key_rom2, E_i_pub_recv, es_r, ss_r) in
31
32       let injbot(timestamp_to_bitstring(timestamp_i_recv: timestamp_t)) = ↩
             dec_ad_hash(timestamp_i_enc_recv, H_r3, k_r3, nonce_0) in
33       (
34         let H_r4: hashoutput_t = mix_hash_bitstring(key_hash, H_r3, ↩
               timestamp_i_enc_recv) in
35         (true, es_r, ss_r, S_i_pub_recv, H_r4, timestamp_i_recv)
36       ) else (
37         (* timestamp did not decrypt *)
38         (false, dummy_g, dummy_g, dummy_g, dummy_hashoutput, dummy_timestamp)
39       )
40     ) else (
41       (* static did not decrypt *)
42       (false, dummy_g, dummy_g, dummy_g, dummy_hashoutput, dummy_timestamp)
43     )
44   ) else (
45     (* mac1 did not verify *)
46     (false, dummy_g, dummy_g, dummy_g, dummy_hashoutput, dummy_timestamp)
47   ).
```

This function does the same computations, just replacing the Diffie-Hellman computations accordingly. At the verification of the MAC, and the two decryptions, the function returns directly in case of error. The first return value is then set to false, which permits the caller of the function to abort the protocol. In lines 25 and 32, pattern matching is used to convert the type and check for correct decryption with `injbot`. In case of success, the function returns all variables needed to continue the protocol by computing the second protocol message.

Second Protocol Message. The second protocol message is sent by the responder. It calls the following function to prepare it:

```
1  letfun prepare2(
2      key_hash: hashkey_t,
3      key_rom3: hashkey_t,
4      I_i_recv: session_index_t, S_i_pub_recv: G_t, E_i_pub_recv: G_t,
5      H_r4: hashoutput_t, es_r: G_t, ss_r: G_t, Q: psk_t) =
6
7    new I_r: session_index_t;
8    new E_r_priv: Z_t;
9    let E_r_pub: G_t = exp(g, E_r_priv) in
10
11   let ee_r: G_t = DH(E_i_pub_recv, E_r_priv) in
12   let se_r: G_t = DH(S_i_pub_recv, E_r_priv) in
13
14   let H_r5: hashoutput_t = mix_hash_G(key_hash, H_r4, E_r_pub) in
15
16   let (tau_r4: key_t, k_r4: key_t, T_r_recv: key_t, T_r_send: key_t) = rom3(↩
         key_rom3, E_i_pub_recv, es_r, ss_r, E_r_pub, ee_r, se_r, Q) in
17   (
18     let H_r6: hashoutput_t = mix_hash_key(key_hash, H_r5, tau_r4) in
19
20     let empty_bitstring_r_enc: bitstring = enc_bitstring(empty_bitstring, ↩
           H_r6, k_r4, nonce_0) in
21     let H_r7: hashoutput_t = mix_hash_bitstring(key_hash, H_r6, ↩
           empty_bitstring_r_enc) in
22
23     let msg_alpha: bitstring = concat_msg_alpha_2(msg_type_init2resp, ↩
           reserved, I_r, I_i_recv, E_r_pub, empty_bitstring_r_enc) in
24     let mac1_r: mac_t = mac(msg_alpha, hash(key_hash, label_mac1, ↩
           G_to_bitstring(S_i_pub_recv))) in
25     (* Dummy mac2 for the moment *)
26     new mac2_r: mac_t;
27
28     (true, I_r, E_r_priv, E_r_pub, T_r_recv, T_r_send, empty_bitstring_r_enc,↩
          mac1_r, mac2_r)
29   ) else (
```

```
30      (false, dummy_session_index, dummy_z, dummy_g, dummy_key, dummy_key, ↩
            dummy_bitstring, dummy_mac, dummy_mac)
31    ).
```

This is an exact translation of the protocol message defined in Section 3.2 execpt line 16. This is the derivation of both the third symmetric key and the transport data keys. They depend on both the initiator's and the responder's ephemeral public keys, all Diffie-Hellman computations, and the pre-shared symmetric key Q. The pattern matching in line 16 cannot fail the way we defined the type conversion, but in `letfun` function definitions, CryptoVerif demands to handle this case, and that is why we need to include line 30. The function returns all values needed to send the second protocol message (`I_r, E_r_pub, empty_bitstring_r_enc, mac1_r, mac2_r`), and continue the protocol (`T_r_recv, T_r_send`). The ephemeral private key is returned for its optional compromise.

The initiator uses the following function to process the second protocol message.

```
1  letfun process2(
2            key_hash: hashkey_t,
3            key_rom3: hashkey_t,
4            I_i: session_index_t, I_r_recv: session_index_t, E_i_priv: Z_t, ↩
                 E_i_pub: G_t, S_i_priv: Z_t, S_i_pub: G_t,
5            E_r_pub_recv: G_t, empty_bitstring_r_enc_recv: bitstring, ↩
                 mac1_r_recv: mac_t, mac2_r_recv: mac_t,
6            H_i4: hashoutput_t,
7            es_i: G_t, ss_i: G_t, Q: psk_t) =
8
9    let ee_i: G_t = DH(E_r_pub_recv, E_i_priv) in
10   let se_i: G_t = DH(E_r_pub_recv, S_i_priv) in
11
12   let msg_alpha: bitstring = concat_msg_alpha_2(msg_type_init2resp, reserved,↩
          I_r_recv, I_i, E_r_pub_recv, empty_bitstring_r_enc_recv) in
13   if check(msg_alpha, hash(key_hash, label_mac1, G_to_bitstring(S_i_pub)), ↩
         mac1_r_recv) then
14   (
15     (* We don't verify mac2 at the moment. *)
16
17     let H_i5: hashoutput_t = mix_hash_G(key_hash, H_i4, E_r_pub_recv) in
18
19     let (tau_i4: key_t, k_i4: key_t, T_i_send: key_t, T_i_recv: key_t) = rom3↩
           (key_rom3, E_i_pub, es_i, ss_i, E_r_pub_recv, ee_i, se_i, Q) in
20     (
21       let H_i6: hashoutput_t = mix_hash_key(key_hash, H_i5, tau_i4) in
22
23       let injbot(=empty_bitstring) = dec_ad_hash(empty_bitstring_r_enc_recv, ↩
             H_i6, k_i4, nonce_0) in
24       (
25         let H_i7: hashoutput_t = mix_hash_bitstring(key_hash, H_i6, ↩
               empty_bitstring_r_enc_recv) in
26         (true, T_i_send, T_i_recv)
```

```
27        ) else (
28          (* empty_bitstring_r_enc_recv did not decrypt *)
29          (false, dummy_key, dummy_key)
30        )
31      ) else (
32        (* weird case where the rom3 pattern matching did not work *)
33        (false, dummy_key, dummy_key)
34      )
35    ) else (
36      (* mac1 did not verify *)
37      (false, dummy_key, dummy_key)
38    ).
```

This does the same computations, just with the according Diffie-Hellman parameters. In case of failed MAC verification, failed pattern matching of `rom3`, or failed decryption, an error tuple is returned. Otherwise all values needed to proceed in the protocol are returned, which are the two transport data keys.

Third Protocol Message. The third protocol message is the first transport data message. We model it separately, because it is only sent from initiator to responder. For the other transport data messages we use functions that are called from both initiator and responder. The function takes as argument the transport data key T_i^{send} and the index `side_index` for the table of sent nonces. The other arguments are needed for the left-or-right message indistinguishability game that is used to model the secrecy properties of the protocol. `clean` is a boolean that indicates if the initiator talks to a honest party and specifically not to the attacker. Also, it is set to `false` if the session is in a compromise scenario that would trivially break the protocol. The boolean `secret_bit` is the global bit that determines which one of the two attacker-provided plaintexts `plaintext_0` or `plaintext_1` should be encrypted. The function proceeds as follows:

```
1  letfun prepare3(clean: bool,
2          secret_bit_I: bool, plaintext_0: bitstring, plaintext_1: bitstring,
3          side_index: bitstring, T_i_send: key_t) =
4
5    if (Zero(plaintext_0) = Zero(plaintext_1)) && (clean || (plaintext_0 = ↩
         plaintext_1)) then
6    (
7      (* Send a transport data message *)
8      let plaintext: bitstring = test(secret_bit_I, plaintext_0, plaintext_1) ↩
           in
9      let ciphertext_keyconfirmation = enc(plaintext, empty_bitstring, T_i_send↩
           , nonce_0) in
10     insert sent_nonces(side_index, nonce_0);
11     (true, ciphertext_keyconfirmation, plaintext)
12   ) else (
13     (* we do not play because either
14        * the plaintexts do not have the same length, or
```

```
15         * we are talking to the attacker and the plaintexts are not equal *)
16       (false, dummy_bitstring, dummy_bitstring)
17     ).
```

The function first checks if it is in any scenario in which it is safe to encrypt something with the transport data key. This is the case if the plaintexts have the same length and it is an clean session as defined above. If it is not a clean session, but the plaintexts are equal, it can also proceed: In this case, encryption does not reveal the secret bit. The game is not entirely aborted in case of a non-clean session because we want to be able to prove correspondence properties. After checking this condition, the function proceeds by choosing the plaintext and encrypting it. The first transport data message is defined to use the zero nonce, which is added to the table after encryption. The functions returns ciphertext and plaintext. We defined the `test` function with the following equations:

```
1  fun test(bool, bitstring, bitstring) : bitstring.
2
3  equation forall x:bitstring, y:bitstring; test(true, x, y) = x.
4  equation forall x:bitstring, y:bitstring; test(false, x, y) = y.
5
6  (* Knowing the equations defined above, this can be deduced, but
7     CryptoVerif can't do this on its own. *)
8  equation forall x:bitstring, b:bool; test(b,x,x) = x.
9  equation forall x:bitstring, y:bitstring, b:bool; Zero(test(b,x,y)) = test (b↩
       ,Zero(x),Zero(y)).
```

The reason for this is that CryptoVerif's process calculus does not allow branches to be re-united. We could use `if` to branch on `secret_bit`, but then we would need to duplicate all code.

The responder calls the following function to handle this first transport data message:

```
1  letfun process3(
2         ciphertext_keyconfirmation_recv: bitstring,
3         T_r_recv: key_t, side_index: bitstring) =
4
5    let injbot(plaintext) = dec(ciphertext_keyconfirmation_recv, ↩
         empty_bitstring, T_r_recv, nonce_0) in
6    (
7      insert recv_nonces(side_index, nonce_0);
8      (true, plaintext)
9    ) else (
10     (* ciphertext did not decrypt *)
11     (false, dummy_bitstring)
12   ).
```

Transport Data Messages. Initiator and responder process both use the following functions to prepare and process a transport data message. The function `prepare_msg` takes the same arguments as `prepare3`, with the attacker-chosen nonce additionally.

```
letfun prepare_msg(
    side_index: bitstring, secret_bit_I: bool,
    plaintext_0: bitstring, plaintext_1: bitstring, nonce: nonce_t,
    clean: bool,
    T_i_send: key_t) =

  if Zero(plaintext_0) = Zero(plaintext_1) && (clean || (plaintext_0 = ↩
      plaintext_1)) then
  (
    get sent_nonces(=side_index, =nonce) in (false, empty_bitstring) else
    insert sent_nonces(side_index, nonce);

    let plaintext = test(secret_bit_I, plaintext_0, plaintext_1) in
    let ciphertext = enc(plaintext, empty_bitstring, T_i_send, nonce) in
    (true, ciphertext, plaintext)
  ) else (
    (false, dummy_bitstring, dummy_bitstring)
  ).
```

The check if we encrypt is exactly the same as in `prepare3`. Before encryption, the function verifies if the nonce was already used. If yes, it aborts and returns an error value. If not, then it inserts the nonce in the table and proceeds with encryption.

Processing of a transport data message by the receiver works equivalently. Before decryption, it is checked if the nonce was already used.

```
letfun process_msg(
    side_index: bitstring, counter_recv: counter_t,
    ciphertext_recv: bitstring, T_i_recv: key_t) =

  let nonce_to_counter(nonce_recv) = counter_recv in
  (
    get recv_nonces(=side_index, =nonce_recv) in (false, empty_bitstring) ↩
        else
    insert recv_nonces(side_index, nonce_recv);
    let injbot(plaintext) = dec(ciphertext_recv, empty_bitstring, T_i_recv, ↩
        nonce_recv) in
    (
      (true, plaintext)
    ) else (
      (* decryption failed *)
      (false, dummy_bitstring)
    )
  ) else (
    (* weird subcase when the nonce can't be casted to a counter *)
    (false, dummy_bitstring)
  ).
```

4.3.3. Execution Environment

In this section, we describe the execution environment of our model, that is the interaction between challenger and attacker. This includes how we set up the game and which oracles the attacker has access to. Also, we will elaborate on our definition of partner sessions. A secondary goal of this section is to describe said components in a style similar to how this is done in eCK-like models and the ACCE model, to facilitate comparison and understanding.

We consider a security game played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. Generally speaking, our goal is to prove that *two* honest parties can execute the protocol securely in an adversarial environment.

Setting Up The Game The adversary starts the security game by providing the challenger with a bit use_psk that indicates if the two honest parties should use a pre-shared key in the protocol:

```
1  process
2    in(c_start, (use_psk: bool));
```

The challenger then proceeds to set up the game. First it randomly chooses the pre-shared key, and then sets the variable `Q` to a choice between this random psk and the constant zero psk.

```
3    new psk: psk_t;
4    let Q: psk_t = optional_psk(use_psk, psk, psk_0) in
```

The function `optional_psk` is just defined as the `test` function used for the two plaintexts: If `use_psk` is true, in evaluates to `psk`, and otherwise to `psk_0`. This permits us to handle both cases in one security game. As a next step, the challenger generates the keys for the hash function and the random oracles that model HKDF:

```
5    new key_hash: hashkey_t;
6    new key_rom1: hashkey_t;
7    new key_rom2: hashkey_t;
8    new key_rom3: hashkey_t;
```

Then, the long-term keys of the two parties are generated:

```
9    new S_i_priv: Z_t;
10   let S_i_pub = exp(g, S_i_priv) in
11   new S_r_priv: Z_t;
12   let S_r_pub = exp(g, S_r_priv) in
```

Finally, the secret bit for the left-or-right message indistinguishability games is chosen, and the challenger hands over control to the attacker by sending the two long-term public keys over a channel:

```
13   new secret_bit : bool;
14   out(c_publickeys, (S_i_pub, S_r_pub));
```

The attacker can then access the following oracles which we will describe in the following.

```
15    (
16     (initiator(key_hash, key_rom1, key_rom2, key_rom3, S_i_priv, S_i_pub, ↩
           S_r_pub, Q, secret_bit)) |
17     (responder(key_hash, key_rom1, key_rom2, key_rom3, S_r_priv, S_i_pub, ↩
           S_r_pub, Q, secret_bit)) |
18     (rom1_oracle(key_rom1)) | (rom2_oracle(key_rom2)) | (rom3_oracle(key_rom3)↩
           ) |
19     (hash_oracle(key_hash)) |
20     (corrupt_S_i(S_i_priv)) |
21     (corrupt_S_r(S_r_priv)) |
22     (corrupt_psk(Q))
23    )
```

Actually, all `in` channels are oracles to the attacker. However, the above processes all start with an `in` channel (they have to, because the challenger just used an `out` channel).

Let us begin with the hash and corruption oracles, because they are shorter and have only one `in` channel.

- The three oracles `rom1_oracle`, `rom2_oracle`, and `rom3_oracle` permit the attacker to call the random oracles with its own values. We show the definition of `rom1_oracle`, the others are defined accordingly with more variables:

  ```
  1  param N_qH1 [noninteractive].
  2  channel ch1, ch2.
  3  let rom1_oracle(k: hashkey_t) =
  4          foreach iH <= N_qH1 do
  5      in(ch1, (x1: G_t, x2: G_t));
  6      out(ch2, hash(k, x1, x2)).
  ```

  The parameters `N_qH1`, `N_qH2`, and `N_qH3` are the number of calls the attacker is allowed to issue to each oracle. As all parameters in CryptoVerif, they implicitely are polynomial in the security parameter.

- The corruption oracles `corrupt_S_i`, `corrupt_S_r`, and `corrupt_psk` permit the attacker to get the respective keys. A variable is set and an event called. The variable allows to react to the compromise in the game: In certain compromise scenarios that would trivially break the protocol, we do not encrypt depending on the secret bit. The event allows to formulate correspondence queries; if in certain compromise scenarios we cannot prove a correspondence, we include the corruption event in an or clause. The corruption oracles are all defined similarly:

  ```
  1  let corrupt_S_i(S_i_priv: Z_t) =
  2    in(c_corrupt_S_i, ());
  3    let S_i_is_corrupted: bool = true in
  4    event S_i_corrupted;
  5    out(c_corrupt_S_i, (S_i_priv)).
  6
  7  let corrupt_S_r(S_r_priv: Z_t) =
  8    in(c_corrupt_S_r, ());
  ```

```
 9    let S_r_is_corrupted: bool = true in
10    event S_r_corrupted;
11    out(c_corrupt_S_r, (S_r_priv)).
12
13  let corrupt_psk(Q: psk_t) =
14    in(c_corrupt_psk, ());
15    let psk_is_corrupted: bool = true in
16    event psk_corrupted;
17    out(c_corrupt_psk, Q).
```

- We do not model the compromise of ephemeral keys dynamically via oracles. Instead, we hardcode the compromise of ephemeral keys statically in separate files. We create one file where E_i^{priv} is compromised, one file where E_r^{priv} is compromised, and another file where both of them are compromised. Compromise is done by sending the ephemeral private key along with the message in which the ephemeral public key is sent. The reason for this is that the size of games is increased by each additional compromise we allow; it at least doubles, because in each branch we potentially have to handle both the case that the ephemeral *is* compromised or *is not* compromised.

  The separation into different model files becomes clearer in Section 4.5.

- The initiator process is included from within the challenger top-level process with a tuple of arguments: As with the functions we defined earlier, it receives the keys for the hash function and random oracles. Then, it receives the initiator's long-term key pair, the responder's public key, the pre-shared key, and the secret bit. Inside, the initiator spawns a replication of processes:

  ```
  1  ! i_N_init_parties <= N_init_parties
  ```

  By this, we model that the initiator can execute the protocol in parallel and sequentially. `N_init_parties` is then the parameter that will appear in the probability formula as number of initiator sessions.

  The initiator process has multiple `in` channels, one for each type of protocol message it can receive, plus one channel that configures a new initiator session:

  ```
  1  in(c_config_initiator, (S_X_pub: G_t, timestamp_i: timestamp_t));
  ```

  Here, the attacker chooses with which long-term public key the initiator should start a new protocol session, and which timestamp it should use for the first protocol message. This corresponds to the Send message with special start symbol which is used in eCK-like models to set up a new session. The initiator process provides three further oracles. First, the one for the reception of the second protocol message:

  ```
  1  in(c_resp2init_recv, (plaintext_0: bitstring, plaintext_1: bitstring, ↩
         (=msg_type_resp2init, =reserved, I_r_recv: session_index_t, =I_i, ↩
         E_r_pub_recv: G_t, empty_bitstring_r_enc_recv: bitstring, ↩
         mac1_r_recv: mac_t, mac2_r_recv: mac_t)));
  ```

It receives all values from the second protocol message, and additionally the two plaintexts for the left-or-right message indistinguishability game that the initiator provides when sending the first transport data message (key confirmation). Second, in a process replication, an oracle that instructs the initiator to send a transport data message:

```
1  ! i_Nis<=N_init_send
2  in(c_N_init_send_config, (plaintext_data_0: bitstring, plaintext_data_1↩
       : bitstring, nonce: nonce_t));
```

We hereby model a number of `N_init_send` parallel or sequential transport data messages. And third, in a process replication, an oracle that lets the initiator receive a transport data message:

```
1  ! i_Nir<=N_init_recv
2  in(c_N_init_recv, (=msg_type_data, =reserved, =I_i, counter_recv: ↩
       counter_t, ciphertext_data_recv: bitstring));
```

We hereby model a number of `N_init_recv` parallel or sequential transport data messages.

With all these oracles, the attacker can test the reaction of the initiator to a protocol message, according to the protocol specification. We describe the details of the initiator process just in the next subsection.

- The responder process is included from within the challenger top-level process just as the initiator process. The only difference is that it receives `S_r_priv` as long-term private key. The responder equally replicates `N_resp_parties` processes, which will appear in the probability formula as number of responder sessions:

```
1  ! i_N_resp_parties <= N_resp_parties
```

The responder has no special activation channel, as a new responder session is spawned by receiving the first protocol message. The responder provides four oracles. First, the one for receiving the first protocol message:

```
1    in(c_init2resp_recv, (=msg_type_init2resp, =reserved, I_i_recv: ↩
         session_index_t, E_i_pub_recv: G_t, static_i_enc_recv: bitstring,↩
          timestamp_i_enc_recv: bitstring, mac1_i_recv: mac_t, mac2_i_recv↩
         : mac_t));
```

The parameters are those of the first protocol message defined in Section 3.2 without special instructions from the attacker. Second, the oracle for receiving the first transport data message (key confirmation):

```
1  in(c_keyconfirm_recv, (=msg_type_data, =reserved, =I_r, =counter_0, ↩
       ciphertext_keyconfirmation_recv: bitstring));
```

And finally the exactly same (modulo variable names) oracles for sending and receiving transport data messages:

```
1  ! i_Nrs<=N_resp_send
2  in(c_N_resp_send_config, (plaintext_0: bitstring, plaintext_1: ↩
       bitstring, nonce: nonce_t));
```

And the one for receiving:

```
1  ! i_Nrr<=N_resp_recv
2  in(c_N_resp_recv, (=msg_type_data, =reserved, =I_r, counter_recv: ↩
       counter_t, ciphertext_data_recv: bitstring));
```

This was the description of the setup of the game and the oracles the attacker can interact with. As a next step, we describe in detail how the initiator and responder oracles react according to the protocol.

The Initiator's Session Oracles.

- Upon reception of the configuration message, the initiator prepares and sends the first protocol message. It does so primarily by calling `prepare1`:

```
1  in(c_config_initiator, (S_X_pub: G_t, timestamp_i: timestamp_t));
2
3  let (I_i: session_index_t,
4       E_i_priv: Z_t, E_i_pub_foo: G_t, static_i_enc: bitstring,
5       timestamp_i_enc: bitstring, mac1_i: mac_t, mac2_i: mac_t,
6       es_i: G_t, ss_i: G_t, H_i4: hashoutput_t) =
7    prepare1(key_hash, key_rom1, key_rom2, S_X_pub, S_i_priv, S_i_pub, ↩
         timestamp_i) in
8
9  event sent1(E_i_pub_foo, static_i_enc, S_i_pub, timestamp_i_enc, ↩
       timestamp_i);
10 out(c_init2resp_send, (ifdef(‘E_i_compr’,‘E_i_priv’,‘dummy_z’), (↩
       msg_type_init2resp, reserved, I_i, E_i_pub_foo, static_i_enc, ↩
       timestamp_i_enc, mac1_i, mac2_i)));
```

In line 9, an event `sent1` is called with arguments all non-constant protocol fields of the first protocol message and the according plaintexts. Then, the first protocol message is output on a channel. Here, we see how we hardcode the compromise of the ephemeral key using the `m4` preprocessor. If we do not configure it with compromise of the ephemeral, only a placeholder value is sent.

- Upon reception of the second protocol message, the initiator first processes the message. It does so by calling the function `process2` defined earlier:

```
1      in(c_resp2init_recv, (plaintext_0: bitstring, plaintext_1: ↩
           bitstring, (=msg_type_resp2init, =reserved, I_r_recv: ↩
           session_index_t, =I_i, E_r_pub_recv: G_t, ↩
           empty_bitstring_r_enc_recv: bitstring, mac1_r_recv: mac_t, ↩
           mac2_r_recv: mac_t)));
2
```

```
3     let (continue: bool, T_i_send: key_t, T_i_recv: key_t) =
4       process2(key_hash, key_rom3, I_i, I_r_recv, E_i_priv, E_i_pub_foo↩
            , S_i_priv, S_i_pub, E_r_pub_recv, empty_bitstring_r_enc_recv↩
            , mac1_r_recv, mac2_r_recv, H_i4, es_i, ss_i, Q) in
5     if continue then
6
7     event rcvd2(if S_X_pub = S_r_pub then true else false, E_i_pub_foo,↩
          static_i_enc, S_i_pub, timestamp_i_enc, timestamp_i, ↩
          E_r_pub_recv, empty_bitstring_r_enc_recv, T_i_send, T_i_recv);
```

The function `process2` can return an error value in case the decryption of the payload fails. In this case the execution does not go beyond line 5. If decryption was successful, event `rcvd2` is issued. Its first parameter is a boolean that indicates if the initiator is talking to the honest responder in this session. In case it is not the honest responder, we do not want to prove a correspondence for `rcvd2` later. The other arguments are all non-constant variables of protocol messages one and two, as well as the derived transport data keys.

The initiator now proceeds to prepare the key confirmation message by calling the function `prepare3`:

```
1     let (continue3: bool, ciphertext_keyconfirmation: bitstring, ↩
          plaintext_keyconfirmation: bitstring) =
2       prepare3(CLEAN, secret_bit_I, plaintext_0, plaintext_1, (↩
            is_initiator, i_N_init_parties), T_i_send) in
3     if continue3 then
4
5     event sent3(if S_X_pub = S_r_pub then true else false, E_i_pub_foo,↩
          static_i_enc, S_i_pub, timestamp_i_enc, timestamp_i, ↩
          E_r_pub_recv, empty_bitstring_r_enc_recv, T_i_send, T_i_recv, ↩
          ciphertext_keyconfirmation, plaintext_keyconfirmation);
6
7     event initiator_keys(I_i, E_i_pub_foo, static_i_enc, ↩
          timestamp_i_enc, mac1_i, mac2_i, I_r_recv, E_r_pub_recv, ↩
          empty_bitstring_r_enc_recv, mac1_r_recv, mac2_r_recv, T_i_send,↩
           T_i_recv, S_i_pub, S_X_pub);
8
9     out(c_keyconfirm_send, (msg_type_data, reserved, I_r_recv, ↩
          counter_0, ciphertext_keyconfirmation));
```

The placeholder `CLEAN` stands for the cleanness predicate of the session that we will describe later. As described earlier, `prepare3` (and `prepare_msg`) will only encrypt depending on the secret bit, if the session is clean. If `prepare3` does return a ciphertext, the oracle continues, and issues two events before it outputs the message on a channel. The first event is `sent3`, which stands for the fact that the initiator sent the key confirmation message. As `rcvd2`, it takes as arguments all non-constant values that are part of the three protocol messages involved so far, the plaintext values belonging to ciphertexts sent, and the transport data keys. The first value

indicates if it is a session between initiator and responder, because for other sessions we do not want to prove authenticity properties. The event `initiator_keys` takes as parameters all non-constant values sent during the first two protocol messages, and additionally the transport data keys and the long-term public keys. This will be used to prove correctness and other correspondences.

- When the attacker asks the initator to send a transport data message, it prepares in using `prepare_msg`, using the same cleanness predicate as before:

```
1       ! i_Nis<=N_init_send
2       in(c_N_init_send_config, (plaintext_data_0: bitstring, ↩
            plaintext_data_1: bitstring, nonce: nonce_t));
3       let (continue_data_send: bool, ciphertext_data_send: bitstring, ↩
            plaintext_data_send: bitstring) = prepare_msg((is_initiator, ↩
            i_N_init_parties), secret_bit_I, plaintext_data_0, ↩
            plaintext_data_1, nonce, CLEAN, T_i_send) in
4       if continue_data_send then
5       event sent4_initiator(if S_X_pub = S_r_pub then true else false, ↩
            E_i_pub_foo, static_i_enc, S_i_pub, timestamp_i_enc, ↩
            timestamp_i, E_r_pub_recv, empty_bitstring_r_enc_recv, ↩
            T_i_send, T_i_recv, ciphertext_keyconfirmation, ↩
            plaintext_keyconfirmation, nonce_to_counter(nonce), ↩
            ciphertext_data_send, plaintext_data_send);
6       out(c_N_init_send, (msg_type_data, reserved, I_r_recv, ↩
            nonce_to_counter(nonce), ciphertext_data_send))
```

If `prepare_msg` does not fail, event `sent4_initiator` is issued with the same sort of parameters as `sent3`, and the transport data message ist sent.

- When the attacker sends the initiator a transport data message, it is processed using `process_msg`:

```
1       ! i_Nir<=N_init_recv
2       in(c_N_init_recv, (=msg_type_data, =reserved, =I_i, counter_recv:↩
             counter_t, ciphertext_data_recv: bitstring));
3       let (continue_data_recv: bool, plaintext_data_recv: bitstring) = ↩
            process_msg((is_initiator, i_N_init_parties), counter_recv, ↩
            ciphertext_data_recv, T_i_recv) in
4       if continue_data_recv then
5       event rcvd4_initiator(if S_X_pub = S_r_pub then true else false, ↩
            E_i_pub_foo, static_i_enc, S_i_pub, timestamp_i_enc, ↩
            timestamp_i, E_r_pub_recv, empty_bitstring_r_enc_recv, ↩
            T_i_send, T_i_recv, ciphertext_keyconfirmation, ↩
            plaintext_keyconfirmation, counter_recv, ciphertext_data_recv↩
            , plaintext_data_recv)
```

If decryption is successful, event `rcvd4_initiator` is issued.

The Responder's Session Oracles.

- Upon reception of a first protocol message, the responder processes it using the `process1` function.

```
in(c_init2resp_recv, (=msg_type_init2resp, =reserved, I_i_recv: session_index_t, E_i_pub_recv: G_t, static_i_enc_recv: bitstring, timestamp_i_enc_recv: bitstring, mac1_i_recv: mac_t, mac2_i_recv: mac_t));

  let (continue1: bool, es_r: G_t, ss_r: G_t, S_i_pub_recv: G_t, H_r4: hashoutput_t, timestamp_i_recv: timestamp_t) =
    process1(
      key_hash, key_rom1, key_rom2, S_r_priv, S_r_pub,
      I_i_recv, E_i_pub_recv, static_i_enc_recv, timestamp_i_enc_recv,
      mac1_i_recv, mac2_i_recv) in
  if continue1 then

  get rcvd_timestamps(=S_r_pub, =S_i_pub_recv, =timestamp_i_recv) in yield else
  insert rcvd_timestamps(S_r_pub, S_i_pub_recv, timestamp_i_recv);

  event rcvd1(if S_i_pub_recv = S_i_pub then true else false, E_i_pub_recv, static_i_enc_recv, S_i_pub_recv, timestamp_i_enc_recv, timestamp_i_recv);
```

If the processing was successful, that is the decryption of the timestamp, the process continues. If the timestamp has been seen before, the process yields, otherwise it stores the timestamp in the table and continues by issuing event `rcvd1`.

Then, for preparation of the second protocol message, `prepare2` is used. If this is successful, the event `sent2` is issued and the second protocol message is output on a channel.

```
  let (continue2: bool, I_r: session_index_t, E_r_priv: Z_t, E_r_pub_foo: G_t, T_r_recv: key_t, T_r_send: key_t, empty_bitstring_enc: bitstring, mac1_r: mac_t, mac2_r: mac_t) =
    prepare2(key_hash, key_rom3,
      I_i_recv, S_i_pub_recv, E_i_pub_recv, H_r4, es_r, ss_r, Q) in
  if continue2 then
  event sent2(E_i_pub_recv, static_i_enc_recv, S_i_pub_recv, timestamp_i_enc_recv, timestamp_i_recv, E_r_pub_foo, empty_bitstring_enc, T_r_recv, T_r_send);
  out(c_resp2init_send, (ifdef('E_r_compr','E_r_priv','dummy_z'), (msg_type_resp2init, reserved, I_r, I_i_recv, E_r_pub_foo, empty_bitstring_enc, mac1_r, mac2_r)));
```

Again, we have the optional static compromise of the ephemeral key along with the protocol message.

- Upon reception of the key confirmation message, it is processed using `process3`.

```
in(c_keyconfirm_recv, (=msg_type_data, =reserved, =I_r, =counter_0,↩
     ciphertext_keyconfirmation_recv: bitstring));
let (continue3: bool, plaintext_keyconfirmation_recv: bitstring) =
  process3(ciphertext_keyconfirmation_recv, T_r_recv, (is_responder↩
      , i_N_resp_parties)) in
if continue3 then

event rcvd3(if S_i_pub_recv = S_i_pub then true else false, ↩
    E_i_pub_recv, static_i_enc_recv, S_i_pub_recv, ↩
    timestamp_i_enc_recv, timestamp_i_recv, E_r_pub_foo, ↩
    empty_bitstring_enc, T_r_recv, T_r_send, ↩
    ciphertext_keyconfirmation_recv, plaintext_keyconfirmation_recv↩
    );
event responder_keys(I_i_recv, E_i_pub_recv, static_i_enc_recv, ↩
    timestamp_i_enc_recv, mac1_i_recv, mac2_i_recv, I_r, ↩
    E_r_pub_foo, empty_bitstring_enc, mac1_r, mac2_r, T_r_recv, ↩
    T_r_send, S_i_pub_recv, S_r_pub);

out(c_wait_before_2nd_part, ());
```

  If the message could be decrypted, event `rcvd3` is issued. Also, because the responder is now convinced that the initiator has derived the same keys, the event `responder_keys` is issued.

- The oracles for transport data messages are the same as for the initiator, just using the appropriate keys, and with adapted names for events. Thus we do not repeat the definition.

4.3.4. Trivial Attacks, Session Cleanness, and Partnering Definition

In eCK- and ACCE-like models, the attacker wins if it can, for a test session it choses, distinguish a random key from a real key, or the ciphertext of two different plaintexts. However, the test session must be "clean". If it is compromised in a way that would break security trivially, the attack does not count. The predicate clean also does not only concern the test session itself, but the partner session which was the communication partner in the protocol execution.

Test Session. In our model, all clean sessions are test sessions, and not just one chosen by the attacker. We model this by playing a left-or-right message indistinguishability game dependent on the secret bit in every session that is clean.

Session Cleanness. In WireGuard, four Diffie-Hellman operations and the pre-shared key contribute to the session key. If the pre-shared key is used and not compromised, the protocol is secure. If it is not used or compromised, security is based on the four Diffie-Hellman operations. If *one* of them cannot be computed by the attacker, then the

transport data keys are secure by Gap Diffie-Hellman. This means all combinations of compromises are fine *but* those where both keys on one side are compromised. We now describe how we model this, separately for each model file into which we split up the proof.

- **Dynamic Compromises of Long-Term Keys, Ephemerals Are *Not* Compromised.** Suppose we are on the side of the initiator. Its long-term key could possibly be compromised, but the ephemeral is not. Thus on the initiator's side, there is no problem. If the responder's long-term key is compromised, the session is still clean if the ephemeral is not compromised. Our assumption in this part of the proof is that ephemerals cannot be compromised, but if the responder's long-term key is compromised, the attacker can impersonate him. The initiator could thus talk to an attacker-led session. However, if the initiator is really talking to the responder, this session is clean. We can query this by a `find` condition on E_r^{pub}, see line 2:

  ```
  1  if defined(S_r_is_corrupted) then (
  2    find j <= N_resp_parties suchthat defined(E_r_pub[j]) && E_r_pub_recv↩
           = E_r_pub[j] then
  3      S_X_pub = S_r_pub
  4    else
  5      false
  6  ) else (
  7    S_X_pub = S_r_pub
  8  )
  ```

  If there is a responder session that generated this ephemeral, we consider the session to be clean and return $S_X^{pub} = S_r^{pub}$, which is the necessary condition for the message indistinguishability game to be played.

  The cleanness predicate on the side of the responder is determined equivalently.

- E_i^{priv} **or** E_r^{priv} **compromised.** In these models we hardcode the compromise of one of the ephemerals (for all sessions). If the ephemeral key of one side is compromised, its long-term key cannot be compromised. In these models we thus do not provide the respective oracle for dynamic compromise of the long-term key. The cleanness predicate is calculated in the same way as before: If the other side's long-term key is not compromised, the session is clean if the honest parties are talking to each other. If it is compromised, but the session is actually with the other side and not with the attacker, the session is clean if the honest parties are talking to each other.

- **Both Ephemerals Compromised.** In this case there is no dynamic compromise of long-term keys. All sessions are clean where the honest parties are talking to each other.

4.3.5. Security Queries

Secrecy. All secrecy properties are queried via

```
query secret secret_bit.
```

We described in Section 4.2.2 how CryptoVerif checks those. This includes the following properties:

- Message Secrecy in general (in sessions without compromises).
- Forward Secrecy, because the model allows long-term keys to be compromised after a session.
- Message Secrecy if the long-term keys are already compromised but the parties can rely on their not-compromised ephemerals.

Correctness. We prove that, if two parties have the same view on a protocol transcript, they derive the same transport data keys. In our case, all variables used in queries are universally quantified.

```
event(responder_keys(
      I_i, E_i_pub, static_i_enc, timestamp_i_enc, mac1_i, mac2_i,
      I_r, E_r_pub, empty_bitstring_enc, mac1_r, mac2_r,
      T_r_recv, T_r_send, S_i_pub, S_r_pub))
&&
event(initiator_keys(
      I_i, E_i_pub, static_i_enc, timestamp_i_enc, mac1_i, mac2_i,
      I_r, E_r_pub, empty_bitstring_enc, mac1_r, mac2_r,
      T_i_send, T_i_recv, S_i_pub, S_r_pub))
==> (T_i_send = T_r_recv && T_i_recv = T_r_send).
```

These events are issued after the derivation of the transport data keys on both sides. Formally, we prove with this query that if an initiator and an responder have the same protocol transcript (that is, the values that they sent respectively received for each protocol message) and derived possibly different transport data keys, the transport data keys are actually the same. This means the protocol works.

Unknown Key-Share Attack. We prove resistance against unilateral and bilateral unknown key-share attacks, by proving that if two parties have derived the same transport data keys, they have the same view on the keys used for the derivation:

```
event(responder_keys(
          I_i_recv, E_i_pub_recv, static_i_enc_recv, timestamp_i_enc_recv, ↩
              mac1_i_recv, mac2_i_recv,
          I_r, E_r_pub, empty_bitstring_enc, mac1_r, mac2_r,
          T_i_send, T_i_recv, S_i_pub_recv, S_r_pub))
&&
event(initiator_keys(
          I_i, E_i_pub, static_i_enc, timestamp_i_enc, mac1_i, mac2_i,
```

```
8             I_r_recv, E_r_pub_recv, empty_bitstring_enc_recv, mac1_r_recv, ↩
                  mac2_r_recv,
9             T_i_send, T_i_recv, S_i_pub, S_r_pub_recv))
10    ==> (E_i_pub = E_i_pub_recv && E_r_pub = E_r_pub_recv && S_i_pub = ↩
          S_i_pub_recv && S_r_pub = S_r_pub_recv).
```

Formally, if an initiator and a responder derive the same transport data keys, with possibly different keys $E_i^{pub}, E_{i,recv}^{pub}$, $E_r^{pub}, E_{r,recv}^{pub}$, $S_i^{pub}, S_{i,recv}^{pub}$, $S_r^{pub}, S_{r,recv}^{pub}$, the respective keys are actually the same.

Authentication of the Second Protocol Message. We prove that if the initiator thinks to have received a second protocol message from the responder, then the responder actually sent this message. More strongly, we prove that each reception of a second protocol message by the initiator implies that the responder has a distinct partner session and sent the second protocol message. We do so by the following correspondence query with injective events:

```
1    inj-event(rcvd2(true, E_i_pub, static_i_enc, S_i_pub, timestamp_i_enc, ↩
         timestamp_i, E_r_pub, empty_bitstring_enc, T_i_send, T_i_recv)) ==>
2    inj-event(sent2(      E_i_pub, static_i_enc, S_i_pub, timestamp_i_enc, ↩
         timestamp_i, E_r_pub, empty_bitstring_enc, T_i_send, T_i_recv))
3    || event(S_r_corrupted).
```

We recall that the first argument of `rcvd2` is a boolean indicating if the initiator thinks to talk to the responder. Only in this case, proving authentication is interesting. The partner session here is a session with the same view on the protocol messages that were sent over channels, and the respective plaintexts. Note that the implication is that either `sent2` occurred, or `S_r_corrupted`. If the responder's long-term key is compromised, the attacker can impersonate him and there is no authentication. However, this query does not depend on the fact if the initiator's long-term key is compromised. This means that this authentication query includes resistance against key compromise impersonation attacks.

Authentication of the Key Confirmation Message. Equally, we prove that each reception of a key confirmation message implies that the initiator has a partner session that sent this message, and distinct receptions correspond to distinct partner sessions. If the initiator's long-term key is compromised, then we do not prove this property. This query does not depend on a possible compromise of the responder's long-term key and thus includes resistance against KCI.

```
1    inj-event(rcvd3(true, E_i_pub, static_i_enc, S_i_pub, timestamp_i_enc, ↩
         timestamp_i,  E_r_pub, empty_bitstring_enc, T_r_recv,  T_r_send,  ↩
         ciphertext_keyconfirmation, plaintext_keyconfirmation )) ==>
2    inj-event(sent3(true, E_i_pub, static_i_enc, S_i_pub, timestamp_i_enc, ↩
         timestamp_i2, E_r_pub, empty_bitstring_enc, T_r_recv2, T_r_send2, ↩
         ciphertext_keyconfirmation, plaintext_keyconfirmation2))
3    || event(S_i_corrupted).
```

Authentication of Transport Data Messages. For transport data messages in both directions, we prove that if such a message is received, then there is a partner session of the other peer that sent this message. Resistance against KCI is handled in the same way. The CryptoVerif code looks very similar to the query for the key confirmation, and thus we do not include it. One difference is that the nonces are included in the event's parameters.

Mutual Authentication. Looking at the authentication of the second message and authentication of the key confirmation message together, we have prove that the protocol provides mutual authentication of initiator and responder.

No Replay of the First Protocol Message. In presence of the table of already seen timestamps, we can prove that the first protocol message cannot be replayed:

```
1    inj-event(rcvd1(true, E_i_pub, static_i_enc, S_i_pub, timestamp_i_enc, ↩
         timestamp_i)) ==>
2    inj-event(sent1(E_i_pub, static_i_enc, S_i_pub, timestamp_i_enc, ↩
         timestamp_i))
3    || event(S_i_corrupted) || event(S_r_corrupted).
```

We prove that for each received first protocol message for which the responder thinks it comes from the responder, the responder has a partner session and sent the first protocol message with the same contents. However, we can prove this only if neither of the two long-term keys is compromised. If S_i^{priv} is compromised, the attacker can impersonate the initiator. If S_r^{priv} is compromised, the attacker can impersonate the initiator to the responder, because resistance against KCI is established only after the key confirmation message.

Note that neither of the correspondence queries depend on a possible corruption of the ephemeral keys or the pre-shared key. This is clear because those are all authentication-related properties, and authentication in this protocol is based on the long-term keys.

4.4. Description of the Proof

We guide the proofs by manually indicating the transformations to apply. This is necessary because we need to disect different protocol execution scenarios (honest parties talking, honest and dishonest party talking), and especially use some transformation that CryptoVerif does not apply automatically, like the splitting up of the result of the last random oracle, and INT-CTXT in case of dynamic compromise. In the following, we detail the proof steps for the most complex of the model files, the one with possible dynamic compromises of the honest party's long-term keys. The proofs for the other model files are very similar, respectively slightly shorter, and can be found in Appendix A.1.

The proofs steps are indicated to CryptoVerif in the `proof` environment, usually at the beginning of the file, and in any case before the top-level process. At the beginning, we introduce case distinctions on the long-term and ephemeral keys. Our (manual) proof strategy here is to isolate the sessions between two honest parties, prove security there,

and in the other cases prove that the protocol will either not complete or the games do not use encryption dependent on the secret bit.

As a first step, we introduce a case distinction in the initiator process, to be able to separately treat the case where it is talking to the honest responder or another party. This is done by an `insert` statement:

```
1  proof {
2    success;
3    insert 38 "if S_X_pub = S_r_pub then";
4    SArename E_i_pub_2;
```

The number 38 here is a so called "occurrence" number. CryptoVerif numbers all terms starting from the beginning of the game, and the number in the insert statement indicates at which occurrence number the term should be inserted. The term that resided at this occurrence number before will then be placed after the newly inserted statement. Inserting an `if` statement will duplicate all code below for both branches. We use the `SArename` command to introduce different names for the initiator's ephemeral public key for the two cases. We did not comment the first line in the proof environment so far: The command `success` instructs CryptoVerif to check if the queries are satisfied. Directly at the beginning, the correspondence for correctness of the protocol is already proved. Equally to above, we introduce a case distinction on the responder's side, branching on the long-term public key it is talking to:

```
5    insert 2791 "if S_i_pub_recv_2 = S_i_pub_1 then";
```

We also distinguish the different possible cases of used ephemeral keys. In the responder process, we find the initiator session that generated an honest ephemeral key, in the belief of creating the protocol message for the honest responder long-term key.

```
6    (* just after the first in at the responder's side. *)
7    insert 2726 "find i <= N_init_parties suchthat defined(E_i_pub_4[i]) && ↩
         E_i_pub_recv_2 = E_i_pub_4[i] then";
```

This case distinction is inserted just after the first protocol message is received. `E_i_pub_4` is the variable name of the initiator's ephemeral key, created when preparing a message to the honest responder long-term key. Similarily, on the initiator's side, we find the responder session that created their ephemeral key for a message to the honest initiator long-term key. This case distinction needs to be made after the initiator receives the responder's ephemeral key with the second protocol message. In the initiator process, there are *two* places where this can happen because of the above case distinction, (1) when talking to the honest responder long-term key, (2) when talking to another long-term key.

```
8    insert 1655 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
         E_r_pub_recv_1 = E_r_pub[j] then";
9    insert 179 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
         E_r_pub_recv_1 = E_r_pub[j] then";
```

Now we start the cryptographic transformations. We first transform the calls to random oracles to find constructs.

```
10   crypto rom(rom3_intermediate);
```

```
11    crypto rom(rom2);
12    crypto rom(rom1);
```

Then we apply the Gap Diffie-Hellman assumption on the honest private keys.

```
13    crypto gdh(exp) S_r_priv_1 E_i_priv_5 S_i_priv_2 E_r_priv_2;
```

This will eliminate a lot of branches because we proved that the attacker cannot know certain results of Diffie-Hellman operations. Indeed, after splitting the result of the third random oracle into the individual keys, CryptoVerif is able to prove the first correspondence property: resistance against unknown key-share attacks.

```
14    crypto split_hashoutput *;
15    success; (* unknown key-share *)
```

As a next step, we let CryptoVerif apply INT-CTXT at all places where it can.

```
16    crypto int_ctxt(enc) *;
17    success; (* some nonce reuse, correctness *)
```

After this step, it is proven for some nonce reuse failure events that they do not occur. Next, we let CryptoVerif apply IND-CPA where it can.

```
1     crypto ind_cpa(enc) *;
2     success; (* more nonce reuse proved *)
```

If there were no dynamic compromises, the security properties could now all be proved. Indeed, in the model file where the pre-shared key is used and all Diffie-Hellman keys are statically compromised, the proof succeeds here.

In case of possible dynamic compromises, more manual transformations need to be applied. Suppose the initiator talks to the honest responder long-term key, and the latter is not compromised. The ephemeral that the initiator received with the second protocol message however is *not* an honest ephemeral (but was replaced by some other public key by the attacker). With the following application of INT-CTXT we establish that the attacker cannot forge the ciphertext in the second protocol message: It might have chosen an ephemeral that it has the private key for, but it does not know the private key of the honest responder long-term key.

```
18    crypto int_ctxt_corrupt(enc) k_19;
19    success;
```

At this stage, the two correspondences on protocol messages received by the initiator can be proved: The initiator can authenticate the second protocol message sent by the responder, and the initiator can authenticate transport data messages sent by the responder.

An analoguous case needs to be treated on the responder's side. Suppose it talks to the honest initiator long-term key, and the latter is not compromised. The ephemeral received with the first protocol message however was not honest. Then the attacker cannot forge the ciphertext of the key confirmation message.

```
20    crypto int_ctxt_corrupt(enc) T_i_send_12;
21    success;
```

At this stage, the correspondences on protocol messages received by the responder can be proved: The responder can authenticate transport data messages sent by the initiator, and the responder can authenticate the third protocol message sent by the initiator (key confirmation).

The only query left is the secrecy of the secret bit. It is proved after some applications of IND-CPA that were blocked by the dynamic compromises.

```
22    crypto ind_cpa(enc) *;
23    success; (* secrecy of secret_bit (message indistinguishability) *)
```

The proof concludes after 204 game transformations in game 205. This might seem a lot, but counts all syntactical transformations.

4.5. Results and Discussion of the Model

We prove all properties described in Section 4.3.5, and do so by separating the proof into different compromise scenarios. For every scenario, we prepare a separate model file and run it in CryptoVerif.

1. Dynamic compromises of long-term keys, ephemerals are not compromised. The pre-shared key might be used or not, and dynamic compromise of it is possible.
2. E_i^{priv} statically compromised, and possible dynamic compromise of S_r^{priv}. The pre-shared key might be used or not, and dynamic compromise of it is possible.
3. E_r^{priv} statically compromised, and possible dynamic compromise of S_i^{priv}. The pre-shared key might be used or not, and dynamic compromise of it is possible.
4. Both E_i^{priv} and E_r^{priv} are statically compromised. The pre-shared key might be used or not, and dynamic compromise of it is possible.
5. All Diffie-Hellman keys are statically compromised, the pre-shared key is used and not compromised.
6. We have one additional model file in which we only prove the correspondence property on the first protocol message in the absence of long-term key compromise. Including this query in the model file of the first scenario led to too many case distinctions and CryptoVerif did not terminate.

We have proofs for all these scenarios (but the one with static compromise of both ephemerals, which we could not finish in time). This means that asymptotic security in the computational model is established in these cases. For the first scenario, we will present the advantage of the attacker to break the secrecy property. This property is proved in the last game, and thus the advantage is larger than for the other properties, and we can use it to bound the overall advantage for this scenario. A bound for the total advantage of the attacker to break any of the security properties of the complete WireGuard handshake can be obtained by the maximum of the advantages from scenario 1 to 5: A session is in *one* of the possible compromise scenarios, and the model files treat disjoint compromise scenarios.

The advantage of the attacker to break the no-replay property on the first message does not have to be consider as part of the advantage to break the whole handshake. We consider it as an additional property we proved.

In the following advantage formula, we use the advantages defined in Section 2.1. The variables n_i and n_r denote the number of initiator and responder session respectively, $n_{rom1}, n_{rom2}, n_{rom3}$ are the number of calls the attacker makes to the random oracles, $|\mathtt{Z\_t}|$ is the size of the space of exponents for the used Diffie-Hellman group, and $|\mathtt{psk\_t}|$ is the size of the pre-shared key space. The advantage of the attacker to calculate the secret bit in the first compromise scenario is

$$\begin{aligned}
&(14n_r + 10n_i)\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathsf{AEAD},\mathcal{A}}(n) + 24\mathsf{Adv}^{\mathrm{coll-res}}_{\mathsf{HASH},\mathcal{A}}(n) + 12n_r\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathsf{AEAD},\mathcal{A}}(n)\\
&+(4n_r + 6n_{rom1}n_i + 12n_{rom2}n_i + 2n_{rom3}n_r + 50n_in_i + 16n_rn_r + 2062n_rn_i + 50n_i)/|\mathtt{Z_t}|\\
&+(4n_{rom1} + 8n_{rom2} + 30n_{rom3} + 48n_rn_{rom3} + 2n_rn_{rom1} + 18n_rn_{rom2}) \cdot \mathsf{Adv}^{\mathrm{gdh}}_{\mathcal{G},\mathcal{A}}(n)
\end{aligned}$$

We simplified the formula given by CryptoVerif, mainly by removing the parameters of the advantages to break a cryptographic primitive. CryptoVerif indicates them dependent on the time and parameter sizes (additionally to the implicit dependency on the security parameter). We observe that the advantage formula is more detailed as the formula given in the theorem of [31]: Our formula includes the number of calls to the random oracles (and potentially the execution time of the attacker, and parameters of the primitives). Also, our formula considers collisions on the private Diffie-Hellman exponents. However, the constants are generally larger. This is because CryptoVerif applies the cryptographic transformations for a lot of subcases, without considering which of them are disjoint subcases. By taking the maximum of the attacker's advantage in disjoint subcases, a lower bound can be computed.

The advantages to break the secrecy of the secret bit in the other compromise scenarios is given in Appendix A.2. For a complete advantage formula, the probabilities from the indifferentiability proof, see Section 4.3.1.1, need to be *added* to the result from CryptoVerif.

Discussion of the Model. We comment briefly on some decisions made for our model.

We remind that we model *two* honest parties instead of a variable number as Dowling and Paterson do in their analysis. Thus, this parameter, n_P in their case, does not appear in our formulas. This does not degrade our analysis, because our model includes that the two honest parties can execute the protocol with dishonest parties (that is, the attacker): The attacker can explicitely set up an initiator process to start a protocol session to a long-term key that is not the honest responder's key. Also, the attacker can send a first protocol message to the responder, using any long-term public key it generated. Thus, as do Dowling and Paterson, we also include that the honest party's long-term keys are used in sessions with dishonest parties (which means that there is no partner session).

In our model, the two honest long-term keys have fixed roles: S_i is only used by a party initiating sessions, and S_r is only used by a party responding to sessions. Intuitively, one might think that this does reflect VPN scenarios, where it is usually a VPN client that initiates a secure connection to a VPN server. However this is not true in WireGuard: In some circumstances, it might be the responder who initiates a new handshake during a

longer run VPN session. Changing our model to allow the long-term keys to be used for both types of sessions would be straightforward: We could let the attacker give a bit to the initiator process to define which of the two honest private keys it should use. For the responder's side, we would call the responder process twice in the top-level process, once with each long-term key pair. However, the proof would need more effort. We would need to deal with a party talking to itself, that is if the attacker instructs the initiator to use the same key for itself and the party it's talking to. Cryptographically, we would need to introduce the square Diffie-Hellman assumption. However, the size of the games would at least double because of the case distinctions, and we expect that this exceeds the capabilities of CryptoVerif on current computers.

5. Conclusion and Future Work

We presented the first mechanised proof of the WireGuard protocol in the computational model. In Chapter 2, we defined the cryptographic primitives used in the protocol, the cryptographic assumptions used in our proof, and the notion of authenticated key exchange including its security properties. In Chapter 3, we briefly described the Noise Protocol Framework upon which WireGuard bases its key exchange protocol, and finally the WireGuard protocol itself. Chapter 4 contains our contribution. We gave an introduction to the game-hopping proof technique in Section 4.1 and the CryptoVerif proof assistant, which produces proofs using this technique, in Section 4.2. We described how we model WireGuard and the security properties we prove in Section 4.3. Our description of the execution environment and the oracles accessible by the attacker follows how this is usually described for eCK and ACCE models. In Section 4.4, we described the proof done in CryptoVerif, and discussed the results in Section 4.5, indicating an asymptotic bound of security. The security properties proved were detailed in Section 4.3.5, and we repeat here that we proved the following properties for the WireGuard protocol: Correctness, message secrecy, forward secrecy, mutual authentication, resistance against key compromise impersonation, resistance against unknown key-share attacks, and resistance against replay of the first protocol message. The latter two properties have not been considered by Dowling and Paterson, who provided the first (non mechanised) computational analysis of the WireGuard protocol [31]. Hereby, we prove all secrecy and authentication properties WireGuard claims to guarantee.

In the spirit of the paper [19] by Cohn-Gordon et al., we want to clearly discuss the scope of our proof, that is, what are our assumptions, and what is our trusted computing base. We use a proof assistant, which makes our proof rely on its correctness. The correctness of CryptoVerif's game transformations has been proved manually. A formal link between these proofs and the code is not established. Also, since the manual proofs have been originally made, changes have been made to CryptoVerif's code base. A comprehensive test suite is used to detect regressions, containing both queries that should be and should not be provable.

Our proof cannot make any statement about security guarantees of an actual WireGuard implementation, and this is left for future work as automated tools for program synthesis from formally verified proofs mature.

Of course it is left to mention that we also rely on the Gap Diffie-Hellman assumption, the security of the symmetric encryption scheme, and work in the Random Oracle model.

Future Work. Future work on this topic naturally divides into three different directions. A short-term goal is to extend our proof to include more properties, like identity hiding and session uniqueness. Those have been proven in the symbolic model with Tamarin [30]. The time frame of a master's thesis was not sufficient to include all desired properties. Also, it would be interesting to calculate an exact attacker advantage by instantiating our proof's result for WireGuard's particular parameters.

Another short-term goal is the proof of different Noise patterns. Of particular interest would be the Noise Pipes patterns, because they are used in WhatsApp to secure client-server communication. The model we prepared for WireGuard's IKpsk2 is easily adaptable to other patterns. However, we expect the proofs for each Noise patterns to require individual, manual work, because of the detailed guidance CryptoVerif needs for case distinctions.

A mid-term goal is to provide a verified implementation of WireGuard's protocol, which is proven to be functionally correct (that is, follow's the protocol's specification), to be memory safe, and to be free of side channels. This work will be based on the HaCl* formally verified crypto library.

A long-term goal is to guarantee the cryptographic properties we proved with CryptoVerif for this implementation. To establish this link, the semantics of CryptoVerif and the F* programming language need to be formally related, such that finally automatic code generation from one language to the other is possible.

Bibliography

[1] Jean-Philippe Aumasson et al. "BLAKE2: Simpler, Smaller, Fast as MD5". In: *Applied Cryptography and Network Security*. International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, June 25, 2013, pp. 119–135. ISBN: 978-3-642-38979-5 978-3-642-38980-1. DOI: 10.1007/978-3-642-38980-1_8. URL: https://link.springer.com/chapter/10.1007/978-3-642-38980-1_8 (visited on 05/07/2018) (cit. on p. 42).

[2] Gilles Barthe et al. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Advances in Cryptology – CRYPTO 2011*. Annual Cryptology Conference. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Aug. 14, 2011, pp. 71–90. ISBN: 978-3-642-22791-2 978-3-642-22792-9. DOI: 10.1007/978-3-642-22792-9_5. URL: https://link.springer.com/chapter/10.1007/978-3-642-22792-9_5 (visited on 05/12/2018) (cit. on p. 3).

[3] Gilles Barthe et al. "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII*. Ed. by Alessandro Aldini, Javier Lopez, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 146–166. URL: https://doi.org/10.1007/978-3-319-10082-1_6 (cit. on pp. 3, 26).

[4] Mihir Bellare and Chanathip Namprempre. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm". In: *Advances in Cryptology — ASIACRYPT 2000*. International Conference on the Theory and Application of Cryptology and Information Security. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Dec. 3, 2000, pp. 531–545. ISBN: 978-3-540-41404-9 978-3-540-44448-0. DOI: 10.1007/3-540-44448-3_41. URL: https://link.springer.com/chapter/10.1007/3-540-44448-3_41 (visited on 05/16/2018) (cit. on pp. 8, 9).

[5] Mihir Bellare and Chanathip Namprempre. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*. July 14, 2007. URL: https://cseweb.ucsd.edu/~mihir/papers/oem.pdf (visited on 05/16/2018) (cit. on pp. 8, 9).

[6] Mihir Bellare and Phillip Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*. https://eprint.iacr.org/2004/331. 2004. URL: https://eprint.iacr.org/2004/331 (cit. on p. 3).

[7] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Advances in Cryptology – Eurocrypt 2006 Proceedings*. Ed. by S. Vaudenay. Vol. 4004. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, May 2006, pp. 409–426 (cit. on p. 4).

[8] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *IEEE Symposium on Security and Privacy (S&P'17)*. Los Alamitos, CA: IEEE Computer Society Press, May 2017, pp. 483–503. URL: https://doi.org/10.1109/SP.2017.26 (cit. on p. 4).

[9] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-Bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: ACM, 2016, pp. 456–467. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978423. URL: http://doi.acm.org/10.1145/2976749.2978423 (visited on 05/11/2018) (cit. on p. 1).

[10] Bruno Blanchet. "A Computationally Sound Mechanized Prover for Security Protocols". In: *IEEE Symposium on Security and Privacy*. May 2006, pp. 140–154. URL: https://doi.org/10.1109/TDSC.2007.1005 (cit. on p. 3).

[11] Bruno Blanchet. "Computationally Sound Mechanized Proofs of Correspondence Assertions". In: *20th IEEE Computer Security Foundations Symposium (CSF'07)*. Los Alamitos, CA: IEEE Computer Society Press, July 2007, pp. 97–111. DOI: 10.1109/CSF.2007.16. URL: https://eprint.iacr.org/2007/128 (cit. on pp. 30, 31).

[12] Bruno Blanchet. *CryptoVerif: A Computationally-Sound Security Protocol Verifier*. Nov. 20, 2017. URL: CryptoVerif:%20A%20Computationally-Sound%20Security%20Protocol%20Verifier (cit. on pp. 27, 29).

[13] Michael Burrows, Martín Abadi, and Roger Needham. "A Logic of Authentication". In: *Proceedings of the Royal Society of London A* 426.1871 (Dec. 1989), pp. 233–271. DOI: https://doi.org/10.1145/77648.77649 (cit. on p. 3).

[14] Ran Canetti, Oded Goldreich, and Shai Halevi. "The Random Oracle Methodology, Revisited". In: *Journal of the ACM* 51.4 (July 2004), pp. 557–594 (cit. on p. 10).

[15] Ran Canetti and Hugo Krawczyk. "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels". In: *Advances in Cryptology — EUROCRYPT 2001*. International Conference on the Theory and Applications of Cryptographic Techniques. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, May 6, 2001, pp. 453–474. ISBN: 978-3-540-42070-5 978-3-540-44987-4. DOI: 10.1007/3-540-44987-6_28. URL: https://link.springer.com/chapter/10.1007/3-540-44987-6_28 (visited on 05/10/2018) (cit. on p. 2).

[16] Ran Canetti and Hugo Krawczyk. "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels". In: *Advances in Cryptology - EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, May 2001, pp. 453–474 (cit. on p. 14).

[17] Donghoon Chang, Mridul Nandi, and Moti Yung. *Indifferentiability of the Hash Algorithm BLAKE*. 623. 2011. URL: https://eprint.iacr.org/2011/623 (cit. on p. 42).

[18] Liqun Chen and Qiang Tang. "Bilateral Unknown Key-Share Attacks in Key Agreement Protocols". In: *Journal of Universal Computer Science* 14.3 (Feb. 2008), pp. 416–440 (cit. on p. 15).

[19] Katriel Cohn-Gordon and Cas Cremers. *Mind the Gap: Where Provable Security and Real-World Messaging Don't Quite Meet*. 982. 2017. URL: https://eprint.iacr.org/2017/982 (visited on 05/17/2018) (cit. on p. 85).

[20] Jean-Sébastien Coron et al. "Merkle-Damgård Revisited: How to Construct a Hash Function". In: *Advances in Cryptology—CRYPTO 2005*. Vol. 3621. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 430–448. URL: https://doi.org/10.1007/11535218_26 (cit. on p. 39).

[21] Cas Cremers. "Examining Indistinguishability-Based Security Models for Key Exchange Protocols: The Case of CK, CK-HMQV, and eCK". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 80–91. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966925. URL: http://doi.acm.org/10.1145/1966913.1966925 (visited on 05/11/2018) (cit. on p. 14).

[22] Cas Cremers. "Key Exchange in IPsec Revisited: Formal Analysis of IKEv1 and IKEv2". In: *16th European Conference on Research in Computer Security (ESORICS'11)*. Ed. by Vijay Atluri and Claudia Diaz. Vol. 6879. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Sept. 2011, pp. 315–334 (cit. on p. 1).

[23] Cas Cremers and Michèle Feltz. "Beyond eCK: Perfect Forward Secrecy under Actor Compromise and Ephemeral-Key Reveal". In: *Designs, Codes and Cryptography* 74.1 (Jan. 1, 2015), pp. 183–218. ISSN: 0925-1022, 1573-7586. DOI: 10.1007/s10623-013-9852-1. URL: https://link.springer.com/article/10.1007/s10623-013-9852-1 (visited on 05/10/2018) (cit. on pp. 2, 14).

[24] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* IT-22.6 (Nov. 1976), pp. 644–654 (cit. on p. 10).

[25] Yevgeniy Dodis et al. "To Hash or Not to Hash Again? (In)Differentiability Results for $$Ĥ2$$ and HMAC". In: *Advances in Cryptology – CRYPTO 2012*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2012, pp. 348–366. ISBN: 978-3-642-32008-8 978-3-642-32009-5. DOI: 10.1007/978-3-642-32009-5_21. URL: https://link.springer.com/chapter/10.1007/978-3-642-32009-5_21 (visited on 05/08/2018) (cit. on p. 42).

[26] Yevgeniy Dodis et al. *To Hash or Not to Hash Again? (In)Differentiability Results for $Ĥ2$ and HMAC*. 382. Full version. 2013. URL: https://eprint.iacr.org/2013/382 (visited on 05/09/2018) (cit. on p. 42).

[27] Danny Dolev and Andrew C. Yao. "On the Security of Public Key Protocols". In: *IEEE Transactions on Information Theory* IT-29.12 (Mar. 1983), pp. 198–208. URL: https://doi.org/10.1109/TIT.1983.1056650 (cit. on p. 2).

[28] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017. San Diego, California, USA, Feb. 27, 2017. DOI: http://dx.doi.org/10.14722/ndss.2017.23160 (cit. on p. 1).

[29] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel*. Whitepaper. May 9, 2018. URL: https://www.wireguard.com/papers/wireguard.pdf (visited on 05/10/2018) (cit. on pp. 1, 19).

[30] Jason A. Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. Jan. 21, 2018. URL: https://www.wireguard.com/papers/wireguard-formal-verification.pdf (visited on 05/10/2018) (cit. on pp. 2, 86).

[31] Benjamin Dowling and Kenneth G. Paterson. *A Cryptographic Analysis of the WireGuard Protocol*. 080. 2018. URL: https://eprint.iacr.org/2018/080 (visited on 05/08/2018) (cit. on pp. 2, 3, 83, 85).

[32] Niels Ferguson and Bruce Schneier. *A Cryptographic Evaluation of IPsec*. Counterpane Internet Security, Inc., 1999. URL: https://www.schneier.com/academic/paperfiles/paper-ipsec.pdf (cit. on p. 1).

[33] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: *Journal of Computer and System Sciences* 28 (1984), pp. 270–299 (cit. on p. 2).

[34] Thomas C. Hales. *The NSA Back Door to NIST*. Volume 61, Number 2. 2014, pp. 190–192. URL: http://www.ams.org/notices/201402/rnoti-p190.pdf (cit. on p. 3).

[35] Shai Halevi. *A Plausible Approach to Computer-Aided Cryptographic Proofs*. 181. 2005. URL: https://eprint.iacr.org/2005/181 (visited on 05/12/2018) (cit. on p. 4).

[36] Tibor Jager et al. "On the Security of TLS-DHE in the Standard Model". In: *Advances in Cryptology – CRYPTO 2012*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2012, pp. 273–293. ISBN: 978-3-642-32008-8 978-3-642-32009-5. DOI: 10.1007/978-3-642-32009-5_17. URL: https://link.springer.com/chapter/10.1007/978-3-642-32009-5_17 (visited on 05/09/2018) (cit. on p. 3).

[37] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. 2nd. Chapman & Hall/CRC, 2014. ISBN: 978-1-4665-7026-9 (cit. on p. 5).

[38] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*. Los Alamitos, CA: IEEE Computer Society Press, Apr. 2017, pp. 435–450 (cit. on p. 4).

[39] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. 2017, pp. 435–450. DOI: 10.1109/EuroSP.2017.38. URL: https://doi.org/10.1109/EuroSP.2017.38 (cit. on pp. 39, 40, 48, 54).

[40] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". In: *Advances in Cryptology – CRYPTO 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648. ISBN: 978-3-642-14623-7. URL: https://doi.org/10.1007/978-3-642-14623-7_34 (cit. on pp. 8, 42).

[41] Hugo Krawczyk. “HMQV: A High-Performance Secure Diffie-Hellman Protocol”. In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Aug. 2005, pp. 546–566. DOI: 10.1007/11535218_33. URL: https://link.springer.com/chapter/10.1007/11535218_33 (cit. on p. 15).

[42] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. “Stronger Security of Authenticated Key Exchange”. In: *Provable Security*. International Conference on Provable Security. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Nov. 1, 2007, pp. 1–16. ISBN: 978-3-540-75669-9 978-3-540-75670-5. DOI: 10.1007/978-3-540-75670-5_1. URL: https://link.springer.com/chapter/10.1007/978-3-540-75670-5_1 (visited on 05/10/2018) (cit. on pp. 2, 14).

[43] Adam Langley and Yoav Nir. “ChaCha20 and Poly1305 for IETF Protocols”. May 2015. URL: https://tools.ietf.org/html/rfc7539 (cit. on p. 20).

[44] Gavin Lowe. “An Attack on the Needham-Schroeder Public-Key Authentication Protocol”. In: *Information Processing Letters* 56.3 (Nov. 10, 1995), pp. 131–133. ISSN: 0020-0190. DOI: 10.1016/0020-0190(95)00144-2. URL: http://www.sciencedirect.com/science/article/pii/0020019095001442 (visited on 05/11/2018) (cit. on p. 3).

[45] Gavin Lowe. “Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar. 27, 1996, pp. 147–166. ISBN: 978-3-540-61042-7 978-3-540-49874-2. DOI: 10.1007/3-540-61042-1_43. URL: https://link.springer.com/chapter/10.1007/3-540-61042-1_43 (visited on 05/11/2018) (cit. on p. 3).

[46] Atul Luykx, Bart Mennink, and Samuel Neves. “Security Analysis of BLAKE2’s Modes of Operation”. In: *IACR Transactions on Symmetric Cryptology* 2016.1 (Dec. 1, 2016), pp. 158–176. ISSN: 2519-173X. DOI: 10.13154/tosc.v2016.i1.158-176. URL: https://tosc.iacr.org/index.php/ToSC/article/view/540 (visited on 05/07/2018) (cit. on p. 42).

[47] Roger M. Needham and Michael D. Schroeder. “Using Encryption for Authentication in Large Networks of Computers”. In: *Commun. ACM* 21.12 (Dec. 1978), pp. 993–999. ISSN: 0001-0782. DOI: 10.1145/359657.359659. URL: http://doi.acm.org/10.1145/359657.359659 (visited on 05/11/2018) (cit. on p. 3).

[48] Tatsuaki Okamoto and David Pointcheval. “The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes”. In: *Public Key Cryptography*. International Workshop on Public Key Cryptography. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Feb. 13, 2001, pp. 104–118. ISBN: 978-3-540-41658-6 978-3-540-44586-9. DOI: 10.1007/3-540-44586-2_8. URL: https://link.springer.com/chapter/10.1007/3-540-44586-2_8 (visited on 05/15/2018) (cit. on pp. 11, 12).

[49] Trevor Perrin. *The Noise Protocol Framework*. Revision: 33. Oct. 4, 2017. URL: https://noiseprotocol.org/noise.html (visited on 05/08/2018) (cit. on pp. 1, 2, 42).

[50] Vasile C. Perta et al. “A Glance through the VPN Looking Glass: IPv6 Leakage and DNS Hijacking in Commercial VPN Clients”. In: *Proceedings on Privacy Enhancing Technologies* 2015.1 (Apr. 1, 2015), pp. 77–91. DOI: 10.1515/popets-2015-0006. URL: https://content.sciendo.com/view/journals/popets/2015/1/article-p77.xml (visited on 05/12/2018) (cit. on p. 1).

[51] Phillip Rogaway. “Authenticated-Encryption with Associated-Data”. In: *Ninth ACM Conference on Computer and Communications Security (CCS-9)*. New York, NY: ACM Press, Nov. 2002, pp. 98–107 (cit. on p. 8).

[52] Victor Shoup. *Sequences of Games: A Tool for Taming Complexity in Security Proofs*. Published: Cryptology ePrint Archive, Report 2004/332. Nov. 2004. URL: https://eprint.iacr.org/2004/332 (cit. on pp. 3, 25, 26).

[53] WhatsApp. *Connecting One Billion Users Every Day*. July 26, 2017. URL: https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day (visited on 05/12/2018) (cit. on p. 2).

[54] WhatsApp. *WhatsApp Encryption Overview*. Technical white paper. Apr. 5, 2016. URL: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf (cit. on p. 2).

Acknowledgements

I am deeply thankful that I have been given the opportunity to prepare my master's thesis in the Prosecco research team at INRIA Paris. I have learned a lot already and I am looking forward to my time there as PhD student. I thank my thesis advisors Karthikeyan Bhargavan, Bruno Blanchet and Harry Halpin for accepting my application and funding my internship. I also thank Professor Müller-Quade for accepting this internship as external master's thesis, and Professor Hofheinz to agree to be my second reviewer. I thank Mario Strefler from the crypto research group at KIT, who helped me establish the first contact to the Prosecco team, and who in the following agreed to by my internal thesis advisor. I want to also thank Jeff, who I met during my studies in Rennes. He was the second person helping me and encouraging me to get in touch with the Prosecco team. I thank Harry Halpin for the spontaneous meeting on that Sunday in September, on my journey through Paris. I thank Bruno Blanchet, with whom I mainly worked during my internship: Thanks for teaching me CryptoVerif, helping me developing the understanding of protocols I now have, and asking me if I want to stay for a PhD. I thank Karthikeyan Bhargavan who helped made me not forget the bigger picture my research is embedded into. I thank Harry Halpin for pointing me to the right literature to help me understand what I am doing, and for pushing me through the final days of my master's thesis. During my internship, I was given the opportunity to attend the Real World Crypto conference, and a European project meeting in Lausanne. Also, I could meet Trevor Perrin, the author of the Noise Protocol Framework, and Jason A. Donenfeld, the author of WireGuard. This has contributed a lot to my time in Prosecco being a very satisfying research experience. I am grateful to everyone who participated in it. I thank everybody in the Prosecco team for supporting me during my internship, especially my office mates who motivated me to arrive earlier in the morning, our team assistant who manages all the necessary bureaucracy, those who supported me during the first steps I took in F*, K. who introduced the Tuesday cake tradition, and M. and N. for their (actually not) stupid jokes. I thank Nadim for his very useful constructive feedback on my work, and exciting discussions about Noise.

I see this master internship as the summit of the double degree in cryptography that I pursued for my master's, studying one year in Rennes, and one year in Karlsruhe. I am thankful to everyone who made this programme possible: Willi Geiselmann, Melina Metzig-Lotter, Ioana Gheta, Susanne Kaliwe, Felix Ulmer, Sylvain Duquesne, and finally J. and A. who participated in this programme at the same time from the German side.

I thank my parents for their unconditional love and support during my whole life. I thank my grandparents, who supported me without hesitation in all of my trips to foreign countries. I thank my girlfriend, who encouraged me to follow my interests (to Paris), for her love and support during this adventure.

A. Appendix

A.1. Proofs for the Different Compromise Scenarios

We provide the proofs done in CryptoVerif for the compromise scenarios defined in Section 4.5. The proof for scenario 1 was given and described in detail in Section 4.4 so we do not repeat it here.

Proof for Scenario 2. From `wireguard.AB.ephemeral_A.no_replay_prot.cv`.

```
proof {
  (* In the initiator, distinguish the cases talking to an honest or
     dishonest responder regarding its longterm public key. *)
  insert 37 "if S_X_pub = S_r_pub then";
  SArename E_i_pub_2;
  (* In the responder, distinguish the cases talking to an honest or
     dishonest initiator regarding its longterm public key. *)
  insert 2792 "if S_i_pub_recv_2 = S_i_pub_1 then";
  (* In the responder, find the initiator session that generated an
     honest ephemeral key, talking to an honest responder longterm key. *)
  insert 2727 "find i <= N_init_parties suchthat defined(E_i_pub_4[i]) && ↩
      E_i_pub_recv_2 = E_i_pub_4[i] then";
  (* In the initiator after receiving the second protocol message from
     the responder, find the responder session that generated an honest
     ephemeral key, talking to an honest initiator (here this means both
     initiator longterm and ephemeral are honest). *)
  insert 1656 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
      E_r_pub_recv_1 = E_r_pub[j] then";
  insert 179 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
      E_r_pub_recv_1 = E_r_pub[j] then";

  crypto rom(rom3_intermediate);
  crypto rom(rom2);
  crypto rom(rom1);
  crypto gdh(exp) S_r_priv_1 E_i_priv_5 S_i_priv_2 E_r_priv_2;
  crypto split_hashoutput *;
  success;
  simplify;
  crypto int_ctxt(enc) *;
  success;
  simplify;
  crypto ind_cpa(enc) *;
```

```
30    success;
31    simplify;
32
33    crypto int_ctxt_corrupt(enc) k_25;
34    success;
35    simplify;
36    crypto ind_cpa(enc) *;
37    success
38  }
```

Scenario 3. From `wireguard.AB.ephemeral_B.no_replay_prot.cv`.

```
1   proof {
2     (* In the initiator, distinguish the cases talking to an honest or
3        dishonest responder regarding its longterm public key. *)
4     insert 37 "if S_X_pub = S_r_pub then";
5     SArename E_i_pub_2;
6     (* In the responder, distinguish the cases talking to an honest or
7        dishonest initiator regarding its longterm public key. *)
8     insert 1248 "if S_i_pub_recv_2 = S_i_pub_1 then";
9     (* In the responder, find the initiator session that generated an
10       honest ephemeral key, talking to an honest responder longterm key. *)
11    insert 1183 "find i <= N_init_parties suchthat defined(E_i_pub_4[i]) && ↩
          E_i_pub_recv_2 = E_i_pub_4[i] then";
12    (* In the initiator after receiving the second protocol message from
13       the responder, find the responder session that generated an honest
14       ephemeral key, talking to an honest initiator (here this means both
15       initiator longterm and ephemeral are honest). *)
16
17    crypto rom(rom3_intermediate);
18    crypto rom(rom2);
19    crypto rom(rom1);
20    crypto gdh(exp) S_r_priv_1 E_i_priv_5 S_i_priv_2 E_r_priv_7;
21    crypto split_hashoutput *;
22    success;
23    simplify;
24    crypto int_ctxt(enc) *;
25    success;
26    simplify;
27    crypto ind_cpa(enc) *;
28    success;
29    simplify;
30
31    (* This covers, on the responder's side, the case of honest S_i_pub
32       but dishonest E_i_pub in the case of possible corruption of
33       S_i_pub: The attacker can't produce a valid ciphertext for
34       protocol message 3 (key confirmation) if the key wasn't
```

```
35      compromised, thus the decryption will fail on the responder's side
36      and the protocol won't continue. Especially the rcvd3 event will
37      not be triggered. *)
38    crypto int_ctxt_corrupt(enc) T_i_send_12;
39    success;
40    simplify;
41    crypto ind_cpa(enc) *;
42    success
43  }
```

Scenario 5. From `wireguard.AB.only_psk.cv`.

```
1   proof {
2     (* In the initiator, distinguish the cases talking to an honest or
3        dishonest responder regarding its longterm public key. *)
4     insert 32 "if S_X_pub = S_r_pub then";
5     SArename E_i_pub_2;
6     (* In the responder, distinguish the cases talking to an honest or
7        dishonest initiator regarding its longterm public key. *)
8     insert 1245 "if S_i_pub_recv_2 = S_i_pub_1 then";
9     (* In the responder, find the initiator session that generated an
10       honest ephemeral key, talking to an honest responder longterm key. *)
11    insert 1180 "find i <= N_init_parties suchthat defined(E_i_pub_4[i]) && ↩
          E_i_pub_recv_2 = E_i_pub_4[i] then";
12    (* In the initiator after receiving the second protocol message from
13       the responder, find the responder session that generated an honest
14       ephemeral key, talking to an honest initiator (here this means both
15       initiator longterm and ephemeral are honest). *)
16    insert 753 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
          E_r_pub_recv_1 = E_r_pub[j] then";
17    insert 174 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
          E_r_pub_recv_1 = E_r_pub[j] then";
18
19    crypto rom(rom3_intermediate);
20    crypto rom(rom2);
21    crypto rom(rom1);
22    crypto gdh(exp) S_r_priv_1 E_i_priv_5 S_i_priv_2 E_r_priv_9;
23    crypto split_hashoutput *;
24    success;
25    simplify;
26    crypto int_ctxt(enc) *;
27    success;
28    simplify;
29    crypto ind_cpa(enc) *;
30    success
```

Scenario 6. From `wireguard.AB.no_longterm_compromises.replay_prot.cv`.

```
proof {
  (* In the initiator, distinguish the cases talking to an honest or
     dishonest responder regarding its longterm public key. *)
  (* We want to have this case distinction after the check for corruption. *)
  insert 36 "if S_X_pub = S_r_pub then";
  SArename E_i_pub_2;
  (* In the responder, distinguish the cases talking to an honest or
     dishonest initiator regarding its longterm public key. *)
  insert 931 "if S_i_pub_recv_2 = S_i_pub_1 then";
  (* In the responder, find the initiator session that generated an
     honest ephemeral key, talking to an honest responder longterm key. *)
  insert 866 "find i <= N_init_parties suchthat defined(E_i_pub_4[i]) && ↩
      E_i_pub_recv_2 = E_i_pub_4[i] then";
  (* In the initiator after receiving the second protocol message from
     the responder, find the responder session that generated an honest
     ephemeral key, talking to an honest initiator (here this means both
     initiator longterm and ephemeral are honest). *)
  insert 610 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
      E_r_pub_recv_1 = E_r_pub[j] then";
  insert 190 "find j <= N_resp_parties suchthat defined(E_r_pub[j]) && ↩
      E_r_pub_recv_1 = E_r_pub[j] then";

  crypto rom(rom3_intermediate);
  crypto rom(rom2);
  crypto rom(rom1);
  crypto gdh(exp) S_r_priv_1 E_i_priv_5 S_i_priv_2 E_r_priv_9;
  crypto split_hashoutput *;
  success;
  simplify;
  crypto int_ctxt(enc) *;
  success;
  simplify;
  crypto ind_cpa(enc) *;
  success
```

A.2. Advantages to Break Secrecy in Different Compromise Scenarios

We indicate the advantages computed by CryptoVerif without further discussion. The scenario numbers correspond to the order they were listed in Section 4.5.

Scenario 2.

$$\begin{aligned}&(24n_r + 10n_i)\mathrm{Adv}^{\mathrm{int-ctxt}}_{\mathrm{AEAD},\mathcal{A}}(n) + 24n_r\mathrm{Adv}^{\mathrm{ind-cpa}}_{\mathrm{AEAD},\mathcal{A}}(n) + 24\mathrm{Adv}^{\mathrm{coll-res}}_{\mathrm{HASH},\mathcal{A}}(n)\\ &+(4n_i + 6n_r + 12n_{rom2}n_i + 2n_{rom3}n_r + 45n_in_i + 12n_rn_r + 3728n_rn_i + 138n_{rom1}n_i)/|\mathtt{Z\_t}|\\ &+(4n_{rom1} + 8n_{rom2} + 30n_{rom3} + 48n_rn_{rom3} + 2n_rn_{rom1} + 18n_rn_{rom2})n_i\mathrm{Adv}^{\mathrm{gdh}}_{\mathcal{G},\mathcal{A}}(n)\end{aligned}$$

Scenario 3.

$$\begin{aligned}&(12n_r + 8n_i)\mathrm{Adv}^{\mathrm{int-ctxt}}_{\mathrm{AEAD},\mathcal{A}}(n) + 92\mathrm{Adv}^{\mathrm{coll-res}}_{\mathrm{HASH},\mathcal{A}}(n) + (6n_r + 4n_i)\mathrm{Adv}^{\mathrm{ind-cpa}}_{\mathrm{AEAD},\mathcal{A}}(n)\\ &+(24n_r + 6n_i + 6n_{rom1}n_i + 12n_{rom2}n_i + 2n_{rom3}n_r + 27n_in_i + 494n_rn_r + 4292n_rn_i + 736n_rn_rn_i\\ &\quad + 3520n_{rom3}n_in_i)/|\mathtt{Z\_t}|\\ &+(2n_{rom1} + 6n_rn_{rom3} + 2n_rn_{rom1} + 18n_rn_{rom2} + 4 + 20n_{rom3} + 6n_{rom2}) \cdot n_r\mathrm{Adv}^{\mathrm{gdh}}_{\mathcal{G},\mathcal{A}}(n)\end{aligned}$$

Scenario 4.

$$\begin{aligned}&(48n_r + 12n_i)\mathrm{Adv}^{\mathrm{int-ctxt}}_{\mathrm{AEAD},\mathcal{A}}(n) + 24\mathrm{Adv}^{\mathrm{coll-res}}_{\mathrm{HASH},\mathcal{A}}(n) + 48n_r\mathrm{Adv}^{\mathrm{ind-cpa}}_{\mathrm{AEAD},\mathcal{A}}(n)\\ &+(2n_i + 6n_{rom1}n_i + 12n_rn_r + 26n_in_i + 1428n_rn_i + 392n_{rom2}n_i)/|\mathtt{Z\_t}|\\ &+(2n_{rom1} + 18n_rn_{rom2} + 2n_rn_{rom1} + 6n_{rom2})n_rn_i\mathrm{Adv}^{\mathrm{gdh}}_{\mathcal{G},\mathcal{A}}(n)\\ &+2n_{rom3}/|\mathtt{psk\_t}|\end{aligned}$$

Scenario 6.

$$\begin{aligned}&(20n_r + 8n_i)\mathrm{Adv}^{\mathrm{int-ctxt}}_{\mathrm{AEAD},\mathcal{A}}(n) + 64\mathrm{Adv}^{\mathrm{coll-res}}_{\mathrm{HASH},\mathcal{A}}(n) + (6n_r + 4n_i)\mathrm{Adv}^{\mathrm{ind-cpa}}_{\mathrm{AEAD},\mathcal{A}}(n)\\ &+(264n_{rom3}n_in_i + 18n_rn_rn_i + 18n_r + 2n_i + 6n_{rom1}n_i + 12n_{rom2}n_i + 4n_{rom3}n_r + 302n_rn_r\\ &\quad + 27n_in_i + 2464n_rn_i)/|\mathtt{Z\_t}|\\ &+(2n_{rom1} + 6n_rn_{rom3} + 2n_rn_{rom1} + 18n_rn_{rom2} + 4 + 20n_{rom3} + 6n_{rom2})\mathrm{Adv}^{\mathrm{gdh}}_{\mathcal{G},\mathcal{A}}(n)\end{aligned}$$

A.3. Running the Proofs in CryptoVerif

The CryptoVerif proof assistant can be downloaded from `http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/`. The models presented in this thesis have been tested to work with the most recent version 2.00. Our work can be reproduced by using the file `runcv` from the archive of our code. Paths defined within the file probably need to be adapted to the local system.

The following is the output of the `runcv` script that we got on our test machine (personal laptop), including running times and memory usage.

```
wireguard.AB.only_psk
../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -oproof wireguard.AB.only_psk.cvres wireguard.AB.only_psk.cv > wireguard.AB.only_psk.out
All queries proved.
189.359s (user 188.458s + system 0.900s), max rss 9647808K

wireguard.AB.longterm_compromises.no_replay_prot
../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -oproof wireguard.AB.longterm_compromises.no_replay_prot.cvres wireguard.AB.longterm_compromises.no_replay_prot.cv > wireguard.AB.longterm_compromises.no_replay_prot.out
All queries proved.
306.548s (user 306.007s + system 0.541s), max rss 4180016K

wireguard.AB.longterm_compromises.replay_prot
../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -oproof wireguard.AB.longterm_compromises.replay_prot.cvres wireguard.AB.longterm_compromises.replay_prot.cv > wireguard.AB.longterm_compromises.replay_prot.out
All queries proved.
934.592s (user 933.815s + system 0.777s), max rss 4181008K

wireguard.AB.no_longterm_compromises.no_replay_prot
../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -oproof wireguard.AB.no_longterm_compromises.no_replay_prot.cvres wireguard.AB.no_longterm_compromises.no_replay_prot.cv
> wireguard.AB.no_longterm_compromises.no_replay_prot.out
All queries proved.
156.486s (user 156.146s + system 0.340s), max rss 1595600K

wireguard.AB.no_longterm_compromises.replay_prot
../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -oproof wireguard.AB.no_longterm_compromises.replay_prot.cvres wireguard.AB.no_longterm_compromises.replay_prot.cv > wireguard.AB.no_longterm_compromises.replay_prot.out
All queries proved.
336.226s (user 335.901s + system 0.325s), max rss 1596240K

wireguard.AB.ephemeral_A.no_replay_prot
../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -oproof wireguard.AB.ephemeral_A.no_replay_prot.cvres wireguard.AB.ephemeral_A.no_replay_prot.cv > wireguard.AB.ephemeral_A.no_replay_prot.out
All queries proved.
285.147s (user 284.615s + system 0.531s), max rss 4181216K

```

```
32 wireguard.AB.ephemeral_B.no_replay_prot
33 ../cryptoverif-dev/xtime ../cryptoverif-dev/cryptoverif -lib wireguard -↩
       oproof wireguard.AB.ephemeral_B.no_replay_prot.cvres wireguard.AB.↩
       ephemeral_B.no_replay_prot.cv > wireguard.AB.ephemeral_B.no_replay_prot.↩
       out
34 All queries proved.
```