





A Tale of Two Worlds, a Formal Story of WireGuard Hybridization

Pascal Lafourcade[1], Dhekra Mahmoud[1], Sylvain Ruhault[2] and Abdul Rahman Taleb[2]

[1]Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Etienne, LIMOS, 63000 Clermont-Ferrand, France
[2]Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), France

Abstract

PQ-WireGuard is a post-quantum variant of **WireGuard** Virtual Private Network (VPN), where Diffie-Hellman-based key exchange is replaced by post-quantum Key Encapsulation Mechanisms-based key exchange. In this paper, we first conduct a thorough formal analysis of PQ-WireGuard's original design, in which we point out and fix a number of weaknesses. This leads us to an improved construction **PQ-WireGuard***. Secondly, we propose and formally analyze a new protocol, based on both WireGuard and PQ-WireGuard*, named **Hybrid-WireGuard**, compliant with current best practices for post-quantum transition about hybridization techniques. For our analysis, we use the SAPIC$^+$ framework that enables the generation of three state-of-the-art protocol models for the verification tools PROVERIF, DEEPSEC and TAMARIN from a single specification, leveraging the strengths of each tool. We formally prove that Hybrid-WireGuard is secure. Eventually, we propose a generic, efficient and usable Rust implementation of our new protocol.

1 Introduction

WireGuard is a Virtual Private Network (VPN) based on the Noise framework [69], introduced in 2017 [36] and integrated in the Linux kernel (version 5.6) [1]. In its protocol, peers initially exchange messages that constitute a *handshake*. A symmetric key is obtained from these messages, which they use to encrypt all subsequent exchanged messages. The protocol uses a key exchange component, which combines long-term and ephemeral *Diffie-Hellman* values. Different formal analyses of WireGuard have been proposed, in both the symbolic and the computational model, with or without computer-aided proof assistants [39, 62, 38, 59]. Even though the analyses point out some weaknesses, they confirm the robust design of the protocol, as most of the claimed properties are met.

With the recent developments on the construction of quantum computers, many protocols, including VPNs, are transitioning to the usage of post-quantum cryptography (*e.g.*, [11, 33, 74, 44]) because Diffie-Hellman key exchange is broken by Shor's quantum factoring algorithm [71], and needs to be replaced. The transition is particularly urgent because of *store-now-decrypt-later* attacks, where adversaries store current data encrypted with classic primitives, and decrypt them later when efficient quantum computers emerge. In 2016, NIST launched a post-quantum standardization competition for new primitives intended to replace currently quantum-broken primitives. For Diffie-Hellman key exchange, post-quantum key encapsulation mechanisms (KEMs) are proposed. At the end of the third round of the competition in 2022 [8], a single post-quantum KEM was selected for standardization, namely ML-KEM [65].

The security of WireGuard against quantum computing is also studied. In its original construction [36], a pre-shared symmetric key is used during the handshake as a means of protection against quantum computers, instead of using post-quantum primitives which are explicitly declared impractical in the protocol [37]. In 2021, a post-quantum version of WireGuard, named PQ-WireGuard, was proposed [51]. It basically replaces the Diffie-Hellman shared secrets with secrets generated using post-quantum KEMs. A computational proof as well as a symbolic proof are provided in [51] to ensure the security of the protocol.

Since post-quantum constructions are still new and their cryptanalysis is not fully mature, some of them could be broken in the near future, hence **genericity** (*i.e.*, being able to replace one scheme by another in an effective way) is advised. Furthermore, during a transitional phase, **hybridization** is recommended: the goal is to rely on both classic and post-quantum schemes simultaneously, ensuring that the final construction remains secure for as long as either the classic or the post-quantum scheme remains secure.

1.1 Our Contributions

We design, formally analyze and implement a new protocol which efficiently hybridizes WireGuard. More specifically, our contributions are:

Formal security analysis of PQ-WireGuard. We propose a new and more comprehensive model of PQ-WireGuard than the one presented in [51] and a more powerful attacker. We point out a mistake in the latter symbolic proof which has an impact on Unknown-Key-Share (UKS) attacks' analysis and we identify attack scenarios that are missed. Furthermore, PQ-WireGuard inherits the lack of anonymity from WireGuard. In addition, we point out that PQ-WireGuard uses a non-standard definition of the KEM encapsulation procedure, which impacts the protocol's implementation and analysis. We therefore propose a new version of the protocol, based on the common KEM's definition, which we refer to as PQ-WireGuard*. We formally prove that this version ensures anonymity, resists Unknown-Key-Share attacks and preserves the remaining properties (agreement and secrecy).

Hybrid-WireGuard. We propose a new protocol, called *Hybrid-WireGuard*, constructed by skillfully combining our improved construction PQ-WireGuard*, and WireGuard. We model this protocol and formally prove that all the security properties are achieved. Moreover, our analysis shows that if an attack scenario against a security property of Hybrid-WireGuard exists, then it is a combination of an attack on PQ-WireGuard* and an attack on WireGuard.

Adaptive analysis. We base our formal analysis on [59]. We point out, however, that the evaluation strategy used in [59], is not practical to analyze Hybrid-WireGuard because of a far more larger set of required evaluations. We therefore propose an adaptive evaluation strategy which greatly reduces the number of necessary evaluations needed to assess security properties, while preserving soundness.

Instantiation and implementation. We instantiate Hybrid-WireGuard with Classic McEliece [18] and ML-KEM [65] algorithms and propose a generic and efficient implementation in Rust. We provide benchmarks for Hybrid-WireGuard and PQ-WireGuard* compared to WireGuard and PQ-WireGuard, showing the practical usability of our constructions.

1.2 Paper organization

In Section 2, we present related work, for WireGuard and PQ-WireGuard-based constructions and tools, protocol analysis with SAPIC$^+$, and hybridization. Then, in Section 3, we detail WireGuard and PQ-WireGuard handshakes. Next, in Section 4, we describe our methodology for the formal verification of WireGuard, PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard, and present the analyzed security properties. Based on this description, we exhibit the results of our symbolic analysis for WireGuard and PQ-WireGuard in Section 5. Our results on PQ-WireGuard lead us to an improved version of the protocol, which we call PQ-WireGuard*, described and symbolically assessed in the same section. After, we introduce our protocol Hybrid-WireGuard in Section 6. We also exhibit the results of our symbolic analysis of the protocol, showing that its security is ensured as long as at least the security of the classic primitives or the security of the post-quantum primitives is ensured. Finally, we present our implementation and instantiation of cryptographic primitives in Section 7, and we conclude in Section 8.

2 Related Work

Constructions and tools based on WireGuard and PQ-WireGuard. In [10], a tweak is proposed to provide WireGuard with post-quantum security. It consists in transmitting the hash of the identity public key instead of the public key itself, hence protecting it from a quantum computer. The goal is to protect against *store-now-decrypt-later* attacks. However, this tweak relies on the strong assumption that peers' public keys are unknown to the attacker (a similar assumption is made in WireGuard [36]). NordLynx [67], a variant of WireGuard, proposes a post-quantum solution: after the successful handshake, a KEM key exchange is performed using ML-KEM, within the already mounted WireGuard tunnel. This solution provides post-quantum security for the final session keys, mitigating attacks from adversaries who can only observe the exchanged messages (*i.e.*, *passive* adversaries). Meanwhile, the handshake remains vulnerable against adversaries who can break the DH keys, and intercept and modify messages (*i.e.*, *active* adversaries).
Rosenpass [76] is an implementation of PQ-WireGuard [51], which aims to provide peers with a post-quantum shared secret. It is supposed to be used side-by-side with WireGuard to provide hybrid security. In other words, the VPN tunnel is mounted with WireGuard, and the latter is provided with a pre-shared key generated using Rosenpass. Rosenpass is then executed every two minutes to refresh the pre-shared key of the peers. This hybridization technique is different from our proposal (*i.e.*, combining cryptographic building blocks). In fact, many protocols use pre-shared keys to achieve post-quantum security (*e.g.*, [43, 50]); this pre-shared key can be generated using any uncompromised secure channel. In addition, RosenPass itself uses pre-shared keys, which leads to another layer of complexity handled by the infrastructure. Indeed, if the latter can handle secure symmetric (*i.e.*, post-quantum) pre-shared keys, these keys can be provided directly to WireGuard, without

using Rosenpass. Eventually, Rosenpass leads to a 4-message handshake: in this work, we propose a 2-message one to ensure hybrid security.

Protocol analysis with Sapic$^+$. Sapic$^+$ [29] unifies the use of ProVerif, Tamarin and DeepSec. It was recently used to analyze EDHOC protocol [53] and WireGuard [59]. Protocols are modeled in the applied Π-calculus [7] in a single Sapic$^+$ file which enables the generation of three state-of-the-art protocol models for Tamarin, ProVerif and DeepSec. In both ProVerif and Tamarin, security analysis regarding trace and equivalence properties for an unbounded number of sessions is supported, whereas DeepSec supports only equivalence-based properties while bounding the number of sessions. The equivalence property analyzed by DeepSec [31], namely *trace equivalence*, is weaker than the *observational equivalence* property supported by Tamarin [15] and ProVerif [28]. In [13, 48, 12], the authors state that observational equivalence may be too strong for privacy properties and show how trace equivalence may be the most suitable property for the matter. However, in [49], the authors exhibit a linkability attack on the e-Passport protocol using the *labeled bisimilarity* property (labeled bisimilarity and observational equivalence are equivalent [7]), although the protocol is proven to be unlinkable in [48] using trace equivalence. Tamarin supports natively associative and commutative symbols such as Diffie-Hellman exponentiation while ProVerif does not. Furthermore, ProVerif is sound but not complete [23] which means that it may find false attacks, whereas Tamarin is both sound and complete in the trace mode [14]. Sapic$^+$ allows to exploit the strengths of each tool, which justifies our choice for this framework.

Hybridization. As discussed earlier, emerging post-quantum KEMs and signatures are not mature enough to be confidently used on their own in security products. Hence, hybridization is currently advised during a transitional period [6, 66, 42], where post-quantum schemes are simultaneously used with classic ones. Hybridization is not trivial as combining classic and post-quantum primitives must be done without introducing further security flaws. Also, the final construction's security must be ensured as long as at least the security of the classic or the security of the post-quantum primitive is ensured. The European Telecommunications Standards Institute (ETSI), for example, discusses techniques for hybrid key exchange and provides computational proofs [41]. Many protocols and implementations already include hybrid solutions. OpenSSH, for instance, recently implemented in version 9.9 a hybrid key exchange using X25519 and ML-KEM [54]. In addition, numerous works provide performance analysis on hybrid variants of the TLS handshake (*e.g.*, [33, 68, 72, 74]), see [9] for a comprehensive analysis of the different solutions. In [70], the authors introduce KEMTLS, a variant of TLS using KEMs instead of signatures for authentication, formally verified in [26] with Tamarin. In the secure messaging domain, the Signal messaging protocol introduces a hybrid key agreement protocol [57]. Apple's IMessage also introduces PQ3 [11], a secure hybrid messaging protocol. A computational proof [73] and a symbolic proof of the latter protocol using Tamarin [61] are also available. For VPNs, a hybrid key exchange [75] is proposed for the IKEv2 protocol [40], later implemented by StrongSwan 6.0 [4], using the liboqs post-quantum library [74]. Our construction of Hybrid-WireGuard follows this line of research.

3 WireGuard and PQ-WireGuard

We describe WireGuard [36] and PQ-WireGuard [51] handshakes. We start by recalling the cryptographic building blocks used by both constructions.

Notations. In all descriptions, $Y \leftarrow \mathsf{f}(X)$ denotes the use of function f, with input bitstring X and output bitstring Y, $X \,||\, Y$ the concatenation of X and Y, $X \xleftarrow{\$} \{0,1\}^n$ a uniformly random generation of a n-bit string and $\perp$ a false proposition.

3.1 Cryptographic Building Blocks

DH key exchange. WireGuard uses Diffie-Hellman (DH) key exchange, based on a cyclic group $\mathbb{G}$ of generator g. We use $\mathsf{DH.gen}()$ to generate the DH key pair (s, S) where s is generated uniformly at random and $\mathsf{S} = g^{\mathsf{s}}$. We also use a DH computation, that takes as input the private key of one DH key pair, and the public key of another, *i.e.*, either (s_1, S_2) or (S_1, s_2), to generate the DH shared secret.

Key encapsulation mechanisms (KEM). A KEM is equipped with three operations: $\mathsf{KEM.gen}()$ generates the key pair (s, S). $\mathsf{KEM.encaps}(\mathsf{S})$ is the probabilistic encapsulation procedure, which takes as input a KEM public key S, and outputs a secret key k and a ciphertext ct. We also use the definition of KEM.encaps introduced in [51] to make the procedure deterministic, by adding an interface to provide random coins R as additional input to the procedure, *i.e.*, $\mathsf{KEM.encaps}(\mathsf{S}, R)$. Finally, $\mathsf{KEM.decaps}(\mathsf{s}, ct)$ is the deterministic decapsulation procedure, which takes as input a KEM private key s, and a ciphertext ct, and outputs a secret key k.
Remark: DH key exchange establishes a shared secret *without any interaction*, if both parties know each other's keys, typically with static long-term DH keys. Importantly, such non-interactive key exchange is *not possible*

using KEMs. Indeed, one peer A holds the keys and the other peer B must encapsulate using A's public key and send the resulting ciphertext: the shared secret is derived from some ephemeral randomness generated by B.

Authenticated encryption with associated data. An AEAD is a symmetric authenticated encryption algorithm. The encryption procedure $\mathsf{AEAD.Enc}(k, N, H, M)$ takes as input a key k, a nonce N, a header H, and a message M, and outputs a ciphertext C. The decryption procedure $\mathsf{AEAD.Dec}(k, N, H, C)$ takes as input a key k, a nonce N, a header H and a ciphertext C, and outputs a message M or $\perp$. We denote by SE an **unauthenticated symmetric encryption scheme**. $\mathsf{SE.Enc}(k, N, M)$ takes as input a key k, a nonce N, and a plaintext M, and outputs a ciphertext C, while $\mathsf{SE.Dec}(k, N, C)$ takes as input a key k, a nonce N, and a ciphertext C, and outputs a plaintext M.

Additional functions. WireGuard uses a **hash function** HASH, a **message authentication code** MAC and a **key derivation function** KDF (with key material k and some input I). In the context of WireGuard, KDF_n is indexed by an integer n, and the former outputs n keys of the same size. We denote the ℓ-th output by $\mathsf{KDF}_n(k, I)[\ell]$.

Table 1: Notations for DH keys and shared secrets, KEM keys, encapsulation outputs and encapsulation random inputs. Key pairs are denoted as (private key, public key). $*^c$ denotes a classic key, $*^{pq}$ a post-quantum key. The random inputs are specific to PQ-WireGuard [51].

DH **keys**		Static	Ephemeral
Initiator		$(\mathsf{s}_\mathsf{i}^\mathsf{c}, \mathsf{S}_\mathsf{i}^\mathsf{c})$	$(\mathsf{e}_\mathsf{i}^\mathsf{c}, \mathsf{E}_\mathsf{i}^\mathsf{c})$
Responder		$(\mathsf{s}_\mathsf{r}^\mathsf{c}, \mathsf{S}_\mathsf{r}^\mathsf{c})$	$(\mathsf{e}_\mathsf{r}^\mathsf{c}, \mathsf{E}_\mathsf{r}^\mathsf{c})$
DH **shared secrets**		Responder	
		Static	Ephemeral
Initiator	Static	$\mathsf{dh}_{\mathsf{s_i s_r}}$	$\mathsf{dh}_{\mathsf{s_i e_r}}$
	Ephemeral	$\mathsf{dh}_{\mathsf{e_i s_r}}$	$\mathsf{dh}_{\mathsf{e_i e_r}}$
KEM **keys**		Static	Ephemeral
Initiator		$(\mathsf{s}_\mathsf{i}^\mathsf{pq}, \mathsf{S}_\mathsf{i}^\mathsf{pq})$	$(\mathsf{e}_\mathsf{i}^\mathsf{pq}, \mathsf{E}_\mathsf{i}^\mathsf{pq})$
Responder		$(\mathsf{s}_\mathsf{r}^\mathsf{pq}, \mathsf{S}_\mathsf{r}^\mathsf{pq})$	-
KEM.encaps **outputs**		Secret key	Ciphertext
$\mathsf{S}_\mathsf{r}^\mathsf{pq}$		shk_1	ct_1
$\mathsf{E}_\mathsf{i}^\mathsf{pq}$		shk_2	ct_2
$\mathsf{S}_\mathsf{i}^\mathsf{pq}$		shk_3	ct_3
KEM.encaps **randoms**		Static	Ephemeral
Initiator		σ_i	r_i
Responder		σ_r	$\mathsf{r}_\mathsf{r}, \mathsf{r}_\mathsf{e}$

3.2 Handshakes

To ease comparison between constructions, all algorithms and key derivations are presented side-by-side in Table 4 and Table 5, with notations from Table 1.

WireGuard handshake. We describe in Table 4 the construction of the WireGuard handshake messages, similarly to [36, 39, 59]. The handshake consists of two messages: InitHello sent by Initiator, followed by RespHello sent by Responder. Both Initiator and Responder hold long-term DH keys $(\mathsf{s}_\mathsf{i}^\mathsf{c}, \mathsf{S}_\mathsf{i}^\mathsf{c})$ and $(\mathsf{s}_\mathsf{r}^\mathsf{c}, \mathsf{S}_\mathsf{r}^\mathsf{c})$ respectively. WireGuard assumes that both peers have exchanged their public keys before any handshake, in a secure authenticated way. Table 5 describes the chain of key derivation done by both peers to agree on the final session keys: tk_i used to encrypt traffic sent by Initiator, and tk_r used to encrypt traffic sent by Responder.
Both Initiator and Responder generate ephemeral DH keys $(\mathsf{e}_\mathsf{i}^\mathsf{c}, \mathsf{E}_\mathsf{i}^\mathsf{c})$ and $(\mathsf{e}_\mathsf{r}^\mathsf{c}, \mathsf{E}_\mathsf{r}^\mathsf{c})$ respectively during the handshake. Then, four DH shared secrets are used in the key derivation, that correspond to the four combinations of static and ephemeral DH keys of both peers: $\mathsf{dh}_{\mathsf{s_i s_r}}$, $\mathsf{dh}_{\mathsf{s_i e_r}}$, $\mathsf{dh}_{\mathsf{e_i s_r}}$ and $\mathsf{dh}_{\mathsf{e_i e_r}}$.

PQ-WireGuard handshake. Since the DH key exchange is not post-quantum secure, it needs to be replaced by post-quantum KEMs. In [51], the authors replace the DH shared secrets by ones generated using KEMs such that the same security properties are ensured. In this case, both Initiator and Responder hold static keys $(\mathsf{s}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}})$ and $(\mathsf{s}_\mathsf{r}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}})$, that correspond to a post-quantum KEM. We describe the handshake and the key derivation, as proposed in [51], in Table 4 and Table 5 respectively. The differences from WireGuard are highlighted in the tables.
DH shared secrets that involve at least one ephemeral key are easily replaced by KEMs. The one corresponding to $\mathsf{dh}_{\mathsf{s_i e_r}}$ is generated by Responder using $\mathsf{S}_\mathsf{i}^{\mathsf{pq}}$ (shk_3 on line 4 for RespHello). The one corresponding to $\mathsf{dh}_{\mathsf{e_i s_r}}$ is generated by Initiator using $\mathsf{S}_\mathsf{r}^{\mathsf{pq}}$ (shk_1 on line 4 for InitHello). The one corresponding to $\mathsf{dh}_{\mathsf{e_i e_r}}$ is generated by Responder using Initiator's ephemeral key $\mathsf{E}_\mathsf{i}^{\mathsf{pq}}$ (shk_2 on line 3 for RespHello). Meanwhile, shared secret $\mathsf{dh}_{\mathsf{s_i s_r}}$, generated using the static keys of the peers, cannot be replaced by a secret generated using a KEM key exchange, because the latter requires at least one interaction. Instead, the authors of [51] replace this shared key by some tweaks to the protocol, making sure that the desired security properties remain satisfied. Observe that $\mathsf{dh}_{\mathsf{s_i s_r}}$ binds both identities of the peers to the key derivation. It also ensures that the first sent message InitHello is already authenticated, helping to mitigate Denial-of-Service (DoS) attacks. To replace this secret, the authors impose using a pre-shared key psk (optional in WireGuard). They discuss that one may rely on the assumption that public keys are actually not public and hence unknown to attackers, similarly to other works making this assumption [36, 10], making the use of the value $\mathsf{psk} \leftarrow \mathsf{HASH}(\mathsf{S}_\mathsf{i}^{\mathsf{pq}} \oplus \mathsf{S}_\mathsf{r}^{\mathsf{pq}})$ enough for security.
In addition, shared secret $\mathsf{dh}_{\mathsf{s_i s_r}}$ does not rely on any ephemeral randomness, adding a layer of protection against potential random state corruption. For this reason, the authors redefine the KEM.encaps procedure to make it deterministic. The random coins are instead provided as input, so that the shared secrets are derived from a trusted source of randomness. To generate the secret corresponding to $\mathsf{dh}_{\mathsf{e_i s_r}}$, Initiator combines with KDF_1 an ephemeral random $\mathsf{r_i}$ with a static one σ_i and provides the output as input to KEM.encaps. Responder does the same during RespHello's construction to generate the secret corresponding to $\mathsf{dh}_{\mathsf{s_i e_r}}$. Finally, the secret corresponding to $\mathsf{dh}_{\mathsf{e_i e_r}}$, involves only ephemeral coins.
We give more details about the security properties achieved by PQ-WireGuard in Section 4. Finally, for their computational proof in [51], the authors choose IND-CCA [17] security for the KEM corresponding to the static keys, and IND-CPA [17] security for the KEM corresponding to the ephemeral keys.

4 Formal Analysis and Claimed Properties

We describe our approach for the analysis of WireGuard, PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard with SAPIC$^+$ and the protocol verifiers PROVERIF, TAMARIN and DEEPSEC. We also describe claimed security properties for the protocols.

4.1 Formal Analysis

We do not go into the full details of the symbolic approach (see [22, 24, 32] for a detailed presentation of this approach).

Threat model. The Dolev-Yao attacker model [35], is typically the assumed adversary in symbolic models. The Dolev-Yao attacker has full control over the network through which messages are exchanged. Attacker capabilities include eavesdropping, removing, duplicating, replaying, substituting and delaying of all messages sent by protocol participants and also include insertion of messages of her choice in public channels. Consideration of attacker capabilities has a strong impact on the analysis. In [59], for instance, considering an attacker's access to the DH precomputation allows the authors to find attack scenarios, that were not visible in previous analyses of WireGuard. Likewise, following the steps of [59], we consider atomic capabilities of the attacker, that is, the attacker may have access to the protocol's atomic terms. As a consequence, when analyzing a protocol, one needs to consider the set of cryptographic keys (or more generally all secrets involved, as one could consider randomness generation) that are involved in a full execution of the protocol.

Cryptographic primitives. In the symbolic model, messages are represented as terms which can be either atomic (to represent fresh values such as keys or random coins) or constructed by applying function symbols. For instance, the terms $\mathsf{pk}(sk)$, $\mathsf{aenc}(m, \mathsf{pk}(sk))$ and $\mathsf{adec}(c, sk)$ can represent a public key function, an asymmetric encryption and an asymmetric decryption, respectively. To model the behavior of cryptographic primitives, function symbols may be ruled by a set of equations *i.e.*, an *equational theory*. For example, the equation $\mathsf{adec}(\mathsf{aenc}(m, \mathsf{pk}(sk)), sk) = m$ states that decrypting an encrypted message m using the proper key pairs sk and $\mathsf{pk}(sk)$, results in the same message m. Thus, cryptography is assumed to be *perfect* in the symbolic model, and one can decrypt only in possession of the secret key sk. It is noteworthy to mention that the sole equalities between the terms are those explicitly specified by the equational theory. If no equation is added for $\mathsf{pk}(sk)$, then it behaves as a perfect one-way function. The latter equation is the standard equation used to model

public key encryption in the symbolic model. In this work, we use this equational theory to model KEMs as in [20, 51].

Trace and equivalence. As presented in Table 3, there are two main classes of security properties in the symbolic approach, namely *trace properties* and *equivalence properties*. Trace properties are defined with regard to the executions of the protocol. A trace property is considered to be satisfied when, in all possible traces or executions of the protocol, the property holds. Several security properties can be expressed as trace properties, such as secrecy (as reachability property, that is, whether the attacker can *reach* or not some secret terms) and authentication (as correspondence assertions, that is, whenever a specific event is reached in the protocol, another event should be previously executed). Most of the existing symbolic tools support the specification of trace properties. Equivalence properties are defined between two scenarios that the attacker should not be able to tell apart. Privacy properties can be expressed as equivalence properties, such as anonymity, unlinkability and strong secrecy. In [38, 51], the authors analyze and formalize anonymity as a trace property. We note that proving that a property holds with equivalences provides a stronger guarantee than when it is proven as a trace property. There are two major equivalence properties supported by the existing tools: *trace equivalence* and *observational equivalence*. In addition to the trace properties, we use both notions of equivalence to analyze privacy.

Security properties analysis. While researchers in symbolic protocol verification have largely converged on common definitions and formalizations of security properties, no such agreement exists about how the results of the analysis should be presented nor described. The most commonly employed approach is to ascertain whether a protocol meets a security property under predetermined assumptions regarding the attacker's capabilities, as demonstrated by the overwhelming majority of analyses, *e.g.*, [55, 19, 76]. A more comprehensive analysis entails conducting a deeper verification to determine what an attacker can do to compromise the security property or what is the most potent threat model that the protocol can still withstand. In this work, we follow the steps of [59, 47] to express security properties as concise logical formulas in Conjunctive Normal Forms (CNF) that define the minimal key compromise conditions required to breach a security property. For instance, if the CNF for the secrecy of the session key is $\mathsf{psk} \wedge (\mathsf{s}_\mathsf{r}^\mathsf{c} \vee \mathsf{e}_\mathsf{i}^\mathsf{c})$, it means that the session key is secret unless psk and $\mathsf{s}_\mathsf{r}^\mathsf{c}$, or psk and $\mathsf{e}_\mathsf{i}^\mathsf{c}$ are compromised.

Evaluation strategy. Expressing security properties as compact formulas involving minimal compromise scenarios requires an evaluation strategy to ensure efficiency and completeness of the evaluation: in [59], the set of required evaluations for each secrecy and agreement property (2^{15} cases) is reduced to a subset of **4860** evaluations by eliminating trivial cases. Nevertheless, even with this reduction, the analysis requires a server with a 1.5 GHz CPU of 256 cores, and 512 Go of RAM. With this architecture, agreement and key secrecy are each evaluated in around 15 minutes, to which one needs to add around 90 minutes for each property, for the computation of the minimal formulas (see [58] for details). In this work, our target is protocols with a potentially larger set of compromise cases, as hybridization involves classic and post-quantum keys. For instance, PQ-WireGuard from [51] requires up to 2^{20} compromise cases, and a composition of WireGuard and PQ WireGuard would require 2^{35} compromise cases. The evaluation strategy proposed in [59] does not scale to such values. Hence, we propose an *adaptive* strategy that drastically reduces the number of necessary steps to obtain results more efficiently, while preserving soundness.

Table 2: Optimization of WireGuard analysis.

Step	[58, 59]	**Our work**
ProVerif queries generation	≈ 15m	≈ 5s
ProVerif queries evaluation	≈ 2h30m	≈ 2m45s
CNF computation	≈ 1h30m	≈ 2s
Tamarin lemma generation	Manual	Automatic

Table 2 shows the impact of our new evaluation strategy on symbolic verification performance, compared to previous works [58, 59]. In our new strategy, we *do not* consider, for all trace properties, all potential key combinations, for an adversary with read or write access to cryptographic keys, nor do we generate all possible queries beforehand, as in [59]. Instead, we first consider the n keys for read access. For each query considering a possible set of $t \leq n$ keys, if it evaluates to true, then we eliminate all supersets of this set from the evaluation, since we know that the corresponding queries will also evaluate to true. This trick eliminates $\binom{n-t}{1}+\ldots+\binom{n-t}{n-t} = 2^{n-t}-1$ cases from the evaluation. The corresponding algorithm is described in Algorithm 1. Next, when all necessary queries for read access are evaluated, we evaluate ones for write access. We also propose an adaptive approach: each disjunction of CNFs computed from read access leads to a single query to evaluate.

Algorithm 1 Evaluation strategy for read access keys.

Input: keys for read access $k_1, \ldots, k_n$

for $t = 1 \ldots n$ **do**
  $S_t \leftarrow \big\{\{k_{i_1}, \ldots, k_{i_t}\} \mid 1 \leq i_1, \ldots, i_t \leq n\big\}$
for $t = 1 \ldots n$ **do**
  for each $E \in S_t$ **do**
    if query conditioned on E is true **then**
      for $j = t + 1 \ldots n$ **do**
        $S_j \leftarrow \{S \mid S \in S_j \text{ and } E \not\subset S\}$

For instance, there are 3 evaluations for all agreement properties for WireGuard (*c.f.*, Table 6). This adaptive analysis requires less than 30 queries to evaluate, instead of 4860 as in [59]. Furthermore, computation of CNF is now straightforward.
We also optimize precision: in ProVerif, a query may be evaluated as cannot be proved, which can be resolved with instruction set preciseActions = true. potentially at the cost of a longer evaluation [25]. In [59], such instruction is set at property level and hence is applicable for all queries of a given property. Here, we are more fine-grained as we only increase precision for a dedicated small set of queries, which also optimizes evaluation.
Eventually, for Hybrid-WireGuard, analysis is also adaptive: CNFs computed from read access, from WireGuard and PQ-WireGuard*, are directly used to assess security properties for an adversary with read access to keys. Then, as before, we evaluate queries for an adversary with write access to keys.

4.2 Claimed Security Properties

The evaluated security properties are presented in Table 3: ✗ (resp. ✓) denotes that the property is not analyzed (resp. is analyzed). We give an informal definition for each mentioned security property, formal queries are detailed in Appendix B.

Resistance against Maximal EXposure (MEX) attacks. These attacks involve revealing static and/or ephemeral keys and/or used randomness to the adversary [56, 45, 46]. We consider these attacks for all properties: our methodology ensures that all combinations are analyzed, between the "minimal case" (no reveal) and the "maximal case" (all static keys, ephemeral keys and used randomness are revealed).

Resistance against Unknown-Key-Share attacks. UKS attacks enable an attacker to coerce honest peers into exchanging keys with parties other than the ones they believe they are communicating with, without being aware of this exchange [21, 27]. This property is analyzed for Initiator and Responder (*unilateral* case) and for both peers (bilateral case), as a *trace property*. We formalize this property as in [51].

Session uniqueness. This property is analyzed for both Initiator and Responder as a *trace property*. It states that a session key computed on each side is unique thanks to the use of ephemeral keys. We formalize this property as in [51].

Anonymity. This property is analyzed for both Initiator and Responder. It ensures that a user is able to participate in the protocol without an attacker being able to reveal their identity. It is defined as an *equivalence property* between two systems involving two distinct identities that the attacker is unable to distinguish. As stated in previous sections, we also analyze anonymity for both equivalence properties (*i.e.*, *trace equivalence* and *observational equivalence*). We formalize this property as in [59] for observational equivalence, and we adapt the formal definition from [30] for trace equivalence.

Message agreement. This property corresponds to full agreement, as defined in [63]: if a peer A receives a message, apparently from another peer B, then B has previously been running the protocol with A, and both peers agreed on sent, received and atomic data used in the protocol. We formalize this property as in [59].

Resistance against Key Compromise Impersonation (KCI). These attacks occur when an attacker gains access to the static secret material of a peer, and is able to impersonate their corresponding peer during a run of the protocol. We do not model this property explicitly as in [51], instead it is directly deduced from our message agreement properties.

Key (strong/mutual) secrecy. Secrecy is considered for the session keys computed on both sides after a complete handshake. In the classic case, these keys involve the four possible DH computations. In the post-quantum case, the session keys involve the three possible KEM computations. Finally, in the hybrid case, the keys involve all the DH and KEM computations. In [51], the authors consider the session keys secret when they are "indistinguishable from a random string" to the attacker. At the same time, in their symbolic analysis, they only consider a weaker definition of secrecy, expressed as a *reachability property* and proven as a *trace*

property, *i.e.*, the attacker cannot recover the entire secret. This property is weaker because to prove or analyze the indistinguishability of a term from a freshly generated random coin, one needs to define secrecy as an *equivalence property*. We examine both definitions: we employ the term **Key secrecy** to refer to the trace property, and **Key strong secrecy** to refer to the equivalence property, as presented in Table 3. Secrecy as a trace property is considered for each peer, *i.e.*, the keys computed on Initiator and Responder sides (since they may not agree on the same keys) and in addition (as done in [51]), we consider **Key mutual secrecy**, which combines key secrecy and agreement. We model key secrecy as in [59], key mutual secrecy as in [51] and *key indistinguishability*'s definition from [19] which we refer to as key strong secrecy.

Key (mutual) forward secrecy. This property is also considered for the session keys computed on both sides after a complete handshake. It is also considered for each peer on both sides, and mutually like the mutual secrecy described above. We analyze this property as a *trace property*, where all static keys are revealed to the attacker at a later phase, *i.e.*, after protocol sessions have been executed. The attacker should not be able to learn the keys of previous sessions. We also consider **Key mutual forward secrecy**, which combines forward key secrecy and agreement. We model key forward secrecy as in [59] and key mutual forward secrecy as in [51].

Note that in Table 3, some security properties are not analyzed for Rosenpass [76], as authors claim that they are directly inherited from PQ-WireGuard. In addition, they propose a tweak to PQ-WireGuard to ensure security against *state disruption attacks*. Such attacks enable an attacker who controls Initiator's local time to inhibit future handshakes [2, 3] (WireGuard and PQ-WireGuard use a timestamp in the first message). However, they do not provide a symbolic proof of the claimed security against these attacks. In addition, these attacks can be mitigated with simple engineering solutions [36]. In our work, we do not model Rosenpass, but only WireGuard and PQ-WireGuard (along with our new constructions).

Table 3: Security properties, analyzed as trace (t) or equivalence (e) properties. [38] and [59] model Wireguard, [51] PQ-WireGuard, [76] Rosenpass, and we model WireGuard, PQ-WireGuard, PQ-WireGuard* (Section 5.2) and Hybrid-WireGuard (Section 6). FS denotes Forward Secrecy.

Property	[38]	[59]	[51]	[76]	**Our work**
MEX resistance	✗	✓^{t,e}	✓^{t}	✗	✓^{t,e}
UKS resistance	✗	✗	✓^{t}	✗	✓^{t}
Session uniquen.	✓^{t}	✗	✓^{t}	✗	✓^{t}
Anonymity	✓^{t}	✓^{e}	✓^{t}	✓^{e}	✓^{e}
Message agreem.	✓^{t}	✓^{t}	✓^{t}	✓^{t}	✓^{t}
KCI resistance	✓^{t}	✗	✓^{t}	✗	✓^{t}
Key secrecy	✓^{t}	✓^{t}	✓^{t}	✓^{t}	✓^{t}
Key strong secr.	✗	✗	✗	✗	✓^{e}
Key mutual secr.	✗	✗	✓^{t}	✗	✓^{t}
Key FS	✓^{t}	✓^{t}	✓^{t}	✓^{t}	✓^{t}
Key mutual FS	✗	✗	✗	✗	✓^{t}
Tools					
ProVerif	✗	✓	✗	✓	✓
Tamarin	✓	✓	✓	✗	✓
DeepSec	✗	✗	✗	✗	✓

5 Analysis of WireGuard and PQ-WireGuard

We describe our formal analysis of WireGuard [36] and PQ-WireGuard [51] in the symbolic model using Sapic$^+$. This leads us to our new construction PQ-WireGuard*, that enhances PQ-WireGuard. For each security property, we derive the corresponding CNF using ProVerif. We recall that a CNF captures all minimal key

compromises scenarios required to break a property. For WireGuard's trace properties, we verify the CNFs with TAMARIN due to its more complete Diffie-Hellman model. For PQ-WireGuard, we omit TAMARIN as no exponentiation is involved. For equivalence properties, we confirm PROVERIF's attack scenarios with DEEPSEC instead of manual trace analysis, since DEEPSEC is complete while PROVERIF is not. To ease comparison, our results are presented side-by-side in Table 6. Our evaluations are available in our companion artifacts [60].

5.1 WireGuard

Our model of WireGuard extends the previous SAPIC$^+$ model of [59] in several ways. First, we consider that a participant of the protocol can play both roles during the execution. In other words, a peer can act as Initiator and/or Responder, as stated in the original specification of WireGuard. We analyze new properties: security against UKS attacks, resistance against KCI, session uniqueness, mutual secrecy, strong secrecy and mutual FS. For analyzing UKS attacks resistance, an attacker can actively participate in the protocol using their own identity, instead of only trying to compromise honest participants. Our results, depicted in Table 6, confirm the analysis from [59] for the same security properties. For newly verified properties, we exhibit new results on WireGuard, that confirm the robustness of the protocol. In particular, we prove that UKS attacks are infeasible thanks to the contributive nature of the DH products included in the key derivation.

5.2 PQ-WireGuard

Like WireGuard's model, we permit role switching between peers. This enables us to detect UKS attack scenarios that would be missed using the model from [59], as it does not account for role switching.

Unknown-Key-Share attacks. As in [51], we consider unilateral and bilateral UKS attacks. We identify a bug in the PQ-WireGuard symbolic model from [51] due to a discrepancy between the protocol description in the paper [51] and its TAMARIN model [52]. Specifically, in Algorithm 2 (line 12) of [51], `ct2` is the ciphertext resulting from the encapsulation of Initiator's ephemeral key $\texttt{epk}_i$. This ciphertext is included in the key derivation chain in Table II in [51] (which corresponds to step 6 of Table 5). We verify the computational proof in the paper and confirm that it aligns with the protocol specifications, yet, the symbolic model does not. In line 221 of the TAMARIN code given in [52], `sct2 = aenc{ka}pkI` represents the ciphertext resulting from the encapsulation of Initiator's static public key `pkI`, which is included in the derivation chain in subsequent lines. In contrast, `ect = aenc{k}pekI`, the ciphertext from encapsulating Initiator's ephemeral key `pekI` is never included in the key derivation chain. While this might appear to be a minor modeling error, its impact is significant: it transforms the lemma `UKS_on_responder_resistance` that was verified under the flawed model into a falsified lemma, as it enables UKS attack scenarios. In fact, the reason the attack disappears when the ciphertext (resulting from encapsulating Initiator's static key) is included in the session key derivation chain, is that the ciphertext (modeled as a standard public key encryption) remains bound to the static key. By incorporating it into the derivation chain, the session key itself also becomes bound to Initiator's static key. Thus, an honest Responder cannot be tricked into establishing a session key with an Initiator possessing a different key. The CNF for the UUKS on Initiator's side is $\mathsf{psk} \wedge (\sigma_\mathsf{i} \vee \mathsf{s}_\mathsf{i}^{\mathsf{pq}}) \wedge (\mathsf{s}_\mathsf{i}^{\mathsf{pq}} \vee \mathsf{r}_\mathsf{r}) \wedge (\mathsf{e}_\mathsf{i}^{\mathsf{pq}} \vee \mathsf{r}_\mathsf{e})$, and for Responder's side is $\mathsf{psk} \wedge (\sigma_\mathsf{i} \vee \mathsf{s}_\mathsf{r}^{\mathsf{pq}}) \wedge (\sigma_\mathsf{r} \vee \mathsf{r}_\mathsf{i}) \wedge (\mathsf{e}_\mathsf{i}^{\mathsf{pq}} \vee \mathsf{r}_\mathsf{e})$. Consequently, if either all static and pre-shared keys are compromised, or if the randomness source is compromised, the protocol maintains UUKS resistance. However, the protocol is vulnerable to UUKS attacks when an adversary compromises both the static keys and randomness of the same peer - a corruption pattern excluded in [51]. However, the CNF for the BUKS property is $\mathsf{e}_\mathsf{i}^{\mathsf{pq}} \vee \mathsf{r}_\mathsf{e}$, that is, BUKS resistance does not hold under the MEX scenario involving compromised randomness, contrary to the resistance requirement claimed by the protocol authors. The attack proceeds as follows: Alice, acting as an honest Initiator with a static public key $\mathsf{S}_\mathsf{i}^{\mathsf{pq}}$ and sharing a default $\mathsf{psk} = \mathsf{HASH}(\mathsf{S}_\mathsf{i}^{\mathsf{pq}} \oplus \mathsf{S}_\mathsf{r}^{\mathsf{pq}})$ with dishonest Eve with $\mathsf{S}_\mathsf{r}^{\mathsf{pq}}$ as static public key, generates an ephemeral key $(\mathsf{e}_\mathsf{i}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}})$, encapsulates the public key of Eve to obtain the shared secret shk_1, and sends an InitHello message. Eve receives Alice's message, decapsulates the received ciphertext with her secret key to obtain shk_1, re-encapsulates shk_1 using Alice's public key, and initiates a new session with Alice (consistent with WireGuard/PQ-WireGuard's dual-role design) by sending an InitHello message with the same psk, $\mathsf{E}_\mathsf{i}^{\mathsf{pq}}$, and the shared secret shk_1. When Alice receives Eve's new message, she processes it as a Responder by decapsulating using her static secret key to obtain the shared secret shk_1, and encapsulates on Eve's sent ephemeral public key $\mathsf{E}_\mathsf{i}^{\mathsf{pq}}$ (also her own ephemeral key) to obtain the shared secret shk_2, then transmits a RespHello response. Eve, possessing either the compromised randomness or the secret key $\mathsf{e}_\mathsf{i}^{\mathsf{pq}}$, completes the attack by decapsulating with her own key to obtain shk_3 and re-encapsulating it with Alice's public key. This results in Alice erroneously establishing a session key with herself while believing it is with Eve. To address this vulnerability in PQ-WireGuard, we modify the key derivation process to incorporate the concatenation of Initiator's and Responder's public static keys respectively, *i.e.*, $\mathsf{HASH}(\mathsf{S}_\mathsf{i}^{\mathsf{pq}} \,||\, \mathsf{S}_\mathsf{r}^{\mathsf{pq}})$. This fix establishes an explicit binding between the session keys and the protocol participants, and also encodes the Initiator-Responder distinction in the derived key material to prevent the role confusion exploited in the BUKS attack.

Initiator and Responder Anonymity. Likewise to WireGuard, our analysis shows that anonymity is not guaranteed neither for Initiator nor for Responder. In [59], two fixes are proposed to ensure anonymity: one based on the pre-shared key psk, and another based on DH shared secret $\mathsf{dh}_{\mathsf{s_i s_r}}$. To reach anonymity in PQ-WireGuard, we consider the fix based on the pre-shared key, as the one based on $\mathsf{dh}_{\mathsf{s_i s_r}}$ is not applicable in the post-quantum case. The fix consists in computing the MAC value m1 in InitHello and RespHello, using the pre-shared key as MAC key, instead of the static KEM key of the other party. This fix requires the adaptation of Responder's InitHello message consumption, as upon receiving InitHello, Responder cannot directly know which pre-shared key to use from its database for the verification of m1. Hence, we change the order of Responder's operations upon receipt of InitHello: Responder has to perform KEM.Decaps and AEAD.Dec operations before m1 verification because decryption of the static field is required to identify Initiator and retrieve the corresponding pre-shared key. Note that the original order is inherited from WireGuard design. As explained in [36], m1 check should protect Responder against DoS attacks: if m1 verification fails, Responder is prevented from computing costly DH and AEAD.Dec computations. We note however that this protection is based on a false assumption, that Initiator's and Responder's static keys (used as MAC keys in WireGuard) are *secret*. Without this assumption, DoS attacks against Responder are *realistic*: knowledge of the static public keys implies the ability to compute an arbitrary InitHello message with a correct m1 field, implying DH and AEAD.Dec computations. Meanwhile, on Initiator's side, no modification is required for RespHello message reception: Initiator has knowledge of which pre-shared key to use and check message authentication code m1 from RespHello message, using the received sid_i field. For completeness, InitHello message consumption for all protocols is detailed in Appendix A.

Probabilistic encapsulation and well-known KEMs. In PQ-WireGuard, a deterministic KEM encapsulation is used to ensure security against MEX attacks (*c.f.*, Section 3). In addition, the authors construct an IND-CPA KEM called Dagger based on SABER [16], a known KEM, to have messages small enough to avoid IP fragmentation. We stress that relying on such unconventional constructions can have security risks since it restricts the choices of libraries that can be used, forcing a developer to re-implement the KEM. In addition, controlling the source of randomness when using cryptographic primitives is unadvised, because of the associated risks when instantiating the random coins. In our improved construction, we propose to make standard choices for the KEMs.
First, we use the standard probabilistic encapsulation instead of the modified deterministic procedure. Nevertheless, our symbolic analysis shows that security against MEX attacks is still ensured, thanks to the use of the pre-shared key psk at an early stage during the key derivation (Step 4 in Table 5), under the assumption that psk is an actual secret pre-shared key. Note that psk is already used at this stage in PQ-WireGuard, but is not used to prove security against MEX attacks. This adaptation, composed with the previous adaptation to ensure resistance against UKS attacks, leads to modify PQ-WireGuard key derivation to include $(\mathsf{HASH}(\mathsf{S}_\mathsf{i}^{\mathsf{pq}} \,||\, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}) \,||\, \mathsf{psk})$. Second, we rely on standard KEM implementations, potentially at the cost of larger messages. This choice allows developers to use well-known cryptographic libraries, implementing standard KEMs with fixed parameters. Since existing KEMs usually aim to achieve IND-CCA security, we use IND-CCA KEMs for both static and ephemeral keys.

PQ-WireGuard*. We integrate all the previously proposed fixes into a modified version of PQ-WireGuard which we refer to as PQ-WireGuard*, described in Table 4 and Table 5. We formally analyze PQ-WireGuard* with the same tools and strategy used for PQ-WireGuard. Our analysis results in Table 6 show that UKS attacks are now infeasible. As for anonymity, our construction of PQ-WireGuard* also embeds modifications to enhance the property. In PQ-WireGuard, this property is never ensured, even when no key nor randomness is revealed to the attacker. However, with PQ-WireGuard* construction, anonymity is guaranteed; note that for this property, CNFs for Initiator and Responder are asymmetric, because of the asymmetry of keys owned by the peers (Responder does not have an ephemeral KEM key pair). We also observe that the other security properties remain unchanged. The only difference is that PQ-WireGuard* relies on common KEM definition, so it does not use terms σ_i and σ_r. Hence, obtained CNFs for agreement properties for PQ-WireGuard* do not contain these terms, as in the case of PQ-WireGuard.

6 Hybrid-WireGuard: Protocol and Analysis

Protocol definition. We introduce a new handshake, described in Table 4 and Table 5, that hybridizes WireGuard: Hybrid-WireGuard combines DH secrets from the WireGuard handshake, with post-quantum KEM secrets from PQ-WireGuard*. The goal is to ensure security of the protocol as long as at least the security of the classic primitives or the security of the post-quantum primitives is ensured. To reach this goal, we propose a construction inspired from existing works (*e.g.*, [44, 54, 75]). In this case, Initiator and Responder hold static long-term DH and KEM key pairs $\big((\mathsf{s}_\mathsf{i}^\mathsf{c}, \mathsf{S}_\mathsf{i}^\mathsf{c}), (\mathsf{s}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}})\big)$ and $\big((\mathsf{s}_\mathsf{r}^\mathsf{c}, \mathsf{S}_\mathsf{r}^\mathsf{c}), (\mathsf{s}_\mathsf{r}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}})\big)$ respectively. We also derive four DH secrets as in WireGuard, and three KEM shared secrets as in PQ-WireGuard*.

Table 4: Handshakes. type : 1 for InitHello, 2 for RespHello (4-byte values); lbl1, lbl2, lbl3: public constants; cookie : 0, except after receiving `CookieReply` message [36]. Colored instructions are blue for WireGuard [36], green for PQ-WireGuard [51] orange for PQ-WireGuard* and purple for Hybrid-WireGuard. KEM.encaps in PQ-WireGuard is a deterministic procedure, while KEM.encaps in PQ-WireGuard* and Hybrid-WireGuard is the standard probabilistic procedure (*c.f.*, Section 3.1). Key notations are summarized in Table 1.

	InitHello construction	**RespHello construction**
WG [36]	**input:** s_i^c, S_i^c, S_r^c 1: $sid_i \xleftarrow{\$} \{0,1\}^{32}$ 2: $(e_i^c, E_i^c) \leftarrow DH.gen()$ 3: $static \leftarrow AEAD.Enc(\kappa_3,\ 0,\ H_3,\ S_i^c)$ 4: $time \leftarrow AEAD.Enc(\kappa_4,\ 0,\ H_4,\ now())$ 5: $inner \leftarrow type \mid\mid sid_i \mid\mid E_i^c \mid\mid static \mid\mid time$ 6: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid S_r^c),\ inner)$ 7: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 8: $InitHello \leftarrow inner \mid\mid m1 \mid\mid m2$	**input:** s_r^c, S_r^c, S_i^c 1: $sid_r \xleftarrow{\$} \{0,1\}^{32}$ 2: $(e_r^c, E_r^c) \leftarrow DH.gen()$ 3: $empty \leftarrow AEAD.Enc(\kappa_9,\ 0,\ H_9,\ \varnothing)$ 4: $inner \leftarrow type \mid\mid sid_r \mid\mid sid_i \mid\mid E_r^c \mid\mid empty$ 5: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid S_i^c),\ inner)$ 6: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 7: $RespHello \leftarrow inner \mid\mid m1 \mid\mid m2$
PQ-WG [51]	**input:** $\sigma_i, s_i^{pq}, S_i^{pq}, S_r^{pq}$ 1: $sid_i \xleftarrow{\$} \{0,1\}^{32}$ 2: $r_i \xleftarrow{\$} \{0,1\}^{256}$ 3: $(e_i^{pq}, E_i^{pq}) \leftarrow KEM.gen()$ 4: $(ct_1, shk_1) \leftarrow KEM.Encaps(S_r^{pq},\ KDF_1(\sigma_i, r_i))$ 5: $static \leftarrow AEAD.Enc(\kappa_3,\ 0,\ H_3,\ HASH(S_i^{pq}))$ 6: $time \leftarrow AEAD.Enc(\kappa_4,\ 0,\ H_4,\ now())$ 7: $inner \leftarrow type \mid\mid sid_i \mid\mid E_i^{pq} \mid\mid ct_1 \mid\mid static \mid\mid time$ 8: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid S_r^{pq}),\ inner)$ 9: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 10: $InitHello \leftarrow inner \mid\mid m1 \mid\mid m2$	**input:** $\sigma_r, s_r^{pq}, S_r^{pq}, S_i^{pq}$ 1: $sid_r \xleftarrow{\$} \{0,1\}^{32}$ 2: $r_e,\ r_r \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$ 3: $(ct_2, shk_2) \leftarrow KEM.Encaps(E_i^{pq},\ r_e)$ 4: $(ct_3, shk_3) \leftarrow KEM.Encaps(S_i^{pq},\ KDF_1(\sigma_r, r_r))$ 5: $empty \leftarrow AEAD.Enc(\kappa_9,\ 0,\ H_9,\ \varnothing)$ 6: $inner \leftarrow type \mid\mid sid_r \mid\mid sid_i \mid\mid ct_2 \mid\mid ct_3 \mid\mid empty$ 7: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid S_i^{pq}),\ inner)$ 8: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 9: $RespHello \leftarrow inner \mid\mid m1 \mid\mid m2$
PQ-WG* (Section 5.2)	**input:** $s_i^{pq}, S_i^{pq}, S_r^{pq}$ 1: $sid_i \xleftarrow{\$} \{0,1\}^{32}$ 2: $(e_i^{pq}, E_i^{pq}) \leftarrow KEM.gen()$ 3: $(ct_1, shk_1) \leftarrow KEM.Encaps(S_r^{pq})\ //r_i \xleftarrow{\$} \{0,1\}^{256}$ 4: $static \leftarrow AEAD.Enc(\kappa_3,\ 0,\ H_3,\ HASH(S_i^{pq}))$ 5: $time \leftarrow AEAD.Enc(\kappa_4,\ 0,\ H_4,\ now())$ 6: $inner \leftarrow type \mid\mid sid_i \mid\mid E_i^{pq} \mid\mid ct_1 \mid\mid static \mid\mid time$ 7: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid psk),\ inner)$ 8: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 9: $InitHello \leftarrow inner \mid\mid m1 \mid\mid m2$	**input:** $s_r^{pq}, S_r^{pq}, S_i^{pq}$ 1: $sid_r \xleftarrow{\$} \{0,1\}^{32}$ 2: $(ct_2, shk_2) \leftarrow KEM.Encaps(E_i^{pq})\ //r_e \xleftarrow{\$} \{0,1\}^{256}$ 3: $(ct_3, shk_3) \leftarrow KEM.Encaps(S_i^{pq})\ //r_r \xleftarrow{\$} \{0,1\}^{256}$ 4: $empty \leftarrow AEAD.Enc(\kappa_9,\ 0,\ H_9,\ \varnothing)$ 5: $inner \leftarrow type \mid\mid sid_r \mid\mid sid_i \mid\mid ct_2 \mid\mid ct_3 \mid\mid empty$ 6: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid psk),\ inner)$ 7: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 8: $RespHello \leftarrow inner \mid\mid m1 \mid\mid m2$
Hybrid-WG (Section 6)	**input:** $s_i^c, S_i^c, S_r^c, s_i^{pq}, S_i^{pq}, S_r^{pq}$ 1: $sid_i \xleftarrow{\$} \{0,1\}^{32}$ 2: $(e_i^c, E_i^c) \leftarrow DH.gen()$ 3: $(e_i^{pq}, E_i^{pq}) \leftarrow KEM.gen()$ 4: $(ct_1, shk_1) \leftarrow KEM.Encaps(S_r^{pq})\ //r_i \xleftarrow{\$} \{0,1\}^{256}$ 5: $static \leftarrow AEAD.Enc(\kappa_3,\ 0,\ H_3,\ HASH(S_i^c \mid\mid S_i^{pq}))$ 6: $time \leftarrow AEAD.Enc(\kappa_4,\ 0,\ H_4,\ now())$ 7: $ct_1^{enc} \leftarrow SE.Enc(KDF_1(\varnothing,\ dh_{e_i s_r}),\ 0,\ ct_1)$ 8: $inner \leftarrow type \mid\mid sid_i \mid\mid E_i^c \mid\mid E_i^{pq} \mid\mid ct_1^{enc} \mid\mid static \mid\mid time$ 9: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid psk),\ inner)$ 10: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 11: $InitHello \leftarrow inner \mid\mid m1 \mid\mid m2$	**input:** $s_r^c, S_r^c, S_i^c, s_r^{pq}, S_r^{pq}, S_i^{pq}$ 1: $sid_r \xleftarrow{\$} \{0,1\}^{32}$ 2: $(e_r^c, E_r^c) \leftarrow DH.gen()$ 3: $(ct_2, shk_2) \leftarrow KEM.Encaps(E_i^{pq})\ //r_e \xleftarrow{\$} \{0,1\}^{256}$ 4: $(ct_3, shk_3) \leftarrow KEM.Encaps(S_i^{pq})\ //r_r \xleftarrow{\$} \{0,1\}^{256}$ 5: $empty \leftarrow AEAD.Enc(\kappa_9,\ 0,\ H_9,\ \varnothing)$ 6: $ct_3^{enc} \leftarrow SE.Enc(KDF_1(\varnothing,\ dh_{s_i e_r}),\ 0,\ ct_3)$ 7: $inner \leftarrow type \mid\mid sid_r \mid\mid sid_i \mid\mid E_r^c \mid\mid ct_2 \mid\mid ct_3^{enc} \mid\mid empty$ 8: $m1 \leftarrow MAC(HASH(lbl3 \mid\mid psk),\ inner)$ 9: $m2 \leftarrow MAC(cookie,\ inner \mid\mid m1)$ 10: $RespHello \leftarrow inner \mid\mid m1 \mid\mid m2$

During key derivation, both the DH secrets and the KEM secrets from WireGuard and PQ-WireGuard* respectively, are concatenated and used in Hybrid-WireGuard (terms C_3, C_4, C_7, C_8 and term κ_3 in Table 5). In addition, we include the necessary modifications to ensure resistance against UKS attacks: we use the concate-

Table 5: Key derivations. Steps 1, 9 are common to all constructions. Session keys are computed as $(tk_i, tk_r) = \mathsf{KDF}_2(C_9, \varnothing)$. Colored instructions are specific to WireGuard [36] (blue), to PQ-WireGuard [51] (green) and to PQ-WireGuard* (orange). Other notations are from Table 1. C_k and H_k are chaining and hash values, κ_k are symmetric keys.

	k	C_k	κ_k	H_k
	1	$\mathsf{HASH}(\mathsf{lbl1})$	-	$\mathsf{HASH}(C_1 \,\|\|\, \mathsf{lbl2})$
WG [36]	2	$\mathsf{KDF}_1(C_1, \mathsf{E}_\mathsf{i}^\mathsf{c})$	-	$\mathsf{HASH}(H_1 \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{c})$
	3	$\mathsf{KDF}_2(C_2, \mathsf{dh}_{\mathsf{e_i s_r}})[1]$	$\mathsf{KDF}_2(C_2, \mathsf{dh}_{\mathsf{e_i s_r}})[2]$	$\mathsf{HASH}(H_2 \,\|\|\, \mathsf{E}_\mathsf{i}^\mathsf{c})$
	4	$\mathsf{KDF}_2(C_3, \mathsf{dh}_{\mathsf{s_i s_r}})[1]$	$\mathsf{KDF}_2(C_3, \mathsf{dh}_{\mathsf{s_i s_r}})[2]$	$\mathsf{HASH}(H_3 \,\|\|\, \mathsf{InitHello.static})$
	5	-	-	$\mathsf{HASH}(H_4 \,\|\|\, \mathsf{InitHello.time})$
	6	$\mathsf{KDF}_1(C_4, \mathsf{E}_\mathsf{r}^\mathsf{c})$	-	$\mathsf{HASH}(H_5 \,\|\|\, \mathsf{E}_\mathsf{r}^\mathsf{c})$
	7	$\mathsf{KDF}_1(C_6, \mathsf{dh}_{\mathsf{e_i e_r}})$	-	-
	8	$\mathsf{KDF}_1(C_7, \mathsf{dh}_{\mathsf{s_i e_r}})$	-	-
PQ-WG [51]	2	$\mathsf{KDF}_1(C_1, \mathsf{E}_\mathsf{i}^\mathsf{pq})$	-	$\mathsf{HASH}(H_1 \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{pq})$
	3	$\mathsf{KDF}_2(C_2, \mathsf{shk}_1)[1]$	$\mathsf{KDF}_2(C_2, \mathsf{shk}_1)[2]$	$\mathsf{HASH}(H_2 \,\|\|\, \mathsf{E}_\mathsf{i}^\mathsf{pq})$
	4	$\mathsf{KDF}_2(C_3, \mathsf{psk})[1]$	$\mathsf{KDF}_2(C_3, \mathsf{psk})[2]$	$\mathsf{HASH}(H_3 \,\|\|\, \mathsf{InitHello.static})$
	5	-	-	$\mathsf{HASH}(H_4 \,\|\|\, \mathsf{InitHello.time})$
	6	$\mathsf{KDF}_1(C_4, \mathsf{ct}_2)$	-	$\mathsf{HASH}(H_5 \,\|\|\, \mathsf{ct}_2)$
	7	$\mathsf{KDF}_1(C_6, \mathsf{shk}_2)$	-	-
	8	$\mathsf{KDF}_1(C_7, \mathsf{shk}_3)$	-	-
PQ-WG* (Section 5.2)	2	$\mathsf{KDF}_1(C_1, \mathsf{E}_\mathsf{i}^\mathsf{pq})$	-	$\mathsf{HASH}(H_1 \,\|\|\, \mathsf{HASH}(\mathsf{S}_\mathsf{r}^\mathsf{pq}))$
	3	$\mathsf{KDF}_2(C_2, \mathsf{shk}_1)[1]$	$\mathsf{KDF}_2(C_2, \mathsf{shk}_1)[2]$	$\mathsf{HASH}(H_2 \,\|\|\, \mathsf{E}_\mathsf{i}^\mathsf{pq})$
	4	$\mathsf{KDF}_2(C_3, \mathsf{HASH}(\mathsf{S}_\mathsf{i}^\mathsf{pq} \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{pq}) \,\|\| \, \mathsf{psk})[1]$	$\mathsf{KDF}_2(C_3, \mathsf{HASH}(\mathsf{S}_\mathsf{i}^\mathsf{pq} \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{pq}) \,\|\| \, \mathsf{psk})[2]$	$\mathsf{HASH}(H_3 \,\|\|\, \mathsf{InitHello.static})$
	5	-	-	$\mathsf{HASH}(H_4 \,\|\|\, \mathsf{InitHello.time})$
	6	$\mathsf{KDF}_1(C_4, \mathsf{ct}_2)$	-	$\mathsf{HASH}(H_5 \,\|\|\, \mathsf{ct}_2)$
	7	$\mathsf{KDF}_1(C_6, \mathsf{shk}_2)$	-	-
	8	$\mathsf{KDF}_1(C_7, \mathsf{shk}_3)$	-	-
Hybrid-WG (Section 6)	2	$\mathsf{KDF}_1(C_1, \mathsf{E}_\mathsf{i}^\mathsf{c} \,\|\|\, \mathsf{E}_\mathsf{i}^\mathsf{pq})$	-	$\mathsf{HASH}(H_1 \,\|\|\, \mathsf{HASH}(\mathsf{S}_\mathsf{r}^\mathsf{c} \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{pq}))$
	3	$\mathsf{KDF}_2(C_2, \mathsf{dh}_{\mathsf{e_i s_r}} \,\|\|\, \mathsf{shk}_1)[1]$	$\mathsf{KDF}_2(C_2, \mathsf{dh}_{\mathsf{e_i s_r}} \,\|\|\, \mathsf{shk}_1)[2]$	$\mathsf{HASH}(H_2 \,\|\|\, \mathsf{E}_\mathsf{i}^\mathsf{c} \,\|\|\, \mathsf{E}_\mathsf{i}^\mathsf{pq})$
	4	$\mathsf{KDF}_2(C_3, \mathsf{dh}_{\mathsf{s_i s_r}} \,\|\| \, \mathsf{HASH}(\mathsf{S}_\mathsf{i}^\mathsf{pq} \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{pq}) \,\|\| \, \mathsf{psk})[1]$	$\mathsf{KDF}_2(C_3, \mathsf{dh}_{\mathsf{s_i s_r}} \,\|\| \, \mathsf{HASH}(\mathsf{S}_\mathsf{i}^\mathsf{pq} \,\|\|\, \mathsf{S}_\mathsf{r}^\mathsf{pq}) \,\|\| \, \mathsf{psk})[2]$	$\mathsf{HASH}(H_3 \,\|\|\, \mathsf{InitHello.static})$
	5	-	-	$\mathsf{HASH}(H_4 \,\|\|\, \mathsf{InitHello.time})$
	6	$\mathsf{KDF}_1(C_4, \mathsf{E}_\mathsf{r}^\mathsf{c} \,\|\|\, \mathsf{ct}_2)$	-	$\mathsf{HASH}(H_5 \,\|\|\, \mathsf{E}_\mathsf{r}^\mathsf{c} \,\|\|\, \mathsf{ct}_2)$
	7	$\mathsf{KDF}_1(C_6, \mathsf{dh}_{\mathsf{e_i e_r}} \,\|\|\, \mathsf{shk}_2)$	-	-
	8	$\mathsf{KDF}_1(C_7, \mathsf{dh}_{\mathsf{s_i e_r}} \,\|\|\, \mathsf{shk}_3)$	-	-
	9	$\mathsf{KDF}_3(C_8, \mathsf{psk})[1]$	$\mathsf{KDF}_3(C_8, \mathsf{psk})[3]$	$\mathsf{HASH}(H_6 \,\|\|\, \mathsf{KDF}_3(C_8, \mathsf{psk})[2])$

Table 6: Symbolic analysis Results. ✓ denotes a property unconditonally ensured, ✗ a property never ensured. Key notations are from Table 1 (blue: WireGuard, orange PQ-WireGuard⋆). For WireGuard, we use and complete results from [59] as it allows to consider the fix for anonymity proposed in [59]. FS denotes Forward Secrecy.

Property	**WireGuard** [59]	**PQ-WireGuard**	**PQ-WireGuard⋆**	**Hybrid-WireGuard**
UKS attacks resistance (ProVerif)				
Unilateral UKS (Init.)	✓	$\mathsf{psk} \wedge (\sigma_r \vee s_i^{pq}) \wedge (r_r \vee s_i^{pq}) \wedge (e_i^{pq} \vee r_e)$	✓	✓
Unilateral UKS (Resp.)	✓	$\mathsf{psk} \wedge (\sigma_i \vee s_r^{pq}) \wedge (r_i \vee s_r^{pq}) \wedge (e_i^{pq} \vee r_e)$	✓	✓
Bilateral UKS	✓	$e_i^{pq} \vee r_e$	✓	✓
Anonymity (DeepSec, ProVerif)				
Init.	$\mathsf{psk} \vee s_r^c \vee e_i^c$	✗	$\mathsf{psk} \vee s_i^{pq} \vee s_r^{pq} \vee r_r \vee r_i$	$\mathsf{psk} \bigvee (s_r^c \vee e_i^c) \bigwedge (s_r^{pq} \vee r_i) \bigvee (s_i^c \vee e_r^c) \bigwedge (s_i^{pq} \vee r_r)$
Resp.	$\mathsf{psk} \vee s_r^c \vee e_i^c$	✗	$\mathsf{psk} \vee s_r^{pq} \vee r_i$	$(\mathsf{psk} \vee s_r^c \vee e_i^c) \bigwedge (\mathsf{psk} \vee s_r^{pq} \vee r_i)$
Session uniqueness (ProVerif, Tamarin)				
Init./Resp.	✓	✓	✓	✓
Key (forward) secrecy, message agreement (ProVerif, Tamarin), **key strong secrecy** (ProVerif, DeepSec)				
Agreem. InitHello	$\mathsf{psk} \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c)$	psk	psk	$\mathsf{psk} \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c)$
Secrecy (Init.) Agreem. RespHello Key strong secrecy	$\mathsf{psk} \wedge (s_r^c \vee e_i^c) \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c)$	$\mathsf{psk} \wedge (s_r^{pq} \vee r_i) \wedge (s_r^{pq} \vee \sigma_i)$	$\mathsf{psk} \wedge (s_r^{pq} \vee r_i)$	$\mathsf{psk} \wedge (s_r^c \vee e_i^c) \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c) \bigwedge (s_r^{pq} \vee r_i)$
Secrecy (Resp.) Agreem. Confirm	$\mathsf{psk} \wedge (s_i^c \vee e_r^c) \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c)$	$\mathsf{psk} \wedge (s_i^{pq} \vee r_r) \wedge (s_i^{pq} \vee \sigma_r)$	$\mathsf{psk} \wedge (s_i^{pq} \vee r_r)$	$\mathsf{psk} \wedge (s_i^c \vee e_r^c) \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c) \bigwedge (s_i^{pq} \vee r_r)$
Secrecy (mutual) FS (Init. / Resp. / mutual)	$\mathsf{psk} \wedge (s_i^c \vee e_r^c) \wedge (s_r^c \vee e_i^c) \wedge (e_i^c \vee e_r^c) \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c)$	$\mathsf{psk} \wedge (s_i^{pq} \vee r_r) \wedge (s_i^{pq} \vee \sigma_r) \wedge (s_r^{pq} \vee r_i) \wedge (s_r^{pq} \vee \sigma_i) \wedge (e_i^{pq} \vee r_e)$	$\mathsf{psk} \wedge (s_i^{pq} \vee r_r) \wedge (s_r^{pq} \vee r_i) \wedge (e_i^{pq} \vee r_e)$	$\mathsf{psk} \wedge (s_i^c \vee e_r^c) \wedge (s_r^c \vee e_i^c) \wedge (e_i^c \vee e_r^c) \wedge (dh_{s_i s_r} \vee s_i^c \vee s_r^c) \bigwedge (s_i^{pq} \vee r_r) \wedge (s_r^{pq} \vee r_i) \wedge (e_i^{pq} \vee r_e)$

nation of the DH product from WireGuard, and the hash of the KEM public keys from PQ-WireGuard*, both used in the corresponding protocols to reach this resistance (terms C_4 and κ_4 in Table 5).
During the handshake, the static field results from an AEAD encryption of Initiator's DH static public key in WireGuard, and the hash of Initiator's KEM static public key in PQ-WireGuard* (as considering the public key would increase InitHello message size above the MTU limit [51]). In Hybrid-WireGuard, the static field is an encryption of the hash of Initiator's concatenated DH and KEM public keys.
For anonymity, another tweak is needed in the case of Hybrid-WireGuard to ensure that compromise of the property requires compromise of both DH and KEM static keys. During the PQ-WireGuard* handshake, if an attacker compromises the ephemeral randomness used during the encapsulation against $\mathsf{S}_\mathsf{r}^{\mathsf{pq}}$ in InitHello, then they can intercept ct_1 to lookup the correct static KEM key and reveal the identity of Responder. Similarly, if an attacker compromises the ephemeral randomness used during the encapsulation against $\mathsf{S}_\mathsf{i}^{\mathsf{pq}}$ in RespHello, then they can intercept ct_3 to lookup the correct static KEM key and reveal the identity of Initiator. This vulnerability is inherited by Hybrid-WireGuard when sending the same ciphertexts ct_1 and ct_3 respectively in InitHello and RespHello. To make up for this issue, we encrypt ct_1 and ct_3 with a symmetric encryption scheme, using as key the output of KDF_1 applied to the DH secret $\mathsf{dh}_{\mathsf{e_i s_r}}$ in the case of InitHello, and $\mathsf{dh}_{\mathsf{s_i e_r}}$ in the case of RespHello. Hence, instead of exchanging ct_1 and ct_3, Initiator sends ${\mathsf{ct}_1}^{\mathrm{enc}}$ created on line 7 for InitHello in Table 4 and ${\mathsf{ct}_3}^{\mathrm{enc}}$ created on line 6 for RespHello. The goal of this tweak is that even in the case of ephemeral randomness compromise, the attacker would still need to lookup all possible DH and KEM keys at the same time. Consequently, compromising only the DH static key or only the KEM key, along with ephemeral randomness compromise, is not enough to break anonymity. Note that ephemeral randomness compromise does not have the same effect in the case of PQ-WireGuard, since the secret is a result of the deterministic encapsulation, using random coins generated from the combination of ephemeral and long-term secret randomness.

Symbolic Analysis Results. For our assessment of Hybrid-WireGuard's security properties, we directly used obtained CNFs for WireGuard and PQ-WireGuard* and added the necessary evaluations to ensure soundness. The results of our symbolic analysis for Hybrid-WireGuard are presented side-by-side with other protocols in Table 6. We put forward that a necessary and sufficient condition to break a given property in our proposed Hybrid-WireGuard construction is to break the same property for WireGuard and for PQ-WireGuard*, with exactly the same set of keys involved. This is illustrated for instance for key secrecy from Initiator's view in Table 6, for WireGuard, PQ-WireGuard* and Hybrid-WireGuard. The obtained CNF for Hybrid-WireGuard is equal to the conjunction of the obtained CNFs for WireGuard and PQ-WireGuard*, which is exactly our target. More generally, an attack scenario against a security property of Hybrid-WireGuard is a combination of an attack on PQ-WireGuard* and an attack on WireGuard. The only exception is for anonymity, due to our solution: as shown in Table 4, line 7 for InitHello (resp. line 6 for RespHello), we encrypt ct_1 (resp. ct_3) with $\mathsf{dh}_{\mathsf{e_i s_r}}$ (resp. $\mathsf{dh}_{\mathsf{s_i e_r}}$) to ensure that breaking the property requires breaking DH and KEM keys.

7 Implementation

We provide an implementation in Rust of our new PQ-WireGuard* and Hybrid-WireGuard protocols, based on the original WireGuard Rust implementation [5]. Our implementations are available in our companion artifacts [60].

7.1 Instantiation of Cryptographic Primitives

In Table 7, we summarize the cryptographic instantiation in WireGuard [36], PQ-WireGuard [51] and our PQ-WireGuard* and Hybrid-WireGuard constructions. In Table 8, we give InitHello and RespHello sizes with respect to the chosen primitives. Note that in all protocols, the symmetric primitives are the same as in WireGuard, since quantum computing does not represent a considerable threat against them. The only exception is the symmetric encryption scheme exclusively used in Hybrid-WireGuard (for anonymity, *c.f.*, Section 6), which corresponds to the unauthenticated version of the AEAD scheme. Meanwhile, the DH key exchange in WireGuard is replaced by two post-quantum KEMs in PQ-WireGuard, one for the static keys, and one for the ephemeral keys (*c.f.*, Section 3). The authors of [51] choose KEMs such that InitHello and RespHello IPv6 packets do not exceed the maximum transmission unit (MTU) of 1280 bytes [34] to avoid fragmentation. Indeed, WireGuard relies on UDP and does not handle fragmentation. The authors additionally take other constraints into account. For instance, they restrict the choices to conservative primitives (in terms of security), that have advanced in the NIST competition and had the potential to be standardized ([51] was published before the final results of the competition), and those that provide L3 security. For the static long-term keys, a KEM with small ciphertext is needed, since the public keys are not exchanged during the handshake. In this case, the authors choose Classic McEliece [18], which has one of the smallest ciphertexts sizes among post-quantum KEM constructions. Meanwhile, its public key is very large (524160 bytes for L3 security). For the ephemeral KEM, the corresponding public key is sent in InitHello message, and the ciphertext is sent in RespHello message. Hence, both parameters

Table 7: Instantiation of cryptographic primitives.

Protocol	**DH**	**KEM**
WG [36]	X25519	-
PQ-WG [51]	-	Classic McEliece L3, Dagger L3
PQ-WG⋆	-	Classic McEliece L3, ML-KEM L1
Hybrid-WG	X25519	
Symmetric (for all protocols)		
AEAD	ChaCha20Poly1305	
HASH	Blake2s	
MAC	keyed−Blake2s	
KDF	HKDF with HMAC − Blake2s	
SE	ChaCha20 (only for Hybrid-WG)	

Table 8: Message sizes, with IPv6 and UDP headers.

Protocol	InitHello	RespHello
WG [36]	196	140
PQ-WG [51]	1248	1160
PQ-WG⋆	1124	1032
Hybrid-WG	1156	1064
Using ML-KEM L3 instead of L1		
PQ-WG⋆	1508	1352
Hybrid-WG	1540	1384
IPv6 MTU	**1280**	

must be small. The authors provide a tweaked construction of SABER [16], which they call Dagger. The latter is only IND-CPA secure, unlike SABER which is IND-CCA secure, but benefits from public key and ciphertext sizes smaller than those of SABER, and enough to fit in the handshake messages with L3 security. Table 8 exhibits the sizes of the messages with the chosen KEMs compared to the original WireGuard handshake. In our benchmarks, we update the Classic McEliece ciphertext size to match the current updated specifications [18]. In this case, it is equal to 156 bytes for L3 security, whereas in [51], it was equal to 188 bytes. Note that Classic McEliece is the only KEM suitable for the static keys, for the messages to fit into the IPv6 MTU, while taking into consideration the ephemeral KEM exchange.
For PQ-WireGuard⋆, as discussed in Section 5.2, we only rely on well-known KEM implementations. We choose Classic McEliece for the static keys, and ML-KEM, a standardized KEM, for the ephemeral keys. Meanwhile, we can only use ML-KEM with its parameters for L1 security and Classic McEliece for L3 security, without surpassing the IPv6 MTU (*c.f.*, Table 8). In the hybrid case, this L1 security matches the security level of the used Diffie-Hellman curve Curve25519, providing 128 bits of security.

7.2 Genericity and Performance

Genericity. We use liboqs [74] for the implementation of post-quantum KEMs, which is a known post-quantum library implemented in C. It is part of the Open-Quantum Safe project, grouping several research institutes and companies. In Rust, we use the latest interface to the C implementation, *i.e.*, oqs crate version 0.10.0. Note that the usage of liboqs can be easily replaced by another post-quantum library, and the chosen post-quantum KEMs can be replaced in the implementation. This provides our implementation with genericity, especially in the case of recent post-quantum constructions. In other words, if a KEM is broken, we can quickly replace it

in the implementation with another one, as long as the parameters sizes respect the constraints of WireGuard usage.

Performance. We measure the efficiency of PQ-WireGuard* and Hybrid-WireGuard handshakes, compared to the original WireGuard handshake. We focus on the construction of InitHello message by Initiator, and the processing of InitHello and construction of RespHello by Responder. We run the experiments on an Intel(R) Core(TM) i7-10510 CPU @1.80GHz. In Table 9, we show the average measured execution time for Initiator and Responder, along with the standard deviation. We observe from the execution time that the PQ-WireGuard* handshake is slightly faster than the original WireGuard handshake. This result shows that the liboqs implementation of the chosen primitives is very efficient. As for Hybrid-WireGuard, since most computation steps from WireGuard and PQ-WireGuard* are done in Hybrid-WireGuard, one cannot hope for Hybrid-WireGuard to be more efficient than the other two protocols. Instead, the best case would be that Hybrid-WireGuard's execution time be a sum of WireGuard's and PQ-WireGuard*'s execution times. We see through Table 9 that Hybrid-WireGuard Initiator and Responder execution times are slightly more efficient than this sum. In fact, the Hybrid-WireGuard handshake corresponds almost to a concatenation of both classic and post-quantum computations, except for some common steps which are not repeated, which explains the obtained results. Finally, recall that Rosenpass [76] aims to provide hybrid security by executing WireGuard, provided with pre-shared keys output from PQ-WireGuard (*c.f.*, Section 2). Rosenpass executes PQ-WireGuard every two minutes to refresh the pre-shared key, and the resulting hybrid handshake of 4 messages leads to more complexity and latency. Our 2-message construction is more efficient.

Table 9: Handshakes local performance. Initiator time: construction of InitHello; Responder time: processing of InitHello and construction of RespHello; execution time is averaged over several hundred executions; timings between parentheses are the standard deviations.

	Handshake performance (ms)	
Protocol	**Initiator**	**Responder**
WG [36]	0.49 (0.12)	1.11 (0.27)
PQ-WG*	0.44 (0.09)	0.84 (0.16)
Hybrid-WG	0.87 (0.18)	1.84 (0.38)

8 Conclusion and Future Work

Using automatic verification tools (ProVerif, DeepSec, Tamarin), we analyze the security of PQ-WireGuard, showing that the latter can be improved to reach anonymity and resilience against UKS attacks. We therefore propose an improved version, PQ-WireGuard*, for which these properties are ensured, without degrading other properties already reached by PQ-WireGuard (agreement, key secrecy, mutual secrecy, forward secrecy, session uniqueness). From WireGuard and PQ-WireGuard*, we construct Hybrid-WireGuard, that offers the best of the two worlds (classic and post-quantum), formally proving its security. We reach our hybridization target, because we show that an attack scenario against a security property of Hybrid-WireGuard is indeed a combination of an attack on PQ-WireGuard* and an attack on WireGuard. Eventually, we bridge the gap between theoretical design and practice, with a generic implementation of our constructions in Rust. Our benchmarks show the comparable efficiency of our constructions and ensure their usability.

Our work also shows the limitations of using post-quantum primitives in WireGuard, specifically related to the sizes of the exchanged messages. As WireGuard does not handle fragmentation, the packets' sizes must not exceed the IPv6 MTU limit (1280 bytes). Note that this problem is inherent to the post-quantum transition of any protocol that does not handle fragmentation. Hence we have two choices. First is to allow additional messages in the handshake. Clearly, this would render WireGuard non-competitive compared to other VPNs: one main advantage of WireGuard is precisely this 2-message handshake. Second, as in [51], is to keep this design. Here, we become limited for KEM algorithms, which is why Classic McEliece is the only KEM suitable for the static keys (even if we only aim for L1 security). For ephemeral keys, [51] tweaks an existing KEM construction for L3 security, while we argue that this is not a desired practice, so we restrict our design to trusted constructions and implementations. This limits us to use ML-KEM, with L1 security. We think that future research should analyze if L3 or L5 levels are reachable for 2-message handshakes.

Acknowledgments. This work is partly supported by ANR project SEVERITAS (ANR-20-CE39-0009), and ANR Project PRIVA-SIQ (ANR-23-CE39-0008).

Ethics Considerations

We strictly adhere to ethical norms, such as refraining from using any sensitive or personal data in this instance. We focused solely on publicly available information and theoretical models. Although the attack scenario described in Section 5.2 violates the claimed security of PQ-WireGuard, it is important to note that this attack scenario concerns protocol design and does not target any particular product that may embed PQ-WireGuard. Furthermore, we carefully considered the potential impact of our findings on security and privacy, and we believe that they do not pose any real risks.

Open Science

We emphasize that, in this work, we are strongly committed to the principle of open science. All our methodology, models, symbolic analysis, SAPIC$^+$ files used to generate protocols models in PROVERIF, TAMARIN and DEEPSEC, scripts to run verifiers, scripts to compute CNFs and Rust implementations are meticulously documented and openly shared through companion artifacts [60]. By doing so, we hope to contribute valuable insights to the community and aim to enhance the reproducibility and replicability of scientific findings.

Our companion artifacts [60] are composed of two folders. The first folder **artifacts_evaluation** concerns the symbolic analysis of WireGuard, WireGuard with fix for anonymity based on `psk` [59], PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard, as described in Section 4, Section 5 and Section 6. The second folder **artifacts_implementation** concerns our Rust implementation of WireGuard, PQ-WireGuard* and Hybrid-WireGuard, as described in Section 7. Each folder contains a `README.md` file that explains how to install all the dependencies (SAPIC$^+$, PROVERIF, TAMARIN, DEEPSEC used for symbolic verification, Python package `sympy` used for the CNF computations on the one hand, and Rust on the other hand). Our target is to ensure reproducibility of our results on a fresh Ubuntu Server 24.04.2 LTS [64].

References

[1] `https://kernelnewbies.org/Linux_5.6`.

[2] `https://lists.zx2c4.com/pipermail/wireguard/2021-August/006916.html`.

[3] `https://nvd.nist.gov/vuln/detail/CVE-2021-46873`.

[4] pq-strongswan. `https://github.com/strongX509/docker/tree/master/pq-strongswan`.

[5] wireguard-rs. `https://github.com/WireGuard/wireguard-rs`.

[6] ANSSI views on the Post-Quantum Cryptography transition, 2022. `https://cyber.gouv.fr/en/publications/anssi-views-post-quantum-cryptography-transition`.

[7] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *Journal of the ACM (JACM)*, 65(1):1 – 103, October 2017.

[8] Gorjan Alagic, David Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Daniel Apon. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process, 2022.

[9] Nouri Alnahawi, Johannes Müller, Jan Oupický, and Alexander Wiesmaier. A comprehensive survey on post-quantum TLS. *IACR Communications in Cryptology*, 1(2), 2024.

[10] Jacob Appelbaum, Chloe Martindale, and Peter Wu. Tiny WireGuard tweak. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*, volume 11627 of *Lecture Notes in Computer Science*, pages 3–20, Rabat, Morocco, July 9–11, 2019. Springer, Cham, Switzerland.

[11] Apple Security Engineering and Architecture (SEAR). iMessage with PQ3: The new state of the art in quantum-secure messaging at scale, 2024. `https://security.apple.com/blog/imessage-pq3/`.

[12] David Baelde, Alexandre Debant, and Stéphanie Delaune. Proving Unlinkability Using ProVerif Through Desynchronised Bi-Processes. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 75–90, 2023.

[13] David Baelde, Stéphanie Delaune, and Solène Moreau. A Method for Proving Unlinkability of Stateful Protocols. In *33rd IEEE Computer Security Foundations Symposium*, 33rd IEEE Computer Security

Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020, Boston, United States, June 2020.

[14] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Symbolically analyzing security protocols using Tamarin. *ACM SIGLOG News*, 4(4):19–30, November 2017.

[15] David Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1144–1155, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Andrea Basso, Jose Maria Bermudo, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR Based KEM, 2020. `https://www.esat.kuleuven.be/cosic/pqcrypto/saber/resources.html`.

[17] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45, Santa Barbara, CA, USA, August 23–27, 1998. Springer Berlin Heidelberg, Germany.

[18] Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Maram Varun, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Round-4 submission to the NIST PQC project, 2022. `https://classic.mceliece.org/nist.html`.

[19] Karthikeyan Bhargavan, Vincent Cheval, and Christopher A. Wood. A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 365–379, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[20] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024: 33rd USENIX Security Symposium*, Philadelphia, PA, USA, August 14–16, 2024. USENIX Association.

[21] Simon Blake-Wilson and Alfred Menezes. Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol. In Hideki Imai and Yuliang Zheng, editors, *PKC'99: 2nd International Workshop on Theory and Practice in Public Key Cryptography*, volume 1560 of *Lecture Notes in Computer Science*, pages 154–170, Kamakura, Japan, March 1–3, 1999. Springer Berlin Heidelberg, Germany.

[22] Bruno Blanchet. Security Protocol Verification: Symbolic and Computational Models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, pages 3–29, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2014.

[24] Bruno Blanchet. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[25] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. `https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf`.

[26] Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. A tale of two models: Formal verification of KEMTLS via Tamarin. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022: 27th European Symposium on Research in Computer Security, Part III*, volume 13556 of *Lecture Notes in Computer Science*, pages 63–83, Copenhagen, Denmark, September 26–30, 2022. Springer, Cham, Switzerland.

[27] Liqun Chen and Qiang Tang. Bilateral Unknown Key-Share Attacks in Key Agreement Protocols. *JUCS - Journal of Universal Computer Science*, 14(3):416–440, 2008.

[28] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *Proceedings of the Second International Conference on Principles of Security and Trust*, POST'13, page 226–246, Berlin, Heidelberg, 2013. Springer-Verlag.

[29] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. SAPIC+: protocol verifiers of the world, unite! In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022: 31st USENIX Security Symposium*, pages 3935–3952, Boston, MA, USA, August 10–12, 2022. USENIX Association.

[30] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: Deciding equivalence properties in security protocols theory and practice. In *2018 IEEE Symposium on Security and Privacy*, pages 529–546, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[31] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. Deepsec: Deciding equivalence properties for security protocols – improved theory and practice. *TheoretiCS*, Volume 3, March 2024.

[32] Véronique Cortier and Steve Kremer. Formal Models and Techniques for Analyzing Security Protocols: A Tutorial. *Foundations and Trends in Programming Languages*, 1(3):117, September 2014.

[33] Eric Crockett, Christian Paquin, and Douglas Stebila. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. In *NIST 2nd Post-Quantum Cryptography Standardization Conference 2019*, August 2019.

[34] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017.

[35] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[36] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society.

[37] Jason A. Donenfeld. WireGuard known limitations, 2024. `https://www.wireguard.com/known-limitations/`.

[38] Jason A. Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol, Draft Revision b956944, DOI: d376f649d7f4b68f616e05e5f64d660e9b23d7af, 2018. `https://www.wireguard.com/papers/wireguard-formal-verification.pdf`.

[39] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 3–21, Leuven, Belgium, July 2–4, 2018. Springer, Cham, Switzerland.

[40] Pasi Eronen, Yoav Nir, Paul E. Hoffman, and Charlie Kaufman. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, September 2010.

[41] European Telecommunications Standards Institute (ETSI). ETSI TS 103 744 V1.1.1 (2020-12): Quantum-safe Hybrid Key Exchanges, 2020. `https://www.etsi.org/deliver/etsi_ts/103700_103799/103744/01.01.01_60/ts_103744v010101p.pdf`.

[42] Federal Office for Information Security. Quantum Technologies and Quantum-Safe Cryptography, 2021. `https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Quantentechnologien-und-Post-Quanten-Kryptografie/quantentechnologien-und-post-quanten-kryptografie_node.html`.

[43] Scott Fluhrer, Panos Kampanakis, David McGrew, and Valery Smyslov. Mixing Preshared Keys in the Internet Key Exchange Protocol Version 2 (IKEv2) for Post-quantum Security. RFC 8784, June 2020.

[44] Markus Friedl, Jan Mojzis, and Simon Josefsson. Secure Shell (SSH) Key Exchange Method Using Hybrid Streamlined NTRU Prime sntrup761 and X25519 with SHA-512: sntrup761x25519-sha512. Internet-Draft draft-ietf-sshm-ntruprime-ssh-03, Internet Engineering Task Force, May 2025. Work in Progress.

[45] Atsushi Fujioka and Koutarou Suzuki. Sufficient condition for identity-based authenticated key exchange resilient to leakage of secret keys. In Howon Kim, editor, *ICISC 11: 14th International Conference on Information Security and Cryptology*, volume 7259 of *Lecture Notes in Computer Science*, pages 490–509, Seoul, Korea, November 30 – December 2, 2012. Springer Berlin Heidelberg, Germany.

[46] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012: 15th International Conference on Theory and Practice of Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 467–484, Darmstadt, Germany, May 21–23, 2012. Springer Berlin Heidelberg, Germany.

[47] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020: 29th USENIX Security Symposium*, pages 1857–1874. USENIX Association, August 12–14, 2020.

[48] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for unbounded verification of privacy-type properties. *CoRR*, abs/1710.02049, 2017.

[49] Ross Horne and Sjouke Mauw. Discovering epassport vulnerabilities using bisimilarity. *Logical Methods in Computer Science*, Volume 17, Issue 2, 02 2020.

[50] Russ Housley. TLS 1.3 Extension for Using Certificates with an External Pre-Shared Key. Internet-Draft draft-ietf-tls-8773bis-08, Internet Engineering Task Force, June 2025. Work in Progress.

[51] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Fiona Johanna Weber, and Philip R. Zimmermann. Post-quantum WireGuard. In *2021 IEEE Symposium on Security and Privacy*, pages 304–321, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[52] Andreas Hülsing, Kai-Chun Ning, Florian Weber, and Phil Zimmermann. Post-Quantum WireGuard. `https://cryptojedi.org/crypto/index.shtml`.

[53] Charlie Jacomme, Elise Klein, Steve Kremer, and Maïwenn Racouchot. A comprehensive, formal and automated analysis of the EDHOC protocol. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 5881–5898, Anaheim, CA, USA, August 9–11, 2023. USENIX Association.

[54] Panos Kampanakis, Douglas Stebila, and Torben Hansen. PQ/T Hybrid Key Exchange in SSH. Internet-Draft draft-kampanakis-curdle-ssh-pq-ke-03, IETF Secretariat, August 2024.

[55] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *2019 IEEE European Symposium on Security and Privacy*, pages 356–370, Stockholm, Sweden, June 17–19, 2019. IEEE Computer Society Press.

[56] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566, Santa Barbara, CA, USA, August 14–18, 2005. Springer Berlin Heidelberg, Germany.

[57] Ehren Kret and Rolfe Schmidt. The PQXDH key agreement protocol, revision 3, 2023. `https://signal.org/docs/specifications/pqxdh/`.

[58] Pascal Lafourcade, Dhekra Mahmoud, and Sylvain Ruhault. Artifacts for NDSS 2024 paper "A Unified Symbolic Analysis of WireGuard". `https://doi.org/10.5281/zenodo.10126618`.

[59] Pascal Lafourcade, Dhekra Mahmoud, and Sylvain Ruhault. A Unified Symbolic Analysis of WireGuard. In *ISOC Network and Distributed System Security Symposium – NDSS 2024*, San Diego, CA, USA, February 26 – March 1, 2024. The Internet Society.

[60] Pascal Lafourcade, Dhekra Mahmoud, Sylvain Ruhault, and Abdul Rahman Taleb. Artifacts for Usenix 2025 paper "A Tale of Two Worlds, a Formal Story of WireGuard Hybridization". `https://doi.org/10.5281/zenodo.15551056`.

[61] Felix Linker, Ralf Sasse, and David Basin. A formal analysis of apple's iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/1395, 2024.

[62] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 231–246, 2019.

[63] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43, 1997.

[64] Canonical Ltd. Get Ubuntu Server. `https://ubuntu.com/download/server`.

[65] Module-Lattice-Based Key-Encapsulation Mechanism Standard. National Institute of Standards and Technology, NIST FIPS PUB 203, U.S. Department of Commerce, August 2024.

[66] National Institute of Standards and Technology (NIST). NIST SP 1800-38: Migration to Post-Quantum Cryptography: Preparation for Considering the Implementation and Adoption of Quantum Safe Cryptography, 2023. `https://csrc.nist.gov/pubs/sp/1800/38/iprd-(1)`.

[67] NordVPN. NordLynx protocol – the solution for a fast and secure VPN connection. `https://nordvpn.com/blog/nordlynx-protocol-wireguard/`.

[68] Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking Post-quantum Cryptography in TLS. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 72–91, Paris, France, April 15–17, 2020. Springer, Cham, Switzerland.

[69] Trevor Perrin. The Noise Protocol Framework, Protocol Revision 34. 2018. `http://www.noiseprotocol.`

org/noise.html.

[70] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1461–1480, Virtual Event, USA, November 9–13, 2020. ACM Press.

[71] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Santa Fe, NM, USA, November 20–22, 1994. IEEE Computer Society Press.

[72] Dimitrios Sikeridis, Panos Kampanakis, and Devetsikiotis Michael. Post-quantum authentication in TLS 1.3: A performance study. In *Usenix Network and Distributed System Security Symposium*, San Diego (CA), United States, February 2020.

[73] Douglas Stebila. Security analysis of the iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/357, 2024.

[74] Douglas Stebila and Michele Mosca. Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 14–37, St. John's, NL, Canada, August 10–12, 2016. Springer, Cham, Switzerland.

[75] Cen Jung Tjhai, Martin Tomlinson, Graham Bartlett, Scott Fluhrer, Daniel Van Geest, Oscar Garcia-Morchon, and Valery Smyslov. Multiple Key Exchanges in the Internet Key Exchange Protocol Version 2 (IKEv2). RFC 9370, May 2023.

[76] Karolin Varner, Benjamin Lipp, Wanja Zaeke, Lisa Schmidt, and Prabhpreet Dua. Rosenpass, 2024. https://rosenpass.eu/whitepaper.pdf.

A InitHello Message Consumption

Table 10 describes Responder's InitHello message consumption modification to ensure Initiator anonymity (Section 5.2). For PQ-WireGuard*, m1 check happens *after* one KEM.Decaps and one AEAD.Dec computations and for Hybrid-WireGuard, it happens after one DH, one KEM.Decaps and one AEAD.Dec computations. Responder anonymity does not require changes of RespHello consumption as Initiator already knows Responder's identity.

Table 10: Responder's InitHello consumption by Responder.

WireGuard [36], anonymity not ensured

input: InitHello, s_r^c, S_r^c
1: parse InitHello as inner || m1 || m2
2: check m1 == MAC(HASH(lbl3 || S_r^c), inner)
3: parse inner as type || sid_i || E_i^c || static || time
4: compute $dh_{e_i s_r}$ // used to compute κ_3
5: $S_i^c \leftarrow$ AEAD.Dec(κ_3, 0, static)
6: lookup Initiator public keys and psk using S_i^c
7: AEAD.Dec(κ_4, 0, time)

PQ-WireGuard [51], anonymity not ensured

input: InitHello, s_r^{pq}, S_r^{pq}
1: parse InitHello as inner || m1 || m2
2: check m1 == MAC(HASH(lbl3 || S_r^{pq}), inner)
3: parse inner as type || sid_i || E_i^{pq} || ct_1 || static || time
4: $shk_1 \leftarrow$ KEM.Decaps(s_r^{pq}, ct_1)
5: h $\leftarrow$ AEAD.Dec(κ_3, 0, static) // h = HASH(S_i^{pq})
6: lookup Initiator public keys and psk using h
7: AEAD.Dec(κ_4, 0, time)

PQ-WireGuard*, anonymity ensured

input: InitHello, s_r^{pq}, S_r^{pq}
1: parse InitHello as inner || m1 || m2
2: parse inner as type || sid_i || E_i^{pq} || ct_1 || static || time
3: $shk_1 \leftarrow$ KEM.Decaps(s_r^{pq}, ct_1)
4: h $\leftarrow$ AEAD.Dec(κ_3, 0, static) // h = HASH(S_i^{pq})
5: lookup Initiator public keys and psk using h
6: check m1 == MAC(HASH(lbl3 || psk), inner)
7: AEAD.Dec(κ_4, 0, time)

Hybrid-WireGuard, anonymity ensured

input: InitHello, s_r^c, S_r^c, s_r^{pq}, S_r^{pq}
1: parse InitHello as inner || m1 || m2
2: parse inner as type || sid_i || E_i^c || E_i^{pq} || ${ct_1}^{enc}$ || static || time
3: compute $dh_{e_i s_r}$ // used to compute κ_3
4: $ct_1 \leftarrow$ SE.Dec($KDF_1(\varnothing, dh_{e_i s_r}), 0, {ct_1}^{enc}$)
5: $shk_1 \leftarrow$ KEM.Decaps(s_r^{pq}, ct_1)
6: h $\leftarrow$ AEAD.Dec(κ_3, 0, static) // h = HASH(S_i^c || S_i^{pq})
7: lookup Initiator public keys and psk using h
8: check m1 == MAC(HASH(lbl3 || psk), inner)
9: AEAD.Dec(κ_4, 0, time)

B Security properties formalization

Our trace properties are formalized as correspondence properties between **events**. Events are annotations that do not change the model, but are inserted at precise locations to allow reasoning about protocol execution. Table 11 describes all the queries used to formalize trace properties. These queries are given for PQ-WireGuard* protocol. The queries for WireGuard, PQ-WireGuard and Hybrid-WireGuard are the same with their dedicated set of keys. The **event** ISend is placed in the Initiator process just before the InitHello sending; the **event** RRec is placed in the Responder process after receiving and accepting the InitHello message from the Initiator; the **event** RKeys is placed in the Responder process just before RespHello sending; the **event** IKeys is in the Initiator process after receiving and accepting the RespHello message from the Responder; the **event** IConfirm is placed in the Initiator process just before the Confirm message sending; the **event** RConfirm is in the Responder process after receiving and accepting the Confirm message.

Table 11: Queries used to model trace properties, for PQ-WireGuard*. Key notations are from Table 1. Note that HASH is denoted H in this table. To obtain queries for WireGuard, PQ-WireGuard, Hybrid-WireGuard, keys shall be adapted: for WireGuard, keys are $\mathsf{psk}, \mathsf{S}_\mathsf{i}^\mathsf{c}, \mathsf{S}_\mathsf{r}^\mathsf{c}, \mathsf{S}_\mathsf{r}^{\mathsf{c}\prime}, \mathsf{E}_\mathsf{i}^\mathsf{c}, \mathsf{E}_\mathsf{i}^{\mathsf{c}\prime}$, for Hybrid-WireGuard, keys are $\mathsf{S}_\mathsf{i}^\mathsf{c}, \mathsf{S}_\mathsf{r}^\mathsf{c}, \mathsf{S}_\mathsf{r}^{\mathsf{c}\prime}, \mathsf{E}_\mathsf{i}^\mathsf{c}, \mathsf{E}_\mathsf{i}^{\mathsf{c}\prime}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}\prime}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}$ and finally, for PQ-WireGuard, $\mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r})$ shall be replaced by $\mathsf{H}(\sigma_\mathsf{i}, \mathsf{r_i}), \mathsf{H}(\sigma_\mathsf{r}, \mathsf{r_r})$ for Agreement and Secrecy properties.

UKS	$\forall\ \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}\prime}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{psk}, \mathrm{ck}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e};$
Initiator unilateral	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \wedge$ $\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}\prime}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{psk}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \Rightarrow \mathsf{S}_\mathsf{r}^{\mathsf{pq}} = \mathsf{S}_\mathsf{r}^{\mathsf{pq}\prime}.$
Responder unilateral	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \wedge$ $\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \Rightarrow \mathsf{S}_\mathsf{i}^{\mathsf{pq}} = \mathsf{S}_\mathsf{i}^{\mathsf{pq}\prime}.$
Bilateral	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \wedge$ $\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}\prime}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \Rightarrow (\mathsf{S}_\mathsf{i}^{\mathsf{pq}} = \mathsf{S}_\mathsf{i}^{\mathsf{pq}\prime}) \wedge (\mathsf{S}_\mathsf{r}^{\mathsf{pq}} = \mathsf{S}_\mathsf{r}^{\mathsf{pq}\prime}).$
Uniqueness	$\forall\ \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{psk}, \mathrm{ck}, \mathsf{k_i}, \mathsf{k_i}', \mathsf{k_r}, \mathsf{k_r}', \mathsf{k_e}, \mathsf{k_e}';$
Initiator	$\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \wedge$ $\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{psk}, \mathrm{ck}, \mathsf{k_i}', \mathsf{k_r}, \mathsf{k_e}))\big) \Rightarrow (\mathsf{E}_\mathsf{i}^{\mathsf{pq}} = \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}) \wedge (\mathsf{k_i} = \mathsf{k_i}').$
Responder	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{k_i}, \mathsf{k_r}, \mathsf{k_e}))\big) \wedge$ $\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}, \mathsf{psk}, \mathrm{ck}, \mathsf{k_i}, \mathsf{k_r}', \mathsf{k_e}'))\big) \Rightarrow (\mathsf{E}_\mathsf{i}^{\mathsf{pq}} = \mathsf{E}_\mathsf{i}^{\mathsf{pq}\prime}) \wedge (\mathsf{k_r} = \mathsf{k_r}') \wedge (\mathsf{k_e} = \mathsf{k_e}').$
Agreement	$\forall\ \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{r_i}, \mathsf{r_r}, \mathsf{r_e};$
InitHello	$\big(\mathbf{event}(\mathrm{RRec}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big) \Rightarrow \big(\mathbf{event}(\mathrm{ISend}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{H}(\mathsf{r_i})))\big).$
RespHello	$\big(\mathbf{event}(\mathrm{IKeys}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big)$ $\Rightarrow \big(\mathbf{event}(\mathrm{RKeys}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big).$
Confirm	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big)$ $\Rightarrow \big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big).$
Secrecy	$\forall\ \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{r_i}, \mathsf{r_r}, \mathsf{r_e};$
Init.'s view	$\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big) \wedge \mathbf{attacker}(\mathrm{ck}).$
Resp.'s view	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big) \wedge \mathbf{attacker}(\mathrm{ck}).$
Mutual	$\big(\mathbf{event}(\mathrm{RConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big) \wedge$ $\big(\mathbf{event}(\mathrm{IConfirm}(\mathrm{ck}, \mathsf{S}_\mathsf{i}^{\mathsf{pq}}, \mathsf{S}_\mathsf{r}^{\mathsf{pq}}, \mathsf{E}_\mathsf{i}^{\mathsf{pq}}, \mathsf{psk}, \mathrm{ck}, \mathsf{H}(\mathsf{r_i}), \mathsf{H}(\mathsf{r_r}), \mathsf{H}(\mathsf{r_e})))\big) \wedge \mathbf{attacker}(\mathrm{ck}).$

C Artifact Appendix

C.1 Abstract

These artifacts accompany research paper "A Tale of Two Worlds, a Formal Story of WireGuard Hybridization". They allow to reproduce all results described in this paper, concerning symbolic evaluation of security properties and implementation of protocols related to hybridization of WireGuard. These artifacts are composed of two folders. The first folder **artifacts_implementation** concerns our Rust implementation of WireGuard, PQ-WireGuard* and Hybrid-WireGuard. The second folder **artifacts_evaluation** concerns our symbolic analysis of WireGuard (with fix for anonymity based on psk), PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard, with the help of Tamarin, ProVerif and DeepSec verifiers.

C.2 Description & Requirements

C.2.1 Security, privacy, and ethical concerns

None.

C.2.2 How to access

Our artifacts are available on a permanent public repository[1]. To access them, execute (MacOS may require to remove \):

```
record_url=$(curl -Ls -o /dev/null -w '%{url_effective}' \
https://doi.org/10.5281/zenodo.15551056)
curl -L -o artifacts-usenix-2025-wireguard-hybridization.zip \
"$record_url/files/artifacts-usenix-2025-wireguard-hybridization."\
"zip?download=1"
unzip artifacts-usenix-2025-wireguard-hybridization
cd artifacts-usenix-2025-wireguard-hybridization
```

C.2.3 Hardware dependencies

Our artifacts do not require any specific hardware. All results are reproducible on a standard laptop with 12 cores of CPU 1.8 GHz and 16 Go of RAM.

C.2.4 Software dependencies

Our artifacts require the use of Docker Engine[2]. For our tests, we used Docker Engine version 28.2.2 and we successfully tested Docker deployment on both Linux Ubuntu 24.04.2 LTS and MacOS Sequoia 15.4. We use Tamarin release version 1.10.0, Maude version 3.5, ProVerif version 2.05, DeepSec version 2.0.2, Python version 3.12.3, Package sympy, GNU parallel, Clang version 18.1.3 and Rust version 1.87.0. All these dependencies are installed automatically when building Dockerfile (Section C.3.1).

C.2.5 Benchmarks

Our artifacts allow to reproduce similar benchmarks as the one given in our paper (Table 9), to compare the efficiency of WireGuard, PQ-WireGuard* and Hybrid-WireGuard. Our benchmarks focus on the construction of InitHello message by Initiator, and on the processing of InitHello and construction of RespHello by Responder. Execution time is averaged on a chosen number of executions of these operations, that is a parameter of the benchmark (Section C.4.2).

C.3 Set-up

C.3.1 Installation

Before building Docker, it is adviced to perform a complete cleanup of Docker environment. For this, execute (warning: this cleanup stops and deletes *all* Docker containers, images, volumes, networks):

```
sh run_docker_clean.sh
```

Then, execute:

[1] https://doi.org/10.5281/zenodo.15551056
[2] https://docs.docker.com/engine/install

```
docker buildx build --platform linux/x86_64 -t hyb-wg .
```

This should build the Docker image with all dependencies (Section C.2.4). Docker build takes around 30 minutes.

C.3.2 Basic Test

To check Docker image, execute:

```
docker run -it hyb-wg
```

Then inside Docker, execute:

```
sh run_basic_test.sh
```

This should launch, successively, Tamarin, ProVerif, DeepSec verifiers and Cargo Rust, hence their respective versions shall successively appear (Section C.2.4). Furthermore, this basic test launches evaluations of trace properties (resistance against unilateral and bilateral unknown key share attacks, session uniqueness, agreement and secrecy, including PFS) for PQ-WireGuard* protocol. A set of CNFs shall be printed, as in Table 13, equal to the ones described in Table 6 of our paper, for PQ-WireGuard* protocol. This basic test shall finish in less than 2 minutes.

C.4 Evaluation workflow

C.4.1 Major Claims

(C1): We propose a Rust implementation of protocols WireGuard, PQ-WireGuard*, Hybrid-WireGuard. This is proven by the experiment (**E1**) described in Section 7.2 of our paper, with results reported in Table 9 of our paper.

(C2): We propose, with Tamarin, ProVerif and DeepSec verifiers, symbolic proofs of trace properties (session uniqueness, resistance against UKS attacks, message agreement and secrecy, including PFS) or observational equivalence properties (anonymity, strong secrecy), for WireGuard (with fix for anonymity based on psk), PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard. This is proven by experiments (**E2**), (**E3**), (**E4**), described in Sections 4, 5, 6 of our paper, whose results are reported in Table 6 of our paper.

C.4.2 Experiments

(E1): *[Efficiency evaluation] [1 human-minutes + 10 compute-minutes]: based on Rust implementation, build an application that mesures efficiency of construction of InitHello message by Initiator, and processing of InitHello and construction of RespHello message by Responder.*
Preparation: Inside Docker image (Section C.3.2), in folder `artifacts_implementation`, file `run_all.sh` contains line `cargo run -- -b 100`: value `100` is the chosen number for the benchmarks, as described in Section C.2.5.
Execution: Run `sh run_all.sh`.
Results: Printed results shall be as in Table 12, where number of executions is the one chosen for benchmarks (Section C.2.5), message sizes correspond to Table 8 of our paper and timings correspond to Table 9 of our paper.

(E2): *[Trace properties evaluation] [1 human-minutes + 4 compute-hours]: based on* Sapic$^+$ *files, generate all* ProVerif *and* Tamarin *files and evaluate trace properties (session uniqueness, resistance against UKS attacks, message agreement and secrecy, including PFS) for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard*, Hybrid-WireGuard.*
Preparation: Inside Docker image (Section C.3.2), in folder `artifacts_evaluation`, the file named `run_trace_properties.sh` contains all necessary commands that allows to, sequentially, perform all evaluations. These use ProVerif verifier, for all protocols, and Tamarin verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard. We did not use Tamarin verifier for PQ-WireGuard and PQ-WireGuard* as these protocols do not rely on `diffie-hellman` builtin.
Execution: Run `sh run_trace_properties.sh`.
Results: Printed results shall be as in Table 13. These results correspond to Table 6 of our paper. To check Tamarin results for WireGuard (with fix for anonymity based on psk), execute:

```
cd wireguard_with_fix_psk/trace_properties/tamarin
cat *.spthy.log | grep "verified\|falsified"
```

All evaluated properties shall be `verified`. To check Tamarin results for Hybrid-WireGuard, execute:

```
cd hybrid_wireguard/trace_properties_necessary_conditions
cat *.spthy.log | grep "verified\|falsified"
```

All evaluated properties shall be `verified`. Finally, check sufficient conditions for Hybrid:

```
cd hybrid_wireguard/trace_properties_sufficient_conditions
sh run_check-pv.sh
```

All queries shall be `false` except for `uuks` and `buks` which shall be `true`.

Table 12: Results for Experiment (**E1**).

```
Average time over 100 executions

WireGuard:
InitHello message size: 196 bytes
RespHello message size: 140 bytes
InitHello construction time: 0.230 ms (std = 0.061 ms)
InitHello consumption time + RespHello construction time: 0.507 ms (std = 0.148 ms)

PQ-WireGuard*: (static kem: Classic-McEliece-460896, ephemeral kem: ML-KEM-512)
InitHello message size: 1124 bytes
RespHello message size: 1032 bytes
InitHello construction time: 0.326 ms (std = 0.168 ms)
InitHello consumption time + RespHello construction time: 0.612 ms (std = 0.148 ms)

Hybrid-WireGuard: (static kem: Classic-McEliece-460896, ephemeral kem: ML-KEM-512)
InitHello message size: 1156 bytes
RespHello message size: 1064 bytes
InitHello construction time: 0.481 ms (std = 0.074 ms)
InitHello consumption time + RespHello construction time: 1.211 ms (std = 0.178 ms)
```

Table 13: Results for Experiment (**E2**).

WireGuard (with fix for anonymity based on psk)

```
bilateral: ∅
unilateral_initiator: ∅
unilateral_responder: ∅
uniqueness_initiator: ∅
uniqueness_responder: ∅
agreement_inithello: psk & (dhsisr | sic | src)
agreement_rechello: psk & (src | eic) & (dhsisr | sic | src)
agreement_confirm: psk & (sic | erc) & (dhsisr | sic | src)
secrecy_isk7: psk & (src | eic) & (dhsisr | sic | src)
secrecy_rsk7: psk & (sic | erc) & (dhsisr | sic | src)
secrecy_mul7: psk & (sic | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_isk7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_rsk7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_mul7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
```

PQ-WireGuard

```
uniqueness_initiator: ∅
bilateral: eipq | re
uniqueness_responder: ∅
unilateral_initiator: psk & (sipq | rr) & (sipq | sigr) & (eipq | re)
secrecy_isk7: psk & (srpq | ri) & (srpq | sigi)
unilateral_responder: psk & (srpq | ri) & (srpq | sigi) & (eipq | re)
agreement_inithello: psk
secrecy_rsk7: psk & (sipq | rr) & (sipq | sigr)
agreement_rechello: psk & (srpq | ri) & (srpq | sigi)
agreement_confirm: psk & (sipq | rr) & (sipq | sigr)
secrecy_isk7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (srpq | sigi) & (eipq | re)
secrecy_mut7: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (srpq | sigi) & (eipq | re)
secrecy_rsk7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (srpq | sigi) & (eipq | re)
secrecy_mut7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (srpq | sigi) & (eipq | re)
```

PQ-WireGuard*

```
unilateral_responder: ∅
bilateral: ∅
uniqueness_responder: ∅
uniqueness_initiator: ∅
unilateral_initiator: ∅
secrecy_isk7: psk & (srpq | ri)
agreement_confirm: psk & (sipq | rr)
agreement_rechello: psk & (srpq | ri)
secrecy_rsk7: psk & (sipq | rr)
secrecy_mut7: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
agreement_inithello: psk
secrecy_isk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
secrecy_mut7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
secrecy_rsk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
```

Hybrid-WireGuard

```
unilateral_responder: ∅
unilateral_initiator: ∅
bilateral: ∅
uniqueness_initiator: ∅
uniqueness_responder: ∅
secrecy_isk7: psk & (srpq | ri) & (src | eic) & (dhsisr | sic | src)
agreement_inithello: psk & (dhsisr | sic | src)
secrecy_rsk7: psk & (sipq | rr) & (sic | erc) & (dhsisr | sic | src)
agreement_rechello: psk & (srpq | ri) & (src | eic) & (dhsisr | sic | src)
agreement_confirm: psk & (sipq | rr) & (sic | erc) & (dhsisr | sic | src)
secrecy_mut7: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic | erc) & (src | eic) & (eic |
    erc) & (dhsisr | sic | src)
secrecy_isk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic | erc) & (src | eic) & (eic
    | erc) & (dhsisr | sic | src)
secrecy_rsk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic | erc) & (src | eic) & (eic
    | erc) & (dhsisr | sic | src)
secrecy_mut7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic | erc) & (src | eic) & (eic
    | erc) & (dhsisr | sic | src)
```

(E3): *[Anonymity evaluation] [1 human-minutes + 4 compute-hours]: evaluate all* ProVerif *and* DeepSec *files that model anonymity for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard⋆, Hybrid-WireGuard.*

Preparation: Inside Docker image (Section C.3.2), in folder `artifacts_evaluation`, the file named `run_anonymity.sh` contains all necessary commands that allows to, sequentially, evaluate all files for anonymity for all protocols. These evaluations use ProVerif verifier, for all protocols, and DeepSec verifier for PQ-WireGuard and PQ-WireGuard⋆. We could not use DeepSec verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard as DeepSec does not handle exponentiation.

Execution: Run `sh run_anonymity.sh`.

Results for WireGuard (with fix for anonymity based on psk). For ProVerif, execute:

```
cd wireguard_with_fix_psk/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Eic`, `-Psk`, `-Src` and for files `responder`, `-Eic`, `-Psk`, `-Src` and shall be `Observational equivalence is true` for files `initiator -no-reveal`, `-Erc-Sic` and for files `responder`, `-no-reveal`, `-Erc-Sic`.

Results for PQ-WireGuard. For DeepSec, execute:

```
cd pq_wireguard/equivalence_properties/Anonymity
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for all files.

Results for PQ-WireGuard⋆. For DeepSec, execute:

```
cd pq_wireguard_star/equivalence_properties/Anonymity
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard_star/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Psk`, `-Ri`, `-Rr`, `-Siq`, `-Srq` and for files `responder`, `-Psk`, `-Ri`, `-Srq` and shall be `Observational equivalence is true` for files `initiator`, `-Eiq-Re` and for file `responder`, `-Siq-Rr-Eiq-Re`.

Results for Hybrid-WireGuard. For ProVerif, execute:

```
cd hybrid_wireguard/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Psk`, `-Eic-Ri`, `-Eic-Srq`, `-Erc-Siq`, `-Erc-Rr`, `-Sic-Rr`, `-Sic-Siq`, `-Src-Ri`, `-Src-Srq` and for files `responder`, `-Psk`, `-Eic-Ri`, `-Src-Ri`, `-Src-Srq`, `-Src-Eic` and shall be `Observational equivalence is true` for files `initiator`, `-all-dh`, `-all-pq`, `-no-reveal`, `-Sic -Erc-Srq-Eiq-Ri-Re`, `-Sic-Src-Eic-Erc-Eiq-Re`, `-Src-Eic-Siq-Eiq-Rr-Re` and for files `res- ponder`, `-all-dh`, `-all-pq`, `-no-reveal`, `-Sic-Erc-Siq-Srq -Eiq-Ri-Rr-Re`, `-Sic-Src-Eic-Erc -Siq-Eiq-Rr-Re`.

(E4): *[Strong secrecy evaluation] [2 human-minutes + 10 compute-hours]: evaluate all* ProVerif *and* DeepSec *files that model strong secrecy for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard⋆, Hybrid-WireGuard.*

Preparation: Inside Docker image (Section C.3.2), in folder `artifacts_evaluation`, the file named `run_strong_secrecy.sh` contains all necessary commands that allows to, sequentially, perform all evaluations. These use ProVerif verifier, for all protocols, and DeepSec verifier for PQ-WireGuard and PQ-WireGuard⋆. We could not use DeepSec verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard as DeepSec does not handle exponentiation.

Execution: Run `sh run_strong_secrecy.sh`.

Results for WireGuard (with fix for anonymity based on psk). For ProVerif, execute:

```
cd wireguard_with_fix_psk/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Src`, `-Psk-DH-Eic`, `-Psk -Sic-Eic` and shall be `Observational equivalence is true` for files `-Psk-Eic-Erc`, `-Psk-Sic-Erc`, `-Sic-Src-Eic-Erc`.

Results for PQ-WireGuard. For DeepSec, execute:

```
cd pq_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-dps.sh
```

Evaluation result shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Srq`, `-Psk-Ri-Ti` and shall be `Observational equivalence is true` for files `-Psk-Siq-Ri-Rr-Re-Eiq-Tr`, `-Psk -Siq-Rr-Re -Ei-Tr-Ti`, `-reveal-all-but-Psk`.

Results for PQ-WireGuard*. For DeepSec, execute:

```
cd pq_wireguard_star/equivalence_properties/Strong-Secrecy
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard_star/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Ri`, `-Psk-Srq` and shall be `Observational equivalence is true` for files `-Psk-Siq-Eiq-Rr-Re`, `-Siq-Srq-Eiq -Ri-Rr-Re`.

Results for Hybrid-WireGuard. For ProVerif, execute:

```
cd hybrid_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-DH-Eic-Ri`, `-Psk-DH-Eic -Srq`, `-Psk-Sic-Eic-Ri`, `-Psk-Sic -Eic-Srq`, `-Psk-Src-Ri`, `-Psk-Src-Srq` and shall be `Observational equivalence is true` for files `-all-dh`, `-all-pq`, `-no-reveal`, `-Sic-Src -Eic-Erc-Siq-Srq-Eiq-Ri -Rr-Re`.

C.5 Notes on Reusability

Our artifacts can be extended to benchmark and evaluate new versions (e.g. fixes) of the protocols we evaluate in our paper.

C.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at the following link: `https://secartifacts.github.io/usenixsec2025/`.