Noise Explorer:
Fully Automated Modeling and Verification for Arbitrary Noise Protocols

Nadim Kobeissi
INRIA Paris, Symbolic Software
nadim@symbolic.software

Georgio Nicolas
Symbolic Software
georgio@symbolic.software

Karthikeyan Bhargavan
INRIA Paris
karthikeyan.bhargavan@inria.fr

April 16, 2019

Abstract

The Noise Protocol Framework, introduced recently, allows for the design and construction of secure channel protocols by describing them through a simple, restricted language from which complex key derivation and local state transitions are automatically inferred. Noise “Handshake Patterns” can support mutual authentication, forward secrecy, zero round-trip encryption, identity hiding and other advanced features. Since the framework’s release, Noise-based protocols have been adopted by WhatsApp, WireGuard and other high-profile applications.

We present Noise Explorer, an online engine for designing, reasoning about, formally verifying and implementing arbitrary Noise Handshake Patterns. Based on our formal treatment of the Noise Protocol Framework, Noise Explorer can validate any Noise Handshake Pattern and then translate it into a model ready for automated verification and also into a production-ready software implementation written in Go or in Rust. We use Noise Explorer to analyze more than 57 handshake patterns. We confirm the stated security goals for 12 fundamental patterns and provide precise properties for the rest. We also analyze unsafe handshake patterns and document weaknesses that occur when validity rules are not followed. All of this work is consolidated into a usable online tool that presents a compendium of results and can parse formal verification results to generate detailed-but-pedagogical reports regarding the exact security goals of each message of a Noise Handshake Pattern with respect to each party, under an active attacker and including malicious principals. Noise Explorer evolves alongside the standard Noise Protocol Framework, having already contributed new security goal verification results and stronger definitions for pattern validation and security parameters.

1 Introduction

Popular Internet protocols such as SSH and TLS use similar cryptographic primitives: symmetric primitives, public key primitives, one-way hash functions and so forth. Protocol stages are also similarly organized, usually beginning with an authenticated key exchange (AKE) stage followed by a messaging stage. And yet, the design methodology, underlying state machine transitions and key derivation logic tend to be entirely different between protocols with nevertheless similar building blocks. The targeted effective security goals tend to be similar, so why can’t the same methodology be followed for everything else?

Standard protocols such as those mentioned above choose a specific set of key exchange protocols to satisfy some stated use-cases while leaving other elements, such as round trips and (notoriously) cipher suites up to the deployer. Specifications use protocol-specific verbose notation to describe the underlying protocol, to the extent that even extracting the core cryptographic protocol becomes hard, let alone analyzing and comparing different modes for security.

Using completely different methodologies to build protocols that nevertheless often share the same primitives and security goals is not only unnecessary, but provably dangerous. The Triple Handshake attack on TLS published in 2014 [1] is based on the same logic that made the attack [2] on the Needham-Schroeder protocol [3] possible almost two decades earlier.

The core protocol in TLS 1.2 was also vulnerable to a similar attack, but since the protocol itself is hidden within layers of packet formats and C-like pseudocode, it was difficult for the attack to be detected. However, upon automated symbolic verification [4, 5], the attack quickly appeared not just in TLS, but also in variants of SSH and IPsec. Flaws underlying more recent attacks such as Logjam [6] were known for years before they were observed when the vulnerable protocol was analyzed. Had these protocols differed only in terms of network messages while still using a uniform, formalized logic for internal key derivation and state machine transitioning designed based on the state of the art of protocol analysis, these attacks could have been avoided.

1.1 The Noise Protocol Framework

$$
\begin{aligned}
&IK:\\
&\qquad \leftarrow s\\
&\qquad \ldots\\
&\qquad \rightarrow e, es, s, ss\\
&\qquad \leftarrow e, ee, se
\end{aligned}
$$

Figure 1: An example Noise Handshake Pattern, `IK`.

The Noise Protocol Framework [7], recently introduced by Trevor Perrin, aims to avert this problem by presenting a simple language for describing cryptographic network protocols. In turn, a large number of semantic rules extend this simple protocol description to provide state machine transitions, key derivation logic and so on. The goal is to obtain the strongest possible effective security guarantees for a given protocol based on its description as a series of network messages by deriving its other elements from a uniform, formally specified logic followed by all protocol designs.

In designing a new secure channel protocol using Noise, one only provides an input using the simple language shown in Fig. 1. As such, from the viewpoint of the protocol designer, Noise protocols can only differ in the number of messages, the types of keys exchanged and the sequence or occurrence of public key transmissions and Diffie-Hellman operations. Despite the markedly non-expressive syntax, however, the occurence and position of the "tokens" in each message pattern can trigger complex state machine evolutions for both parties, which include operations such as key derivation and transcript hash mixing.

Let's examine Fig. 1. Before the AKE begins, the responder shares his static public key. Then in the first protocol message, the initiator sends a fresh ephemeral key, calculates a Diffie-Hellman shared secret between her ephemeral key and the recipient's public static key, sends her public static key and finally calculates a Diffie-Hellman shared secret between her static key and the responder's public static key. The responder then answers by generating an ephemeral key pair and sending his ephemeral public key, deriving a Diffie-Hellman shared secret between his ephemeral key and the ephemeral key of the initiator and another Diffie-Hellman shared secret between his static key and the ephemeral key of the initiator. Both of these AKE messages can also contain message payloads, which, depending on the availability of sufficient key material, could be AEAD-encrypted (in this particular handshake pattern, this is indeed the case).

As we can see, quite a few operations have occured in what would at first glance appear to be simple syntax for a simple protocol. Indeed, underlying these operations is a sophisticated state machine logic tasked with mixing all of the derived keys together, determining when it is safe (or possible) to send encrypted payloads and ensuring transcript consistency, among other things. This is the value of the Noise Protocol Framework: allowing the protocol designer to describe what they need their protocol to do fairly effectively using this simple syntax, and leaving the rest to a sturdy set of underlying rules.

1.2 Noise Explorer: Formal Verification for any Noise Handshake Pattern

Noise Explorer, the central contribution of this work, capitalizes on the strengths of the Noise Protocol Framework in order to allow for automated protocol verification to no longer be limited only to monolithic, pre-defined protocols with their own notation. In this work, we formalize Noise's syntax, semantics, state transitions and handshake pattern validity rules. We then present translation logic to go from handshake patterns directly into full symbolic models ready for automated verification using the ProVerif [8, 9] automatic protocol verifier.

$IN:$

$\rightarrow e, s$

$\leftarrow e, ee, se$

Figure 2: An example Noise Handshake Pattern, `IN`.

This allows us to then construct Noise Explorer, an online engine that allows for designing, validating and subsequently generating cryptographic models for the automated formal verification of any arbitrary Noise Handshake Pattern. Models generated using Noise Explorer allow for the verification of Noise-based secure channel protocols against a battery of comprehensive ProVerif queries. Noise Explorer also comes with the first compendium of formal verification results for handshake patterns, browsable online using an interactive web application that presents dynamically generated diagrams indicating every cryptographic operation and security guarantee relevant to every message within the handshake pattern.

Noise Explorer can also automatically generate software implementations for arbitrary handshake patterns in Go or in Rust, programming languages that are currently highly popular in server use case scenarios where secure channel protocols are often deployed. Generated Go and Rust implementations are resistant to side channel attacks and are optimized for performance and memory efficiency.

1.3 Contributions

Formal semantics for Noise Handshake Patterns by translation to the applied pi-calculus. §3 presents a formal translation from valid Noise Handshake Patterns into processes in the applied pi-calculus [10]. The resulting model accounts for the cryptographic operations, state machine transitions, message exchanges and top-level processes illustrating protocol execution. This translation serves as a formal semantics of Noise and is implemented within a new tool called Noise Explorer that takes Noise handshake patterns as input and generates a ProVerif script as output. We also formalize the validity rules of Noise as typing rules on the source patterns.

Symbolic security analysis of Noise Handshake Patterns using ProVerif. In §4, shows how we can encode the security goals documented in the Noise Protocol Framework specification as security queries in ProVerif syntax. We handle all five "confidentiality" security goals from the specification and extend the two "authentication" goals to four. Noise Explorer can automatically verify all these goals for ProVerif scripts generated from handshake patterns.

Formal verification results for 57 Noise Handshake Patterns in the Noise Protocol Framework specification. §5 sees all of the previous contributions come together to provide formal verification results for 57 handshake patterns.[1] We find that while most of the results match those predicted by the specification authors, our extended model for "authentication" queries allows for more nuanced results. Furthermore, in §6, we analyze unsafe handshake patterns and discover a potential for forgery attacks should handshake patterns not be followed properly.

Software implementation generation for arbitrary Noise Handshake Patterns. §6 discusses a new feature in Noise Explorer which allows for the automated generation of software implementations for any handshake pattern.

---

[1] Anyone can use Noise Explorer to increase this number by designing, validating and then automatically verifying their own Noise Handshake Pattern.

1.4 Related Work

Our formal analysis of Noise Handshake Patterns occurs in the symbolic model using ProVerif. Other modern secure channel protocols were recently verified using similar methodology [11,12]. Tamarin [13], another protocol verifier in the symbolic model, was used by Donenfeld and Milner [14] to verify the WireGuard [15] VPN protocol, which employs the `IKpsk2` handshake pattern.

Game-based proofs have also been obtained for WireGuard by Dowling and Paterson [16] and by Lipp, Blanchet and Bhargavan [17]. The latter work relies on semi-automated verification using the CryptoVerif [18] prover.

All of these analyses on WireGuard do not constitute a comprehensive evaluation of the Noise Protocol Framework as a whole, since they implicate only a single handshake pattern. However, their results on `IKpsk2` were in line with the findings of our own symbolic analysis. After Noise Explorer's initial release as a software framework, Girol [19] published an independent symbolic analysis of Noise using Tamarin which confirmed some of Noise Explorer's findings.

Regarding the generation of secure protocol implementations, we note in §7.2 that while our generated Go and Rust code achieves significant side channel resistance, we are unable to reason on the code itself for notions of functional correctness or security, but rather depend on its structural relationship to the protocol logic in the ProVerif models, which are generated using the same pipeline. However, work on Project Everest [20] has shown that it is possible to use specialized programming languages, such as F⋆ [21], to build a full formally verified HTTPS stack. It is worth noting that TLS is a significantly more complex protocol than any handshake pattern, so such an approach might also prove useful here in the future.

As for the Noise Protocol Framework itself, it has inspired the creation of the STROBE [22] Protocol Framework, which also allows for building new network protocols as well as new cryptographic primitives. Wong proposes merging Noise and STROBE into a single framework called Disco [23]. Finally, Hall-Andersen, Wong, Sullivan and Chator [24] also propose nQuic, a variant of QUIC-TLS [25] that uses Noise for its key exchange and as the basis of its packet protector with no semantic transport changes.

2 Formal Verification in the Symbolic Model

The main goal of this work is to use the ProVerif automated protocol verifier to obtain answers to our formal verification queries. In this section, we describe the parts of ProVerif that are relevant to our analysis.

ProVerif uses the applied pi-calculus, a language geared towards the description of network protocols, as its input language. It analyzes described protocols under a Dolev-Yao model, which effectively mimics an active network attacker. ProVerif models are comprised of a section in which cryptographic protocol primitives and operations are described as `funs` or `letfuns` and a "top-level process" section in which the execution of the protocol on the network is outlined.

In ProVerif, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). The attacker is an arbitrary ProVerif process running in parallel with the protocol, which can read and write messages on public channels and can manipulate them symbolically. Parallel and unbounded numbers of executions of different parts of the protocol are supported.

In the symbolic model, cryptographic primitives are represented as "perfect black-boxes"; a hash function, for example, is typically modeled as a one-way function that is invulnerable to collisions (although all hash functions do have collisions, even if they are hard to find). Encryption primitives are perfect pseudorandom permutations. Hash functions are perfect one-way maps. Using these primitives, one may build complex primitives such as authenticated encryption with associated data (AEAD). Diffie-Hellman exponentiation is defined in terms of algebraic equations, as we shall see below.

2.1 Verification Context

ProVerif models can be seen as symbolically executing the protocol: a typical run includes a process in which Alice initiates a session with Bob, a process in which Alice initiates a session with Charlie, a process in which Bob acts a responder to Alice and a process in which Bob acts as a responder to Charlie. Charlie is a compromised participant whose entire state is controlled by the attacker. Each process in the top-level process are executed in parallel. The top-level process is executed in an unbounded number of sessions. Within the processes, transport messages are again executed in an unbounded number of sessions in both directions. Fresh key material is provided for each ephemeral generated in each session within the unbounded number of sessions: no ephemeral key reuse occurs between the sessions modeled.

2.2 Cryptographic Primitives

Noise Handshake Patterns make use of cryptographic primitives which in this work we will treat as constructions in the symbolic model. We consider the following cryptographic primitives:

- **$\mathsf{KP}()$:** Generates a new Diffie-Hellman key pair consisting of a private key x and a public key g^x.
- **$\mathsf{DH}(x \leftarrow \mathsf{KP}(), y)$:** Derives a Diffie-Hellman shared secret between the private key within the key pair x and the public key y.
- **$\mathsf{E}(k, n, ad, p)$:** Encrypts and generates an authentication tag for plaintext p using key k and nonce n, optionally extending the authentication tag to cover associated data ad. The output is considered to be Authenticated Encryption with Associated Data (AEAD) [26].
- **$\mathsf{D}(k, n, ad, c)$:** Decrypts and authenticates ciphertext c using key k and nonce n. Associated data ad must also be included if it was defined during the encryption step for authentication to pass on both c and ad.
- **$\mathsf{R}(k)$:** Returns a new key by applying a pseudorandom function on k.
- **$\mathsf{H}(d)$:** A one-way hash function on data d.
- **$\mathsf{HKDF}(ck, ik)$:** A Hash-Based Key Derivation function [27] that takes keys (ck, ik) and outputs a triple of keys. In some instances, the third key output is discarded and not used. The function is similar to the original HKDF definition but with ck acting as the salt and with a zero-length *"info"* variable.

In ProVerif, Diffie-Hellman is implemented as a `letfun` that takes two `key`-type values (representing points on the Curve25519 [28] elliptic curve) along with an `equation` that essentially illustrates the Diffie-Hellman relationship $g^{ab} = g^{ba}$ in the symbolic model.[2] DH and KP (implemented as `generate_keypair`) are then implemented as `letfuns` on top of that construction:[3]

```
fun dhexp(key, key):key.
equation forall a:key, b:key;
dhexp(b, dhexp(a, g)) = dhexp(a, dhexp(b, g)).
```

Encryption is implemented as a function that produces a bitstring (representing the ciphertext) parametrized by a key, nonce, associated data and plaintext. Decryption is a reduction function that produces the correct plaintext only when the appropriate parameters are given, otherwise the process ends:

```
fun encrypt(key, nonce, bitstring, bitstring):bitstring.

fun decrypt(key, nonce, bitstring, bitstring):aead reduc
forall k:key, n:nonce, ad:bitstring, plaintext:bitstring;
decrypt(k, n, ad, encrypt(k, n, ad, plaintext)) = aeadpack(true, ad, plaintext).
```

---

[2] Recall that, in the symbolic model, any arithmetic property such as additivity is not a given and must be modeled specifically.

[3] `keypairpack` and `keypairunpack` are a `fun` and `reduc` pair that allow compressing and decompressing a tuple of `key` values into a `keypair`-type value for easy handling throughout the model. Whenever the suffixes `pack` and `unpack` appear from now on, it is safe to assume that they function in a similar pattern.

Finally, H and HMAC are implemented as one-way functions parametrized by two bitstrings (for ease of use in modeling in the case of H, and for a keyed hash representation in the case of HMAC) while HKDF is constructed on top of them.

2.3 ProVerif Model Components

In the ProVerif model of a Noise Handshake Pattern, there are nine components:

1. **ProVerif parameters.** This includes whether to reconstruct a trace and whether the attacker is active or passive.
2. **Types.** Cryptographic elements, such as keys and nonces, are given types. Noise Handshake Message state elements such as `CipherStates`, `SymmetricStates` and `HandshakeStates` (see §3) are given types as well as constructors and reductors.
3. **Constants.** The generator of the g Diffie-Hellman group, HKDF constants such as `zero` and the names of principals (Alice, indicating the initiator, Bob, indicating the recipient, and Charlie, indicating a compromised principal controlled by the attacker) are all declared as constants.
4. **Message parsing and serialization.** Functions are declared for bitstring concatenation, useful for constructing and destructing the message buffers involved in the Noise Protocol Framework's `WriteMessage` and `ReadMessage` functions.
5. **Cryptographic primitives.** DH, KP, E, D, H and HKDF are modeled as cryptographic primitives in the symbolic model.
6. **State transition functions.** All functions defined for `CipherState`, `SymmetricState` and `HandshakeState` are implemented in the applied pi-calculus.
7. **Channels.** Only a single channel is declared, `pub`, representing the public Internet.
8. **Events and queries.** Here, the protocol events and security queries relevant to a particular handshake pattern are defined. This includes the four authentication queries and five confidentiality queries discussed in §4.
9. **Protocol processes and top-level process.** This includes the `WriteMessage` and `ReadMessage` function for each handshake and transport message, followed by the top-level process illustrating the live execution of the protocol on the network.

3 Representing Noise in the Applied Pi-Calculus

The Noise Protocol Framework is restricted only to describing messages between two parties (initiator and responder), the public keys communicated and any Diffie-Hellman operations conducted. Messages are called Noise "Message Patterns". They make up authenticated key exchanges, which are called Noise "Handshake Patterns". Noise supports authenticated encryption with associated data (AEAD) and Diffie-Hellman key agreement. Noise does not currently support any signing operations.

The full description of a Noise-based secure channel protocol is contained within its description of a handshake pattern, such as the one seen in Fig. 1. The initial messages within a handshake pattern, which contain *tokens* representing public keys or Diffie-Hellman operations are called *handshake messages*. After handshake messages, *transport messages* may occur carrying encrypted payloads. Here is an overview of the tokens that may appear in a handshake message:

- e, s**.** The sender is communicating their ephemeral or static public key, respectively.
- ee, es, se, ss**.** The sender has locally calculated a new shared secret. The first letter of the token indicates the initiator's key share while the second indicates the responder's key share. As such, this token remains the same irrespective of who is sending the particular handshake message in which it occurs.

Syntax

$k ::=$	public DH keys
e	ephemeral DH key
s	static DH key
$t ::=$	tokens
k	public DH key
$k_1 k_2$	shared DH secret (ee, es, se, or ss)
psk	pre-shared key
$p ::=$	pre-messages
ϵ	end of pre-message (empty)
k, p	pre-message with public DH key
$m ::=$	messages
ϵ	end of message (empty)
t, m	message with token
$h_r ::=$	handshake (responder's turn)
ϵ	end of handshake
$\xleftarrow{m}\ h_i$	responder message, then initiator
$h_i ::=$	handshake (initator's turn)
ϵ	end of handshake
$\xrightarrow{m}\ h_r$	initiator message, then responder
$n ::=$	noise patterns
$\xrightarrow{p_1}\xleftarrow{p_2}\ h_i$	pre-messages, then handshake

Figure 3: Noise Handshake Pattern Syntax.

- *psk*. The sender is mixing a pre-shared key into their local state and the recipient is assumed to do the same.

Optionally, certain key materials can be communicated before a protocol session is initiated. A practical example of how this is useful could be secure messaging protocols, where prior knowledge of an ephemeral key pair could help a party initiate a session using a zero-round-trip protocol, which allows them to send an encrypted payload without the responder needing to be online.

These *pre-message patterns* are represented by a series of messages occuring before handshake messages. The end of the pre-message stage is indicated by a "…" sign. For example, in Fig. 1, we see a pre-message pattern indicating that the initiator has prior knowledge of the responder's public static key before initiating a protocol session.

3.1 Validating Noise Handshake Pattern Syntax

Noise Handshake Patterns come with certain validity rules:

- **Alternating message directions.** Message direction within a Noise Handshake Pattern must alternate (initiator → responder, initiator ← responder), with the first message being sent by the initiator.

- **Performing Diffie-Hellman key agreement more than once.** Principals must not perform the same Diffie-Hellman key agreement more than once per handshake.

- **Sending keys more than once.** Principals must not send their static public key or ephemeral public key more than once per handshake.

- **Transport messages after handshake messages.** Noise Handshake Patterns can only contain transport handshake messages at the very bottom of the pattern.

Validity Rules

$d ::= \leftarrow \mid \rightarrow$ — *direction*: left or right

$\bar{t} ::= k^{d} \mid k_1k_2 \mid psk$ — tokens w. DH keys

$\Gamma ::= \{\bar{t}_0, \dots, \bar{t}_n\}$ — *context*: set of prior tokens

$$\mathit{tokens}^{d}(m) \triangleq \{k^{d} \mid k \in m \cap \{e, s\}\} \cup (m \setminus \{e, s\})$$

$\boxed{\text{Pre-Message Validity: } \Gamma \vdash^{d} p}$

$$\text{PreEmpty}\;\frac{}{\Gamma \vdash^{d} \epsilon}$$

$$\text{PreKey}\;\frac{k^{d} \notin \Gamma \quad \Gamma \cup \{k^{d}\} \vdash^{d} p}{\Gamma \vdash^{d} k, p}$$

$\boxed{\text{Message Validity: } \Gamma \vdash^{d} m}$

$$\text{MsgEmpty}^{\rightarrow}\;\frac{\begin{array}{c} ss \in \Gamma \Rightarrow se \in \Gamma \quad se \in \Gamma \Rightarrow ee \in \Gamma \\ psk \in \Gamma \Rightarrow e^{\rightarrow} \in \Gamma \end{array}}{\Gamma \vdash^{\rightarrow} \epsilon}$$

$$\text{MsgEmpty}^{\leftarrow}\;\frac{\begin{array}{c} ss \in \Gamma \Rightarrow es \in \Gamma \quad es \in \Gamma \Rightarrow ee \in \Gamma \\ psk \in \Gamma \Rightarrow e^{\leftarrow} \in \Gamma \end{array}}{\Gamma \vdash^{\leftarrow} \epsilon}$$

$$\text{MsgKey}\;\frac{k^{d} \notin \Gamma \quad \Gamma \cup \{k^{d}\} \vdash^{d} m}{\Gamma \vdash^{d} k, m}$$

$$\text{MsgDH}\;\frac{\begin{array}{c} k_1^{\rightarrow} \in \Gamma \quad k_2^{\leftarrow} \in \Gamma \quad k_1k_2 \notin \Gamma \\ \Gamma \cup \{k_1k_2\} \vdash^{d} m \end{array}}{\Gamma \vdash^{d} k_1k_2, m}$$

$$\text{MsgPSK}\;\frac{psk \notin \Gamma \quad \Gamma \cup \{psk\} \vdash^{d} m}{\Gamma \vdash^{d} psk, m}$$

$\boxed{\text{Handshake Validity: } \Gamma \vdash h_i}$

$$\text{HSEmpty}\;\frac{}{\Gamma \vdash \epsilon}$$

$$\text{HSMessageI}\;\frac{\Gamma \vdash^{\rightarrow} m \quad \Gamma \cup \mathit{tokens}^{\rightarrow}(m) \vdash h_r}{\Gamma \vdash \xrightarrow{m} h_r}$$

$$\text{HSMessageR}\;\frac{\Gamma \vdash^{\leftarrow} m \quad \Gamma \cup \mathit{tokens}^{\leftarrow}(m) \vdash h_i}{\Gamma \vdash \xleftarrow{m} h_i}$$

$\boxed{\text{Noise Pattern Validity: } \vdash n}$

$$\text{NoiseValid}\;\frac{\begin{array}{c} \{\} \vdash^{\rightarrow} p_1 \quad \{\} \vdash^{\leftarrow} p_2 \\ \mathit{tokens}^{\rightarrow}(p_1) \cup \mathit{tokens}^{\leftarrow}(p_2) \vdash h_i \end{array}}{\vdash \xrightarrow{p_1} \xleftarrow{p_2} h_i}$$

Figure 4: Noise Pattern Validity Rules

- **Appropriate key share communication.** Principals cannot perform a Diffie-Hellman operation with a key share that was not communicated to them prior.
- **Unused key shares.** Noise Handshake Patterns should not contain key shares that are not subsequently used in any Diffie-Hellman operation.
- **Transport messages.** Noise Handshake Patterns cannot consist purely of transport messages.

The Noise Handshake Pattern syntax is more formally described in Fig. 3, while validity rules are formalized in Fig. 4.

3.2 Local State

Each principal in a Noise protocol handshake keeps three local state elements: `CipherState`, `SymmetricState` and `HandshakeState`. These states contain each other in a fashion similar to a Russian Matryoshka doll, with `HandshakeState` being the largest element, containing `SymmetricState` which in turn contains `CipherState`.

- **`CipherState`** contains k (a symmetric key) and n (a nonce), used to encrypt and decrypt ciphertexts.
- **`SymmetricState`** contains a `CipherState` tuple (k, n), an additional key ck and a hash function output h.
- **`HandshakeState`** contains a `SymmetricState` along with additional local public keys (s, e) and remote public keys (rs, re).

Each state element comes with its own set of state transformation functions. These functions are triggered by the occurence and position of tokens within a handshake pattern. We present a description of the state transition functions as seen in the Noise Protocol Framework specification, but restricted to a representation that follows implementing handshake patterns in the symbolic model.

3.2.1 CipherState

A `CipherState` comes with the following state transition functions:

- **`InitializeKey(key)`:** Sets $k = \texttt{key}$. Sets $n = 0$.
- **`HasKey()`:** Returns true if k is non-empty, false otherwise.
- **`SetNonce(nonce)`:** Sets $n = \texttt{nonce}$.
- **`EncryptWithAd(ad, p)`:** If k is non-empty returns $\mathsf{E}(k, n, ad, p)$ then increments n. Otherwise returns p.
- **`DecryptWithAd(ad, c)`:** If k is non-empty returns $\mathsf{D}(k, n, ad, c)$ then increments n. Otherwise returns c. n is not incremented if authenticated decryption fails.
- **`Rekey()`:** Sets $k = \mathsf{R}(k)$.

In ProVerif, `InitializeKey` simply returns a `cipherstate`-type value packed with the input key and a starting nonce. `hasKey` unpacks an input `cipherstate` and checks whether the key is defined. The rest of the functions are based on similarly evident constructions:

```
letfun encryptWithAd(cs:cipherstate, ad:bitstring, plaintext:bitstring) =
      let (k:key, n:nonce) = cipherstateunpack(cs) in
      let e = encrypt(k, n, ad, plaintext) in
      let csi = setNonce(cs, increment_nonce(n)) in
      (csi, e).

letfun decryptWithAd(cs:cipherstate, ad:bitstring, ciphertext:bitstring) =
      let (k:key, n:nonce) = cipherstateunpack(cs) in
      let d = decrypt(k, n, ad, ciphertext) in
      let (valid:bool, adi:bitstring, plaintext:bitstring) = aeadunpack(d) in
      let csi = setNonce(cs, increment_nonce(n)) in
      (csi, plaintext, valid).

letfun reKey(cs:cipherstate) =
      let (k:key, n:nonce) = cipherstateunpack(cs) in
      let ki = encrypt(k, maxnonce, empty, zero) in
      cipherstatepack(bit2key(ki), n).
```

3.2.2 SymmetricState

A `SymmetricState` comes with the following state transition functions:

- **`InitializeSymmetric(name)`:** Sets $ck = h = \mathsf{H}(\texttt{name})$.
- **`MixKey(ik)`:** Sets $(ck, tk) = \mathsf{HKDF}(ck, \texttt{ik})$ and calls $\texttt{InitializeKey}(tk)$.
- **`MixHash(data)`:** Sets $h = \mathsf{H}(h \parallel \texttt{data})$.[4]
- **`MixKeyAndHash(ik)`:** Sets $(ck, th, tk) = \mathsf{HKDF}(ck, \texttt{ik})$, then calls $\texttt{MixHash}(th)$ and $\texttt{InitializeKey}(tk)$.
- **`GetHandshakeHash()`:** Returns h.
- **`EncryptAndHash(p)`:** Sets $c = \texttt{EncryptWithAd}(h, \texttt{p})$. Calls $\texttt{MixHash}(c)$ and returns c.

---

[4] $\parallel$ denotes bitstring concatenation.

- **`DecryptAndHash(c)`:** Sets $p = \texttt{DecryptWithAd}(h, \texttt{c})$. Calls $\texttt{MixHash}(c)$ and returns c and returns p.

- **`Split()`:** Sets $(tk_1, tk_2) = \textsf{HKDF}(ck, \texttt{zero})$. Creates two `CipherStates` (c_1, c_2). Calls c_1`.InitializeKey`(tk_1) and c_2`.InitializeKey`(tk_2). Returns (c_1, c_2), a pair of `CipherStates` for encrypting transport messages.[5]

In ProVerif, these functions are implemented based on `letfun` declarations that combine previously declared `funs` and `letfuns`:

```
      letfun initializeSymmetric(protocol_name:bitstring) =
      let h = hash(protocol_name, empty) in
      let ck = bit2key(h) in
      let cs = initializeKey(bit2key(empty)) in
      symmetricstatepack(cs, ck, h).

letfun mixKey(ss:symmetricstate, input_key_material:key) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (ck:key, temp_k:key, output_3:key) = hkdf(ck, input_key_material) in
      symmetricstatepack(initializeKey(temp_k), ck, h).

letfun mixHash(ss:symmetricstate, data:bitstring) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      symmetricstatepack(cs, ck, hash(h, data)).

letfun mixKeyAndHash(ss:symmetricstate, input_key_material:key) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (ck:key, temp_h:key, temp_k:key) = hkdf(ck, input_key_material) in
      let (cs:cipherstate, temp_ck:key, h:bitstring) = symmetricstateunpack(mixHash(
          symmetricstatepack(cs, ck, h), key2bit(temp_h))) in
      symmetricstatepack(initializeKey(temp_k), ck, h).

letfun getHandshakeHash(ss:symmetricstate) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      (ss, h).

letfun encryptAndHash(ss:symmetricstate, plaintext:bitstring) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (cs:cipherstate, ciphertext:bitstring) = encryptWithAd(cs, h, plaintext) in
      let ss = mixHash(symmetricstatepack(cs, ck, h), ciphertext) in
      (ss, ciphertext).

letfun decryptAndHash(ss:symmetricstate, ciphertext:bitstring) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (cs:cipherstate, plaintext:bitstring, valid:bool) = decryptWithAd(cs, h, ciphertext
          ) in
      let ss = mixHash(symmetricstatepack(cs, ck, h), ciphertext) in
      (ss, plaintext, valid).

letfun split(ss:symmetricstate) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (temp_k1:key, temp_k2:key, temp_k3:key) = hkdf(ck, bit2key(zero)) in
      let cs1 = initializeKey(temp_k1) in
      let cs2 = initializeKey(temp_k2) in
      (ss, cs1, cs2).
```

3.2.3 HandshakeState

A `HandshakeState` comes with the following state transition functions:

- **`Initialize(hp, i, s, e, rs, re)`:** hp denotes a valid Noise Handshake Pattern. i is a boolean which denotes whether the local state belongs to the initiator. Static public keys (s, rs)

---

[5] `zero` is meant to denote a null bitstring.

may be left empty or may be pre-initialized in the event that any of them appeared in a pre-message. Calls `InitializeSymmetric(hp.name)`. Calls `MixHash()` once for each public key listed in the pre-messages within `hp`.

- **`WriteMessage(p)`:** Depending on the tokens present in the current handshake message, different operations occur:
  - e: Sets $e \leftarrow \mathsf{KP}()$. Appends g^e to the return buffer. Calls `MixHash`(g^e).
  - s: Appends `EncryptAndHash`(g^s) to the buffer.
  - ee: Calls `MixKey`($\mathsf{DH}(e, re)$).
  - es: Calls `MixKey`($\mathsf{DH}(e, rs)$) if initiator, `MixKey`($\mathsf{DH}(s, re)$) if responder.
  - se: Calls `MixKey`($\mathsf{DH}(s, re)$) if initiator, `MixKey`($\mathsf{DH}(e, rs)$) if responder.
  - ss: Calls `MixKey`($\mathsf{DH}(s, rs)$).

  Then, `EncryptAndHash(p)` is appended to the return buffer. If there are no more handshake messages, two new `CipherStates` are returned by calling `Split()`.

- **`ReadMessage(m)`:** Depending on the tokens present in the current handshake message, different operations occur:
  - e: Sets `re` to the public ephemeral key retrieved from `m`.
  - s: Sets `temp` to the encrypted public static key retrieved from `m`. Sets `rs` to the result of `DecryptAndHash(temp)`, failing on authenticated decryption error.
  - ee: Calls `MixKey`($\mathsf{DH}(e, re)$).
  - es: Calls `MixKey`($\mathsf{DH}(e, rs)$) if initiator, `MixKey`($\mathsf{DH}(s, re)$) if responder.
  - se: Calls `MixKey`($\mathsf{DH}(s, re)$) if initiator, `MixKey`($\mathsf{DH}(e, rs)$) if responder.
  - ss: Calls `MixKey`($\mathsf{DH}(s, rs)$).

  Then, `DecryptAndHash` is called on the message payload extracted from m. If there are no more handshake messages, two new `CipherStates` are returned by calling `Split()`.

3.3 Dynamically Generating ReadMessage and WriteMessage Functions in the Applied Pi-Calculus

In Noise Explorer (our analysis framework for Noise Handshake Patterns), cryptographic primitives and state transition functions are included from a pre-existing set of Noise ProVerif headers written as a part of this work and are not automatically generated according to a set of rules. Events, queries, protocol processes and the top-level process, however, are fully generated using translation rules that make them unique for each handshake pattern.

In our generated ProVerif models, each handshake message and transport message is given its own `WriteMessage` and `ReadMessage` construction represented as `letfun`s. These functions are constructed to invoke the appropriate state transition functions depending on the tokens included in the message pattern being translated. An example generated translation can be see in Fig. 5, which concerns the first message in `IK` (Fig. 1): $\rightarrow e, es, s, ss$.

The state transition rules described in the Noise specification are implicated by the tokens within the message pattern. By following these rules, Noise Explorer generates a symbolic model that implements the state transitions relevant to this particular message pattern. From the initiator's side:

- e: Signals that the initiator is sending a fresh ephemeral key share as part of this message. This token adds one state transformation to `writeMessage_a`: `mixHash`, which hashes the new key into the session hash.

```
letfun writeMessage_a(me:principal, them:principal, hs:handshakestate, payload:bitstring, sid:sessionid) =
let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key, initiator:bool) = handshakestateunpack(hs) in
let (ne:bitstring, ciphertext1:bitstring, ciphertext2:bitstring) = (empty, empty, empty) in
let e = generate_keypair(key_e(me, them, sid)) in
let ne = key2bit(getpublickey(e)) in
let ss = mixHash(ss, ne) in
(* No PSK, so skipping mixKey *)
let ss = mixKey(ss, dh(e, rs)) in
let s = generate_keypair(key_s(me)) in
let (ss:symmetricstate, ciphertext1:bitstring) = encryptAndHash(ss, key2bit(getpublickey(s))) in
let ss = mixKey(ss, dh(s, rs)) in
let (ss:symmetricstate, ciphertext2:bitstring) = encryptAndHash(ss, payload) in
let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
let message_buffer = concat3(ne, ciphertext1, ciphertext2) in
(hs, message_buffer).
```

```
letfun readMessage_a(me:principal, them:principal, hs:handshakestate, message:bitstring, sid:sessionid) =
let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key, initiator:bool) = handshakestateunpack(hs) in
let (ne:bitstring, ciphertext1:bitstring, ciphertext2:bitstring) = deconcat3(message) in
let valid1 = true in
let re = bit2key(ne) in
let ss = mixHash(ss, key2bit(re)) in
(* No PSK, so skipping mixKey *)
let ss = mixKey(ss, dh(s, re)) in
let (ss:symmetricstate, plaintext1:bitstring, valid1:bool) = decryptAndHash(ss, ciphertext1) in
let rs = bit2key(plaintext1) in
let ss = mixKey(ss, dh(s, rs)) in
let (ss:symmetricstate, plaintext2:bitstring, valid2:bool) = decryptAndHash(ss, ciphertext2) in
if ((valid1 && valid2) && (rs = getpublickey(generate_keypair(key_s(them))))) then (
  let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
  (hs, plaintext2, true)).
```

Figure 5: The `WriteMessage` and `ReadMessage` `letfun` constructions for the first message in `IK` (Fig. 1), generated according to translation rules from Noise Handshake Pattern to ProVerif. The appropriate state transition functions are invoked in accordance with the occurence and ordering of tokens in the message pattern.

- *es*: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds one state transformation to `writeMessage_a`: `mixKey`, which calls the HKDF using as input the existing `SymmetricState` key and $\mathsf{DH}(e, rs)$, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.

- *s*: Signals that the initiator is sending a static key share as part of this message. This token adds one state transformation to `writeMessage_a`: `encryptAndHash` is called on the static public key. If any prior Diffie-Hellman shared secret was established between the sender and the recipient, this allows the initiator to communicate their long-term identity with some degree of confidentiality.

- *ss*: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's static key as part of this message. This token adds one state transformation to `writeMessage_a`: `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and $\mathsf{DH}(s, rs)$, the Diffie-Hellman share calculated from the initiator's static key and the responder's static key.

Message A's payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes `encryptAndHash`, which performs AEAD encryption on the payload, with the session hash as the associated data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

On the receiver end:

- *e*: Signals that the responder is receiving a fresh ephemeral key share as part of this message. This token adds one state transformation to `readMessage_a`: `mixHash`, which hashes the new key into the session hash.

- *es*: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds one state transformation to `readMessage_a`: `mixKey`, which calls the HKDF function using, as input, the existing SymmetricState key, and $\mathsf{DH}(s, re)$, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.

- s: Signals that the responder is receiving a static key share as part of this message. This token adds one state transformation to `readMessage_a`: `decryptAndHash` is called on the static public key. If any prior Diffie-Hellman shared secret was established between the sender and the recipient, this allows the initiator to communicate their long-term identity with some degree of confidentiality.

- ss: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's static key as part of this message. This token adds one state transformation to `readMessage_a`: `mixKey`, which calls HKDF function using, as input, the existing `SymmetricState` key and $\mathsf{DH}(s, rs)$, the Diffie-Hellman share calculated from the initiator's static key and the responder's static key.

Message A's payload invokes the following operation: `decryptAndHash`, which performs AEAD decryption on the payload, with the session hash as the associated data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

3.4 Other Specification Features

The Noise specification defines 15 "fundamental patterns", 23 "deferred patterns" and 21 "PSK patterns". `IK` (Fig. 1) and `IN` (Fig. 2) are two fundamental patterns. Deferred patterns are essentially modified fundamental patterns where the communication of public keys or the occurence of Diffie-Hellman operations is intentionally delayed. PSK pattterns are patterns in which a pre-shared key token appears. Fig. 6 illustrates a deferred pattern based on the fundamental pattern shown in Fig. 1.

The full Noise specification extends somewhat beyond the description given as part of this work, including features such as "identity hiding" and "dummy keys." Some of these features are potentially valuable and slated as future work.

$$\begin{aligned}
&I1K: \\
&\qquad \leftarrow s \\
&\qquad \cdots \\
&\qquad \rightarrow e, es, s \\
&\qquad \leftarrow e, ee \\
&\qquad \rightarrow se
\end{aligned}$$

Figure 6: An example Noise Handshake Pattern, `I1K`. This is a deferred pattern based on `IK`, shown in Fig. 1.

4 Modeling Noise Security Goals in the Symbolic Model

Since our goal is to evaluate the security guarantees achieved by arbitrary Noise Handshake Patterns, it is crucial to have a set of well-defined security goals on which to base our analysis. We want to formulate these "security grades" in ProVerif as event-based queries. This implies specifying a number of events triggered at specific points in the protocol flow as well as queries predicated on these events.

A set of the queries for the security goals described in this section is generated for each handshake and transport message within a handshake pattern, allowing for verification to occur in the comprehensive context described in §2.

The Noise specification defines different handshake patterns to suit different scenarios. These patterns come with different security properties depending on which keys and shared secrets are employed and when. Two types of security grades are defined: *"authentication"* grades dealing with the authentication of a message to a particular sender (and optionally, receiver) and *"confidentiality"* grades dealing with a message's ability to resist the obtention of plaintext by an unauthorized party.

For example, the handshake pattern illustrated in Fig. 1 is described in the original specification as claiming to reach strong security goals: handshake and transport message are attributed authentication grades of 1, 2, 2 and 2 respectively, and confidentiality grades of 2, 4, 5 and 5. Other handshake patterns, such as the one described in Fig. 2, sacrifice security properties to deal away with the need to share public keys beforehand or to conduct additional key derivation steps (authentication: 0, 0, 2, 0 and confidentiality: 0, 3, 1 5.) We describe each of these authentication and confidentiality grades in fuller detail below.

In our analysis, we leave the confidentiality grades intact. However, we introduce two new additional security grades, 3 and 4, which provide more nuance for the existing authentication grades 1 and 2. In our analysis, authentication grades 1 and 2 hold even if the authentication of the message can be forged towards the recipient if the sender carries out a separate session with a separate, compromised recipient. Authentication grades 3 and 4 do not hold in this case. This nuance does not exist in the authentication grades defined in the latest Noise specification.

In all examples below, Bob is the sender and Alice is the recipient. The message in question is message D, i.e. the fourth message pattern within the handshake pattern. In the event of a non-existent PSK, the `LeakPsk` event is removed from the query. A principal c refers to any arbitrary principal on the network, which includes compromised principal Charlie.

4.1 Events

The following events appear in generated ProVerif models:

- **`SendMsg(principal, principal, stage, bitstring)`** takes in the identifier of the message sender, the identifier of the recipient, a "stage" value and the plaintext of the message payload. The "stage" value is the output of a function parametrized by the session ID, a unique value generated for each execution of the protocol using ProVerif's `new` keyword, and an identifier of which message this is within the handshake pattern (first message, second message, etc.)

- **`RecvMsg(principal, principal, stage, bitstring)`** is a mirror event of the above, with the first principal referring to the recipient and the second referring to the sender.

- **`LeakS(phasen, principal)`** indicates the leakage of the long-term secret key of the principal. `phasen` refers to which "phase" the leak occured: in generated ProVerif models, phase 0 encompasses protocol executions that occur while the session is under way, while phase 1 is strictly limited to events that occur after the session has completed and has been closed.

- **`LeakPsk(phasen, principal, principal)`** indicates the leakage of the pre-shared key (PSK) of the session between an initiator (specified as the first principal) and a responder in the specified phase.

4.2 Authentication Grades

Grade 0 indicates no authentication: the payload may have been sent by any party, including an active attacker.

4.2.1 Sender authentication

In this query, we test for sender authentication and message integrity. If Alice receives a valid message from Bob, then Bob must have sent that message to someone, or Bob had their static key compromised before the session began, or Alice had their static key compromised before the session began:

$$\begin{aligned}
&\mathit{RecvMsg}(\mathit{alice}, \mathit{bob}, \mathit{stage}(d, \mathit{sid}), m) \longrightarrow\\
&\mathit{SendMsg}(\mathit{bob}, c, \mathit{stage}(d, \mathit{sid}), m) \vee\\
&(\mathit{LeakS}(\mathit{phase}_0, \mathit{bob}) \wedge \mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob})) \vee\\
&(\mathit{LeakS}(\mathit{phase}_0, \mathit{alice}) \wedge \mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob}))
\end{aligned}$$

4.2.2 Sender authentication and key compromise impersonation resistance

In this query, we test for sender authentication and Key Compromise Impersonation resistance. If Alice receives a valid message from Bob, then Bob must have sent that message to someone, or Bob had their static key compromised before the session began.

$$\begin{aligned}&\mathit{RecvMsg}(\mathit{alice}, \mathit{bob}, \mathit{stage}(d, \mathit{sid}), m) \longrightarrow\\&(\mathit{bob}, c, \mathit{stage}(d, \mathit{sid}), m) \vee\\&\mathit{LeakS}(\mathit{phase}_0, \mathit{bob})\end{aligned}$$

4.2.3 Sender and received authentication and message integrity

If Alice receives a valid message from Bob, then Bob must have sent that message to Alice specifically, or Bob had their static key compromised before the session began, or Alice had their static key compromised before the session began. This query is not present in the original Noise specification and is contributed by this work.

$$\begin{aligned}&\mathit{RecvMsg}(\mathit{alice}, \mathit{bob}, \mathit{stage}(d, \mathit{sid}), m) \longrightarrow\\&\mathit{SendMsg}(\mathit{bob}, \mathit{alice}, \mathit{stage}(d, \mathit{sid}), m) \vee\\&(\mathit{LeakS}(\mathit{phase}_0, \mathit{bob}) \wedge \mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob})) \vee\\&(\mathit{LeakS}(\mathit{phase}_0, \mathit{alice}) \wedge \mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob}))\end{aligned}$$

4.2.4 Sender and receiver authentication and key compromise impersonation resistance

If Alice receives a valid message from Bob, then Bob must have sent that message to Alice specifically, or Bob had their static key compromised before the session began. This query is not present in the original Noise specification and is contributed by this work.

$$\begin{aligned}&\mathit{RecvMsg}(\mathit{alice}, \mathit{bob}, \mathit{stage}(d, \mathit{sid}), m) \longrightarrow\\&\mathit{SendMsg}(\mathit{bob}, \mathit{alice}, \mathit{stage}(d, \mathit{sid}), m) \vee\\&\mathit{LeakS}(\mathit{phase}_0, \mathit{bob})\end{aligned}$$

4.3 Confidentiality Grades

Grade 0 indicates no confidentiality: the payload is sent in cleartext.

4.3.1 Encryption to an ephemeral recipient

In these queries, we test for message secrecy by checking if a passive attacker or active attacker is able to retrieve the payload plaintext only by compromising Alice's static key either before or after the protocol session. Passing this query under a passive attacker achieves confidentiality grade 1, while doing so under an active attacker achieves confidentiality grade 2 (encryption to a known recipient, forward secrecy for sender compromise only, vulnerable to replay.)

$$\begin{aligned}&\mathit{attacker}_{p1}(\mathit{msg}_d(\mathit{bob}, \mathit{alice}, \mathit{sid})) \longrightarrow\\&(\mathit{LeakS}(\mathit{phase}_0, \mathit{alice}) \ \vee\ \mathit{LeakS}(\mathit{phase}_1, \mathit{alice})) \wedge\\&(\mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob}) \vee\\&\mathit{LeakPsk}(\mathit{phase}_1, \mathit{alice}, \mathit{bob}))\end{aligned}$$

In the above, $\mathit{attacker}_{p1}$ indicates that the attacker obtains the message in phase 1 of the protocol execution.

4.3.2 Encryption to a known recipient, weak forward secrecy

In this query, we test for forward secrecy by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Alice's static key before the protocol session, or after the protocol session along with Bob's static public key (at any time.) Passing this query under a passive attacker achieves confidentiality grade 3, while doing so under an active attacker achieves confidentiality grade 4 (encryption to a known recipient, weak forward secrecy only if the sender's private key has been compromised.)

$$\begin{aligned}&\mathit{attacker}_{p1}(\mathit{msg}_d(\mathit{bob}, \mathit{alice}, \mathit{sid})) \longrightarrow\\&(\mathit{LeakS}(\mathit{phase}_0, \mathit{alice}) \ \wedge\ \mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob})) \vee\\&(\mathit{LeakS}(p_x, \mathit{alice}) \ \wedge\ \mathit{LeakPsk}(p_y, \mathit{alice}, \mathit{bob}) \ \wedge\\&\mathit{LeakS}(p_z, \mathit{bob}))\end{aligned}$$

In the above, p_x, p_y, p_z refer to any arbitrary phases.

4.3.3 Encryption to a known recipient, strong forward secrecy

In this query, we test for strong forward secrecy by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key before the protocol session. Passing this query achieves confidentiality grade 5.

$$\begin{aligned}&\mathit{attacker}_{p1}(\mathit{msg}_d(\mathit{bob}, \mathit{alice}, \mathit{sid})) \longrightarrow\\&(\mathit{LeakS}(\mathit{phase}_0, \mathit{alice}) \wedge \mathit{LeakPsk}(\mathit{phase}_0, \mathit{alice}, \mathit{bob}))\end{aligned}$$

4.4 Limitations on Modeling Security Grades

Our analysis of authentication grades comes with an important limitation: When Noise Explorer generates the authentication queries below, it uses two different values, sid_a and sid_b, to refer to the session ID as registered in the trigger events by Alice and Bob. This differs from, and is in fact less accurate than the queries described previously, which use the same session ID, *sid*, for both Alice and Bob. We are forced to adopt this approach due to performance limitations in our models during verification should we choose to use a single *sid* value for both Alice and Bob. However, we argue that since processes with differing *sid* values cause decryption operations that use shared secrets derived from ephemeral keys to fail, and therefore for those processes to halt, we still obtain essentially the same verification scenarios that these queries target.

Additionally, with regards to our confidentiality grades, whenever a pattern contains a PSK and `LeakPSK` events start to get involved, we ideally account for cases where one long-term secret is compromised but not the other. This indicates that we may need a richer notion of authenticity and confidentiality grades than the 1-5 markers that the Noise specification provides. For consistency, we are still using the old grades, but to truly understand and differentiate the security provided in many cases, we recommend that the user view the detailed queries and results as generated by Noise Explorer and available in its detailed rendering of the verification results.

5 Verifying Arbitrary Noise Handshake Patterns with Noise Explorer

A central motivation to this work is the obtention of a general framework for designing, reasoning about, formally verifying, implementing and comparing any arbitrary Noise Handshake Pattern. Noise Explorer is a web framework that implements all of the formalisms and ProVerif translation logic described so far in this work in order to provide these features.

Noise Explorer is ready for use by the general public today at `https://noiseexplorer.com`. Here are Noise Explorer's main functionalities:

Pattern	Auth.	Conf.	Pattern	Auth.	Conf.	Pattern	Auth.	Conf.
N	0	2	XK1	0 2 4 4 4	0 1 5 5 5	Npsk0	0	2
K	1	2	X1K1	0 2 0 4 4 4	0 1 5 3 5 5	Kpsk0	1	2
X	1	2	X1X	0 2 0 4 4 4	0 1 5 3 5 5	Xpsk1	1	2
NN	0 0 0 0	0 1 1 1	XX1	0 0 4 4 4	0 1 3 5 5	NNpsk0	0 0 0 0	0 1 1 1
NK	0 2 0 2	2 1 5 1	X1X1	0 0 0 4 4 4	0 1 3 3 5 5	NNpsk2	0 0 0 0	0 1 1 1
NX	0 2 0 2	0 1 5 1	K1N	0 0 2 0 2	0 1 1 5 1	NKpsk0	0 4 0 4	2 1 5 1
XN	0 0 2 0 2	0 1 1 5 1	K1K	0 4 4 4 4	2 1 5 5 5	NKpsk2	0 4 0 4	2 1 5 1
XK	0 2 4 4 4	2 1 5 5 5	KK1	0 4 4 4	0 3 5 5	NXpsk2	0 4 0 4	0 1 5 1
XX	0 2 4 4 4	0 1 5 5 5	K1K1	0 4 4 4	0 1 5 5 5	XNpsk3	0 0 4 0 4	0 1 1 5 1
KN	0 0 2 0	0 3 1 5	K1X	0 4 4 4 4	0 1 5 5 5	XKpsk3	0 2 4 4 4	2 1 3 5 5
KK	1 4 4 4	2 4 5 5	KX1	0 0 4 4 4	0 3 3 5 5	KNpsk0	0 0 4 0	0 3 1 5
KX	0 4 4 4	0 3 5 5	K1X1	0 0 4 4 4	0 1 3 5 5	KNpsk2	0 0 4 0	0 3 1 5
IN	0 0 2 0	0 3 1 5	I1N	0 0 2 0 2	0 1 1 5 1	KKpsk0	1 4 4 4	2 4 5 5
IK	1 4 4 4	2 4 5 5	I1K	0 4 4 4 4	2 1 5 5 5	KKpsk2	1 4 4 4	2 3 5 5
IX	0 4 4 4	0 3 5 5	IK1	0 4 4 4	0 3 5 5	KXpsk2	0 4 4 4	0 3 5 5
NK1	0 2 0 2	0 1 5 1	I1K1	0 4 4 4 4	0 1 5 5 5	INpsk1	0 0 4 0	0 3 1 5
NX1	0 0 0 2 0	0 1 3 1 5	I1X	0 4 4 4 4	0 1 5 5 5	INpsk2	0 0 4 0	0 3 1 5
X1N	0 0 0 0 2 0	0 1 1 3 1 5	IX1	0 0 4 4 4	0 3 3 5 5	IKpsk2	1 4 4 4	2 3 5 5
X1K	0 2 0 4 4 4	2 1 5 3 5 5	I1X1	0 0 4 4 4	0 1 3 5 5	XXpsk3	0 0 4 4 4	0 1 3 5 5

Figure 7: Verification results for 57 Noise Handshake Patterns. The numbers in the Authentication and Confidentiality column represent the obtained score for each message, starting from the first message in the Handshake Pattern concerned.

Designing and validating Noise Handshake Patterns. This allows protocol designers to immediately obtain validity checks that verify if the protocol conforms to the latest Noise specification.[6]

Generating cryptographic models for formal verification using automated verification tools. Noise Explorer can compile any Noise Handshake Pattern to a full representation in the applied pi-calculus including cryptographic primitives, state machine transitions, message passing and a top-level process illustrating live protocol execution. Using ProVerif, we can then test against sophisticated security queries starting at basic confidentiality and authentication and extending towards forward secrecy and resistance to key compromise impersonation.

Exploring the first compendium of formal verification results for Noise Handshake Patterns. Since formal verification for complex Noise Handshake Patterns can take time and require fast CPU hardware, Noise Explorer comes with a compendium detailing the full results of almost all handshake patterns described in the latest revision of the original Noise specification. These results are presented with a security model that is more comprehensive than the original specification, as described in §4.

5.1 Accessible High Assurance Verification for Noise-Based Protocols

Noise Explorer users are free to specify any arbitrary Noise Handshake Pattern of their own design. Once this input is validated, formal verification models are generated. The ProVerif verification output can then be fed right back into Noise Explorer, which will then generate detailed interactive pages describing the analysis results.

The initial view of the results includes a pedagogical plain-English paragraph for each message summarizing its achieved security goals. For example, the following paragraph is generated for message D (i.e. the fourth message pattern) of `IK`:

"Message D, sent by the responder, benefits from ***sender and receiver authentication*** *and is* ***resistant to Key Compromise Impersonation****. Assuming the corresponding private keys are secure, this*

---

[6] As of writing, Revision 34 is the latest draft of the Noise specification. Noise Explorer is continuously updated in collaboration with the authors of the Noise specification.

authentication cannot be forged. Message contents benefit from ***message secrecy*** *and* ***strong forward secrecy****: if the ephemeral private keys are secure and the initiator is not being actively impersonated by an active attacker, message contents cannot be decrypted by an adversary."*

Furthermore, each message comes with a detailed analysis view that allows the user to immediately access a dynamically generated representation of the state transition functions for this particular message as modeled in ProVerif and a more detailed individual writeup of which security goals are met and why. We believe that this *"pedagogy-in-depth"* that is provided by the Noise Explorer web framework will allow for useful, push-button analysis of any constructed protocol within Noise that is comprehensive.

Noise Explorer's development was done in tandem with discussions with the Noise author: pre-release versions were built around revision 33 of Noise and an update to support revision 34 of the framework was released in tandem with the specification revision draft. Revision 34 also included security grade results for deferred patterns that were obtained directly via Noise Explorer's compendium of formal analysis results. We plan to continue collaborating with the Noise author indefinitely to support future revisions of Noise.

5.2 Noise Explorer Verification Results

Noise Explorer was used to generate ProVerif models for more than 57 Noise Handshake Patterns, all of which were subsequently verified with the results shown in Fig. 7. We found that all of the handshake patterns evaluated met the security goals postulated in the original Noise Specification. Verification times varied between less than 30 minutes for simpler (and less secure) patterns (such as `NN`) to more than 24 hours for some of the more ambitious patterns, such as `IK`. All of the results are accessible publicly using Noise Explorer's compendium interface[7] and the official Noise specification has updated in order to take our results into account.

6 Modeling for Forgery Attacks using Noise Explorer

Using ProVerif, we were able to test for the necessity of certain Noise Handshake Patterns and to document a forgery attack within certain handshake patterns that becomes possible when these rules are not followed appropriately. Essentially, we can compose well-known attack vectors (invalid Diffie-Hellman key shares, repeated AEAD nonces) to attack patterns that rely only on static-static key derivation (`ss`) for authentication.

Consider the pattern `KXS` below:

$$\begin{aligned} KXS: & \\ & \rightarrow s \\ & \dots \\ & \rightarrow e \\ & \leftarrow e, ee, s, ss \end{aligned}$$

This is a variation of the handshake pattern `KX` that uses `ss` instead of `se`, and `es`, so it is a little more efficient while satisfying the same confidentiality and authentication goals. In particular, the responder can start sending messages immediately after the second message.

However, there is an attack if the responder does not validate ephemeral public values. Suppose a malicious initiator were to send an invalid ephemeral public key `e`, say $e = 0$. Then, because of how Diffie-Hellman operations work on X25519, the responder would compute $ee = 0$ and the resulting key would depend only on the static key `ss`. Note that while the responder could detect and reject the invalid public key, the Noise specification explicitly discourages this behavior.

---

[7] `https://noiseexplorer.com/patterns/`

Since the responder will encrypt messages with a key determined only by `ss` (with a nonce set to 0), the malicious initiator can cause it to encrypt two messages with the same key and nonce, which allows for forgery attacks. A concrete man-in-the-middle attack on this pattern is as follows:[8]

In the pre-message phase, A sends a public static key share s_A to B. In the first session:

1. A malicious C initiates a session with B where he pretends to be A. C sends $e = Z$ such that Z^x would evaluate to Z for any x. This effectively allows us to model for forcing an X25519 zero-value key share in the symbolic model.
2. B receives $e = Z$ and accepts a new session with:
   - $h_{B0} = \mathsf{H}(\texttt{pattern\_name})$
   - $ck_{B1} = h_{B0}$
   - $h_{B1} = \mathsf{H}(h_{B0}, s_A, e = Z)$
3. B generates re_1, computes $ee = Z$ and sends back $(re_1, ee = Z, s_B, ss_{AB}, msg_a)$ where s_B is encrypted with $ck_{B2} = \mathsf{H}(ck_{B1}, ee = Z)$ as the key, 0 as the nonce and $h_{B2} = \mathsf{H}(h_{B1}, re_1, ee = Z)$ as associated data.
4. msg_a is encrypted with $ck_{B3} = \mathsf{H}(ck_{B2}, ss_{AB})$ as the key, 0 as the nonce and $h_{B3} = \mathsf{H}(h_{B2}, s_B)$ as associated data.
5. C discards this session but remembers the encrypted message.

In a second session:

1. A initiates a session with B by sending e. So, at A:
   - $h_{A0} = \mathsf{H}(\texttt{pattern\_name})$
   - $ck_{A1} = h_{A0}$
   - $h_{A1} = \mathsf{H}(h_{A0}, s_A, e)$
2. C intercepts this message and replaces it with the invalid public key $Z = 0$.
3. B receives $e = Z$ and accepts a new session with:
   - $h_{B0} = \mathsf{H}(\texttt{pattern\_name})$
   - $ck_{B1} = h_{B0}$
   - $h_{B1} = \mathsf{H}(h_{B0}, s_A, e = Z)$
4. B generates re_2, computes $ee = Z$ and sends back $(re_2, ee = Z, s_B, ss_{AB}, msg_b)$ where s_B is encrypted with $ck_{B2} = \mathsf{H}(ck_{B1}, ee = Z)$ as the key, 0 as the nonce and $h_{B2} = \mathsf{H}(h_{B1}, re)$ as associated data.
5. msg_b is encrypted with $ck_{B3} = \mathsf{H}(ck_{B2}, ss_{AB})$ as the key, 0 as the nonce and $h_{B3} = \mathsf{H}(h_{B2}, s_B)$ as associated data.
6. C intercepts this response.

Notably, the encryption keys (ck_{B3}) and the nonces (0) used for msg_a in session 1 and msg_b in session 2 are the same. Hence, if the underlying AEAD scheme is vulnerable to the repeated nonces attack, C can compute the AEAD authentication key for ck_{B3} and tamper with msg_a and msg_b to produce a new message msg_c that is validly encrypted under this key. Importantly, C can also tamper with the associated data h_{B3} to make it match any other hash value.

---

[8] For simplicity, here we use H to represent the more complex key derivation and mixing functions.

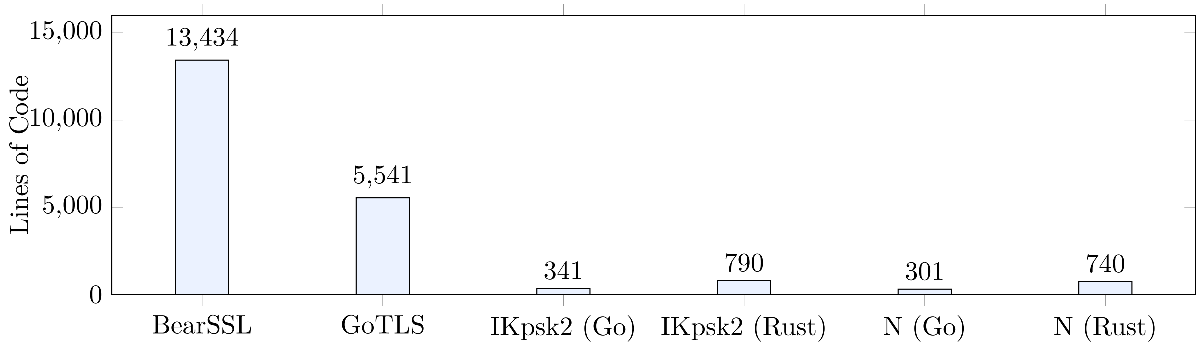

Figure 8: Implementation sizes for BearSSL, Go's standard TLS library and two Noise Handshake Pattern implementations generated by Noise Explorer in Go and Rust. Due to the modular nature of TLS, the BearSSL and GoTLS figures are best-effort approximations.

C replaces the message with $(re = Z, ee = Z, s_B, ss_{AB}, msg_c)$ and sends it to A, where s_B is re-encrypted by C using ck_{B2} which it knows and msg_c is forged by C using the AEAD authentication key for ck_{B3}. A receives the message $(re = Z, ee = Z, s_B, ss_{AB}, msg_c)$ and computes $ck_{A2} = H(ck_{A1}, ee = Z)$ and $h_{A2} = H(h_{A1}, ee = Z)$. A then decrypts s_B. A then computes $ck_{A3} = H(ck_{A2}, ss_{AB})$ and $h_{A3} = H(h_{A2}, s_{AB})$ and decrypts msg_c. This decryption succeeds since $ck_{A3} = ck_{B3}$. The attacker C therefore has successfully forged the message and the associated data.

At a high level, the above analysis can be read as indicating one of three shortcomings:

1. **Using `ss` in Noise Handshake Patterns must be done carefully.** A handshake pattern validation rule could be introduced to disallow the usage of `ss` in a handshake unless it is accompanied by `se` or `es` in the same handshake pattern.
2. **Diffie-Hellman key shares must be validated.** Implementations must validate incoming Diffie-Hellman public values to check that they are not one of the twelve known integers [28] which can cause a scalar multiplication on the X25519 curve to produce an output of 0.
3. **Independent sessions must be checked for AEAD key reuse.** Ephemeral and static public key values are mixed into the encryption key derivation step.

As a result of this analysis, revision 34 of the Noise specification included clearer wording of the validity rule which would render the `KXS` pattern invalid:

"After calculating a Diffie-Hellman shared secret between a remote public key (either static or ephemeral) and the local static key, the local party must not perform any encryptions unless it has also calculated a Diffie-Hellman key share between its local ephemeral key and the remote public key. In particular, this means that:

- *After an `se` or `ss` token, the initiator must not send a payload unless there has also been an `ee` or `es` token respectively.*
- *After an `es` or `ss` token, the responder must not send a payload unless there has also been an `ee` or `se` token respectively."*

7 Automatic Generation of Software Implementations

Noise Explorer can also automatically generate software implementations for arbitrary Noise Handshake Patterns. We target Go and Rust for code generation given their relevance in systems where secure channel protocols are frequently required as well as their adoption by Mozilla, Google, CloudFlare, Facebook and others.

Noise Explorer's ability to generate software implementations complements its ability to assist with designing and formally verifying handshake patterns: immediately after the pattern is designed and its security guarantees are ascertained, it can be prototyped in production systems with a software implementation that is safe to use.

Like its ProVerif model output, Noise Explorer's generated implementations are fully human-readable. If a generated implementation is part of the original implementations described in the Noise specification, then it is also possible to automatically test it against test vectors obtained from Cacophony [29], a Haskell implementation of Noise.

One major benefit of Noise is that it allows for the design and implementation of use case-specific secure channel protocols. For example, WireGuard implements the `IKpsk2` handshake pattern while WhatsApp chooses to implement `XX` and `IK` in different parts of the application. This leads to generated implementations being able to do away with much of the state machine complexity that has plagued TLS, often resulting serious vulnerabilities [30].

7.1 Rust Implementations: Extra Safety Guarantees

Both Go and Rust implementations generated by Noise Explorer are tested to perform fully in constant time. However, generated Rust implementations benefit from the following extra features:

- Usage of memory safe, provably functionally correct, and secret independent Curve25519 and ChaCha20Poly1305 operations provided by FFI bindings to reference implementations from the HACL* [31] high-assurance cryptography library.
- All Hash, Key, Message, and Nonce objects are constrained to their appropriate sizes as defined in the specification.
- Clearing operations for Key and Hash objects are strictly unoptimizable by LLVM and overwrite data with zeros in order to securely erase sensitive key data from memory.
- Nonce objects can only be incremented after successful encryption/decryption and their constructor/destructor functions are called exclusively and simultaneously with those of a wrapping `CipherState`.
- Ephemeral keys are similarly cleared immediately after use in an encryption/decryption operation.

7.2 Limitations of Generated Implementations

The automated generation of secure, side channel resistant implementations of arbitrary handshake patterns in Rust and Go demonstrates that the same compilation pipeline that we use for formal verification model generation can also be used to generate interoperable implementations.

However, while we are able to link the protocol logic of the generated implementations to that of the generated symbolic models, we do not prove anything with regards to the functional correctness or security of the implementation code itself. That is left for future work, and likely can be best accomplished by using specialized frameworks for writing formally verifiable software, such as F* [21].

In Figure 8, we show a cursory comparison between BearSSL [32], Go's standard TLS implementation as well as two handshake patterns generated from Noise Explorer. In all four test cases, care was taken to make the comparison as fair as possible by only including protocol element types that figure across all of the implementations. A strong difference in code "auditability" is apparent.

8 Conclusion and Future Work

In this work, we have provided the first formal treatment of the Noise Protocol Framework. We translate Noise Handshake Patterns into the applied pi-calculus and ormally verify them using ProVerif for a series of target security properties. We coalesce our results into Noise Explorer, an online framework for designing, validating, verifying, reasoning about and implementing arbitrary handshake patterns.

Noise Explorer has already had an impact as the first automated formal analysis targeting any and all handshake patterns. Verification results obtained from Noise Explorer were integrated into the original specification and refinements were made to the validation rules and security goals as a result of the scrutiny inherent to our analysis.

Ultimately, it is not up to us to comment on whether Noise presents a “good” framework, per se. However, we present confident results that its approach to protocol design allows us to cross a new bridge for not only designing and implementing more robust custom secure channel protocols, but also applying existing automated verification methodologies in new and more ambitious ways.

For the formal verification aspect, future work could include the automated generation of computational models to be verified using CryptoVerif and of verified implementations of handshake patterns. The scope of our formalisms could also be extended to include elements of the Noise specification, such as queries to test for identity hiding.

For Noise Explorer’s software implementation features, future work includes targeting the generation of WebAssembly [33] implementations of arbitrary handshake patterns. We also plan to support `no_std` in our Rust implementations, which would allow us to compile for embedded devices.

Acknowledgements

We would like to thank Bruno Blanchet for his invaluable insight with regards to modeling and optimization in ProVerif and for his valuable feedback on earlier drafts of this paper. We would like to thank Trevor Perrin, Jason Donenfeld, Jeff Burdges, Ismail Khoffi, Benjamin Lipp, Loup Vaillant David, Justin Cormack and Ximin Luo for their feedback and encouragement. We also present our thanks to Thomas Pornin for his insightful feedback on BearSSL implementation size estimation.

Noise Explorer has received funding from the following European Union Horizon 2020 projects:

- NEXTLEAP Project (grant agreement №688722).
- CIRCUS (grant agreement №683032).

Additional funding was received from the Web3 Foundation.

References

[1] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy (S&P)*, pages 98–113, 2014.

[2] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information processing letters*, 56(3), 1995.

[3] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti. Verified contributive channel bindings for compound authentication. In *Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[5] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1773–1788, New York, NY, USA, 2017. ACM.

[6] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 5–17, 2015.

[7] Trevor Perrin. The Noise protocol framework, 2015. Available at `http://www.noiseprotocol.org`.

[8] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226–246. Springer, 2013.

[9] 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, October 2016.

[10] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[11] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[12] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 483–502. IEEE, 2017.

[13] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 78–94, 2012.

[14] Jason Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol, 2017. https://www.wireguard.com/formal-verification/.

[15] Jason A Donenfeld. WireGuard: next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[16] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. Cryptology ePrint Archive, Report 2018/080, 2018. `https://eprint.iacr.org/2018/080`.

[17] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[18] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar Formal Protocol Verification Applied*, page 117, 2007.

[19] Guillaume Girol. Formalizing and verifying the security protocols from the noise framework. Master's thesis, ETH Zurich, 2019.

[20] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[21] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in $\mathsf{F}^\star$. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

[22] Mike Hamburg. The STROBE protocol framework. *IACR Cryptology ePrint Archive*, 2017:3, 2017.

[23] David Wong. Disco: Modern session encryption. Cryptology ePrint Archive, Report 2019/180, 2019. `https://eprint.iacr.org/2019/180`.

[24] Mathias Hall-Andersen, David Wong, Nick Sullivan, and Alishah Chator. nQUIC: Noise-based QUIC packet protection. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 22–28. ACM, 2018.

[25] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[26] Phillip Rogaway. Authenticated-encryption with associated-data. In *Ninth ACM Conference on Computer and Communications Security (CCS-9)*, pages 98–107, Washington, DC, November 2002. ACM Press.

[27] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology (CRYPTO)*, pages 631–648. 2010.

[28] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography (PKC)*, pages 207–228, 2006.

[29] John Galt. cacophony: A library implementing the Noise protocol, 2019. `http://hackage.haskell.org/package/cacophony`.

[30] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean-Karim Zinzindohoué. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[31] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017.

[32] Thomas Pornin. BearSSL: a smaller SSL/TLS library, 2019. `https://bearssl.org/`.

[33] WebAssembly Core Specification, 2018. `https://webassembly.github.io/spec/core/bikeshed/index.html`.