Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations

Son Ho, Jonathan Protzenko, Abhishek Bichhawat, Karthikeyan Bhargavan

Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations (Long Version)

Son Ho
Inria Paris

Jonathan Protzenko
Microsoft Research

Abhishek Bichhawat
IIT Gandhinagar

Karthikeyan Bhargavan
Inria Paris

***Abstract*—The Noise protocol framework defines a succinct notation and execution framework for a large class of 59+ secure channel protocols, some of which are used in popular applications such as WhatsApp and WireGuard. We present a verified implementation of a Noise protocol compiler that takes *any* Noise protocol, and produces an optimized C implementation with extensive correctness and security guarantees. To this end, we formalize the complete Noise stack in F*, from the low-level cryptographic library to a high-level API. We write our compiler also in F*, prove that it meets our formal specification once and for all, and then specialize it on-demand for any given Noise protocol, relying on a novel technique called *hybrid embedding*. We thus establish functional correctness, memory safety and a form of side-channel resistance for the generated C code for each Noise protocol. We propagate these guarantees to the high-level API, using defensive dynamic checks to prevent incorrect uses of the protocol. Finally, we formally state and prove the security of our Noise code, by building on a symbolic model of cryptography in F*, and formally link high-level API security goals stated in terms of *security levels* to low-level cryptographic guarantees. Ours are the first comprehensive verification results for a protocol compiler that targets C code and the first verified implementations of any Noise protocol. We evaluate our framework by generating implementations for all 59 Noise protocols and by comparing the size, performance, and security of our verified code against other (unverified) implementations and prior security analyses of Noise.**

I. Introduction

Modern distributed applications rely on a variety of secure channel protocols, including TLS, QUIC, Signal, IPsec, SSH, WireGuard, OpenVPN, and EDHOC. Despite the similarity in their high-level goals, each of these protocols makes significantly different design choices based on the target network architecture, authentication infrastructure, and desired security goals. For example, the Transport Layer Security (TLS) protocol is used to secure live TCP connections between clients and servers using the X.509 public key infrastructure. In contrast, the Signal messaging protocol aims to provide strong confidentiality guarantees like post-compromise security [1] for long-running asynchronous messaging conversations between smartphones. All these protocols form a cornerstone of Internet security, so the correctness and security of their varied designs and diverse implementations is a tangible concern.

Security Analyses of Secure Channels. Several prior works establish security theorems for well-known secure channel protocols. However, as protocols get more complex, building and checking pen-and-paper proofs for complete protocols becomes infeasible. To address this, formal verification tools are now routinely applied to obtain mechanized security proofs for cryptographic protocols. For example, tools like ProVerif [2] and Tamarin [3] have been used to automatically analyze protocols like TLS and Signal [4], [5], [6], by relying on abstract symbolic assumptions on the underlying cryptography. Computational provers like CryptoVerif [7] and Computational RCF [8] have also been used to verify some of these protocols, providing more precise security guarantees than symbolic tools, but requiring more human intervention [9], [10], [5], [6].

We refer the reader to [11] for a full survey of computer-aided cryptographic proofs. On the whole, verification tools have now reached a level of maturity such that they can analyze the high-level design of most modern cryptographic protocols.

Verified Protocol Implementations. Even if the design of a protocol has been verified, writing a secure implementation remains a challenge. Protocol implementations have to account for many details that are left out of high-level security proofs, such as the crypto library, message formats, state machines, key storage and management, multiple concurrent sessions, and a high-level user-facing API that is easy for non-cryptographers to use. Each of these components has been subject to notable bugs resulting in embarrassing vulnerabilities like HeartBleed [12] and SMACK-TLS [13]. Many of these flaws were not found even through extensive testing.

In response, several works have sought to build high-assurance protocol implementations using formal verification tools. The most notable of these is miTLS [10], a verified reference implementation of the TLS 1.2 protocol in F#, built hand-in-hand with modular proofs of computational security at the code-level. Follow-up works verify efficient C implementations of various components of TLS 1.3, including the TLS packet formats [14], the cryptographic library [15], and the record layer [16]. Other works have built high-assurance protocol implementations in OCaml [17], JavaScript [5], [6], WebAssembly [18], and Java [19].

Despite these advances, verifying a full cryptographic protocol implementation written in a performance-oriented language like C is highly resource-intensive and can take years of work. Consequently such projects have only been attempted for important protocols like TLS. In this paper, we tackle the

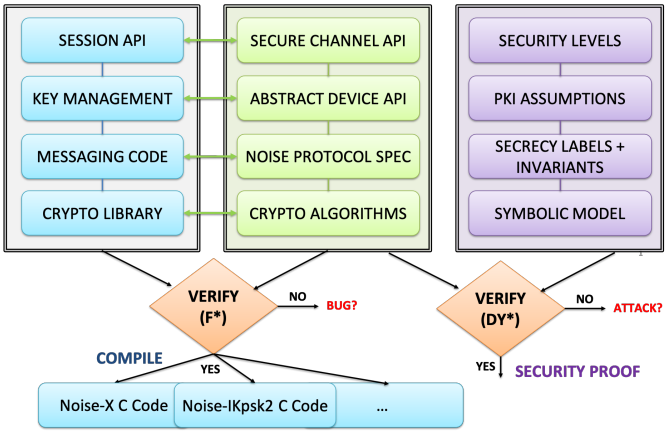

Fig. 1. **Noise* Architecture.** Left: Noise protocol stack implemented in Low*; Middle: generic formal specification of Noise in F*; Right: security specifications for each layer using the DY* framework. After verification, the Low* code is specialized and compiled to obtain C code for each protocol.

problem of generalizing and scaling up the security analysis of protocol implementations in a way that they can be applied to entire families of cryptographic protocols. Hence, we lower the human effort involved to build verified protocol libraries.

The Noise Protocol Framework. We target verified implementations of the Noise Protocol framework, which provides a general notation and execution rules for a large class of secure channel protocols. The Noise specification [20] currently describes 59 protocols, specifies message-level security properties for each of these protocols, and precisely defines all the cryptographic steps needed to send and receive protocol messages. Although these 59 protocols are centered around Diffie-Hellman and pre-shared keys, the specification language is itself extensible and can easily handle protocols with signatures and key encapsulation mechanisms in the future.

Noise is an ideal target for formal verification in that it covers a large class of similar protocols. For the same reason, it is a challenging target, since we would like to develop generic proofs that apply to all 59 Noise protocols and their implementations, rather than verify each protocol individually.

Several prior works present formal analyses for various Noise protocols [21], [22], [23], [9] and multiple open source libraries implement various subsets of Noise. However, until this work, there has been no verified implementation of Noise. Consequently, many security-critical protocol elements, including key management and state machines remain unstudied. Our goal is to develop a library of verified high-performance implementations of Noise protocols in C, with formal proofs of correctness and security that cover all these low-level details.

Our Approach. We build a verified implementation of Noise, following the methodology depicted in Figure 1. All our code is written and verified using the F* programming language [24].

We first write a formal specification of Noise in F* (middle column) by carefully encoding the message-level functions described in the Noise specification document [20] and linking them to F* specifications of crypto algorithms. Our specification can be read as an *interpreter* for the Noise protocol notation, and we can use it to execute any Noise protocol. We extend this interpreter with F* specifications of key validation and management and a high-level session API, both which are left unspecified in the Noise document. Hence, we obtain a full specification for the Noise protocol stack, starting from the crypto layer to the user-facing API (see Section II).

Next, we write a low-level implementation of Noise (left column) using Low* [25], a subset of F*, and prove that it matches the formal specification. We use the HACL* verified cryptographic library to instantiate the cryptographic layer [26]. We develop a protocol compiler using a novel technique called *hybrid embedding* that allows us to write and verify generic code for all Noise patterns, prove them correct against the interpreter spec once and for all, and then specialize and compile the verified code into standalone C implementations for each Noise protocol (See Section III).

On top of our protocol compiler, we design and build a session management layer that handles multiple sessions in parallel and handles error conditions. We write a verified key storage module that securely stores long-term keys both in-memory and on-disk. Finally, we build a verified high-level user-facing API that provides a simple, secure, misuse-resistant interface for applications (See Section IV).

Our Low* code is verified with respect to our formal specification of Noise, but this does not mean that it is secure. For example, our protocol API may accidentally expose long-term keys to the adversary, or it may allow data from two sessions to be mixed up, which may not violate the Noise spec but would result in serious security vulnerabilities. To fill this gap, we extend our verification with a symbolic security analysis of the full protocol specification using a recent framework called DY* [27]. We set and prove security goals for each layer in our implementation (right column), linking a symbolic model of cryptography all the way to verified high-level API security goals. Notably, our analysis is generic and verifies all Noise protocols in a single proof, unlike prior work which needed to run verification tools on each individual protocol. (See Section V).

Finally, we demonstrate our framework by compiling verified implementations for all 59 Noise patterns and compare the results with prior work (See Section VI).

Summary of Contributions. We present the first verified implementations for Noise and the first verified protocol compiler that generates C code. Our compiler proof relies on a novel technique, called hybrid embedding, which is of independent interest. We also provide verified implementations of key storage and a high-level protocol API, both of which are novel and reusable in other developments. Finally, we provide the first modular mechanized symbolic security proofs of Noise at the level of a detailed executable protocol specification.

II. A Formal Functional Specification of Noise

The Noise Protocol Specification [20] defines a succinct notation and precise execution rules for a family of secure channel protocols that primarily use Diffie-Hellman and pre-shared keys for confidentiality and authentication, yielding a

Protocol Name	Message Sequence	Payload Security Properties			
		←		→	
		Auth	Conf	Auth	Conf
X	← s				
	...				
	→ e, es, s, ss $[d_0]$	-	-	A1	C2
	→ $[d_1, d_2, \ldots]$	-	-	A1	C2
NX	→ e	A0	C0	A0	C0
	← e, ee, s, es $[d_0]$	A2	C1	A0	C0
	↔ $[d_1, d_2, \ldots]$	A2	C1	A0	C5
XX	→ e	A0	C0	A0	C0
	← e, ee, s, es $[d_0]$	A2	C1	A0	C0
	→ s, se $[d_1]$	A2	C1	A2	C5
	↔ $[d_2, d_3, \ldots]$	A2	C5	A2	C5
IKpsk2	← s				
	...				
	→ e, es, s, ss $[d_0]$	A0	C0	A1	C2
	← e, ee, se, psk $[d_1]$	A2	C4	A1	C2
	→ $[d_2]$	A2	C4	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5

Fig. 2. **Example Noise Protocols and Security Guarantees.** X: a one-way authenticated encryption protocol; NX: an interactive Diffe-Hellman key exchange with an unauthenticated initiator; XX: an interactive mutually-authenticated key exchange using Diffie-Hellman; IKpsk2: an interactive mutually-authenticated key exchange using Diffie-Hellman and a pre-shared key; At each stage of a protocol, we note the expected authentication level (A0-A2) and confidentiality level (C0-C5) for messages in each direction (← / →).

total of 59 protocols with varying authentication and secrecy properties. We begin by an informal overview of the syntax and semantics of Noise protocols, before describing our formal specification of Noise in the F* programming language [24].

A. Noise Protocol Notation

Four example Noise protocols are shown in Figure 2. The message sequence for each protocol is divided into three phases. The first phase (before the dotted line) consists of *pre-messages* exchanged by the two parties out-of-band before the protocol begins. The second phase is the main *handshake* where the two parties exchange fresh key material to establish a series of payload encryption keys with gradually stronger security guarantees. Once the handshake is complete, the protocol enters the third *transport* phase where both parties can freely exchange encrypted application messages in both directions.

The handshake is described as a sequence of messages between an initiator (I) and a responder (R), where each message is as a sequence of *tokens*. Each participant maintains a *chaining key* k_i that it uses to derive the *payload encryption key* at each step; both of which are initially set to public constants derived from the protocol name. The chaining key evolves as each handshake token is processed.

Consider a handshake between I and R, where I has a static Diffie-Hellman key-pair (i, g^i) and generates an ephemeral key-pair (x, g^x); R has a static key-pair (r, g^r) and ephemeral key-pair (y, g^y); and the two may share a pre-shared key psk. Then the semantics of each token sent from I to R is as follows (tokens in the reverse direction are handled similarly):

- e: means that I includes g^x in the message;
- s: I includes its static public key (g^i) in the message, encrypted under the current payload encryption key;
- es: I computes the ephemeral-static Diffie-Hellman shared secret g^{xr} and mixes it into the chaining key c_i, obtaining a new chaining key c_{i+1} and payload encryption key k_{i+1};
- se: I mixes the static-ephemeral shared secret g^{iy} into c_i;
- ee: I mixes the ephemeral-ephemeral secret g^{xy} into c_i;
- ss: I mixes the static-static shared secret g^{ir} into c_i;
- psk: I mixes the pre-shared key psk into c_i.

After processing each sequence of tokens according to the above rules, at the end of each message, the sender (I) also includes a (possibly empty) payload encrypted under the current payload encryption key. These payloads are implicit in Noise notation, but we note them explicitly $(d_0, d_1, \ldots)$ in Figure 2.

On receiving a message constructed using the above rules, the responder R performs the dual operations to parse the remote ephemeral key (e), decrypt the remote static key (s), and computes the same sequence of chaining and payload encryption keys to decrypt the payload. In addition to the keys, each participant also maintains a hash of the protocol *transcript*, which is added as associated data to each encrypted handshake payload (to prevent handshake message tampering.)

X: One-Way Encryption. The protocol X is a one-way protocol that encrypts data in a single direction, from an initiator I to a responder R. As such, this protocol can be considered a replacement for constructions like NaCl Box [28] or HPKE [29] for encrypting files or one-way messages.

We now break down the notation for this protocol, which appears in Figure 2 under "message sequence". The pre-message token s, assumes that I has received R's *static* public key g^r before the handshake. The handshake itself consists of a single message (from I to S) with four tokens (e, es, s, ss) followed by an encrypted payload (d_0). Here, ephemeral-static Diffie-Hellman (es) serves to provide confidentiality for k_1 (even if I's static key were compromised), whereas static-static Diffie-Hellman (ss) is used to authenticate I. After the handshake, I can send any number data messages $(d_1, d_2 \ldots)$ to R, using the final payload encryption key.

NX: Server-Authenticated Key Exchange. The protocol NX is a unilaterally authenticated key exchange protocol, where R is authenticated but I is not. Hence, this protocol can be seen as a replacement for TLS as it is used on the Web. The main difference from X is that it has no pre-messages, and has a second message that uses ephemeral-ephemeral Diffie-Hellman (ee) to provide forward secrecy.

XX: Mutual-Authentication. We can extend NX to a mutually-authenticated protocol by adding a third handshake message that uses I's static key (se). This yields a different Noise protocol called XX, which is one of the protocols used in WhatsApp. Both NX and XX are single round-trip (1-RTT) protocols since the initiator has to wait for the response before it can send its first encrypted message. However, in scenarios where I already knows R's static public key (g^r) via a pre-message, it can use this prior knowledge to start sending data with the first message (0-RTT), but with different secrecy and confidentiality guarantees.

IKpsk2: Mutual-Authentication and 0-RTT. The IKpsk2 protocol, which is used by the WireGuard VPN, supports

mutual authentication and 0-RTT by relying on both Diffie-Hellman and pre-shared keys, and hence provides some of the strongest security properties among all Noise protocols.

The protocol starts like X but includes authenticated messages in both directions; it uses four Diffie-Hellman operations and also a pre-shared key in the second message (psk token) for additional protection against compromised static keys (and future quantum adversaries). Removing the psk token yields a protocol called IK, which is also used in WhatsApp.

B. Formalizing Noise in F*

We define a series of F* types that encode the syntax of Noise protocols. We define algebraic datatypes (enumerations) for pre-message and message tokens. We then define a handshake pattern as a record type containing a protocol name, a pre-message from I to R (premessage_ir), a pre-message from R to I (premessage_ri), and a list of handshake messages in alternating directions (first I to R, then R to I, and so on):

```
type premessage_token = | PS | PE
type message_token = | S | E | SS | EE | SE | ES | PSK
type handshake_pattern = {
  name : string;
  premessage_ir : option (list premessage_token);
  premessage_ri : option (list premessage_token);
  messages : list (list message_token)}
```

We also define some convenient notations in F* to construct a handshake_pattern. For example, IKpsk2 is written as(note that we omit the implicit payloads d_i):

```
let pattern_IKpsk2 =
  hs "IKpsk2" [
    ∼<<∼ [PS];
    ∼>∼ [E; ES; S; SS];
    ∼<∼ [E; EE; SE; PSK]]
```

The Noise specification defines a set of syntactic validity rules to ensure that the resulting protocols are implementable and secure. An example functional constraint is that a protocol should not use the token ee before e has been sent in both directions: both participants must have received their counterpart's public ephemeral key in order to use it in the Diffie-Hellman. A security constraint is that a session key based on a psk token should not be used for encryption unless an e has also been sent (otherwise there could be encryption nonce reuse.) We encode these rules as a boolean function over handshake patterns, and check that it holds for all 59 patterns.

```
val well_formed: handshake_pattern → bool
```

Types for the Handshake State. To formalize the execution rules, we closely follow the Noise specification by defining the handshake state and functions over this state. Each type and function in our specification is parameterized by a config type specifying three cryptographic algorithms: a Diffie-Hellman group, an AEAD encryption scheme, and a hash algorithm:

```
type config = dh_alg & aead_alg & hash_alg
```

The cipher_state type consists of an AEAD key and a counter; it can be used for AEAD encryption and decryption:

```
type cipher_state = {k : option aead_key; n : nat}
```

The symmetric_state type represents the cryptographic state of a Noise handshake. It contains a hash of the protocol transcript (essentially all the message tokens processed so far), the current session key, called chaining_key in Noise, and a cipher_state (derived from the chaining_key) which is used for encrypting static keys and payloads during the handshake:

```
type symmetric_state (cfg : config) = {
  h : hash cfg; ck : chaining_key cfg; c_state : cipher_state}
```

The main handshake_state type contains the full state of a Noise handshake for a given participant; it includes the current symmetric_state and all the private, public, and shared keys currently known to the participant:

```
type handshake_state (cfg : config) = {
  sym_state : symmetric_state cfg;
  static : option (keypair cfg);
  ephemeral : option (keypair cfg);
  remote_static : option (public_key cfg);
  remote_ephemeral : option (public_key cfg);
  preshared : option preshared_key}
```

Message Processing Functions. The Noise specification document describes a series of functions over the three state objects, which we faithfully encode in F*. The highest-level operations defined by the document are functions for sending or receiving one handshake or data message. We describe the F* code for the handshake sending functions below.

First, we define a function that implements the sending operation for a single token as a case analysis over the 7 possible tokens (we show two cases below):

```
let send_message_token (cfg:config) (initiator is_psk:bool)
                       (tk:token) (st:handshake_state cfg) :
    result (bytes & handshake_state cfg) =
  match tk with
  | S → (match st.static with
         | None → Fail No_key
         | Some k →
           (match encrypt_and_hash cfg k.pub st.sym_state with
            | Fail x → Fail x
            | Res (cipher, sym_st') →
                Res (cipher, { st with sym_state = sym_st'; })))
  | EE → dh_update cfg st.ephemeral st.remote_ephemeral st
  | ...
```

The function send_message_token takes as arguments: a config, a boolean flag indicating whether the sender is the initiator, a boolean flag indicating whether the current protocol uses psk, a token tk and a handshake state st. If the token is an S, the code finds the sender's static key (st.static), encrypts it and adds to the transcript hash (encrypt_and_hash), returning the ciphertext (cipher) and the updated handshake state. If the token is an EE, the sender reads its ephemeral private key (st.ephemeral), the peer's ephemeral public key (st.remote_ephemeral) and calls the dh_update function that computes the Diffie-Hellman shared secret, mixes it into the current chaining_key, and returns an empty bytestring and the updated handshake state. The other cases are similar.

Building on this token-level function, we then write a function send_message_tokens that recursively calls send_message_token to process an arbitrary list of tokens, and use it to define a high-level function send_messagei for sending the i-th handshake message in a handshake_pattern.

A similar sequence of functions builds up to the top-level handshake receive function recv_messagei. Using these and other message-level functions in our specification, we can construct or process any pre-message, handshake message, or application data message in a Noise protocol.

Comparison with Prior Noise Models. Three features of our specification are notable. First, our F* code is executable and precisely matches the Noise specification at the byte level. Indeed, by linking our specification code with the HACL* cryptographic library, we are able to extensively test our specification against test-vectors from other Noise implementations. Second, we use recursive functions to model protocols and messages of arbitrary length, even though in practice, we may only care about the 59 protocols in the current Noise specification. Third, our code is structured as a *protocol interpreter*, and hence provides a single generic functional specification for *all* Noise protocols.

These three features are in contrast with prior formal models of Noise protocols that were written for various security analyses [21], [22], [23], [9]. These models ignore many low-level protocol details, are not precise at the byte level, and are not testable. Their modeling languages cannot handle generic recursion or protocol interpreters, and so require a separate model for each Noise protocol. We believe our F* specification more closely captures the spirit of Noise and serves as a formal companion to the Noise specification.

C. Noise Protocol Security Guarantees

Different Noise protocols offer different security guarantees. Even within a single protocol, the confidentiality and authentication guarantees obtained by the initiator and responder often differ. These guarantees typically improve with each handshake message and stabilize after the handshake completes. For example, IKpsk2 allows application data to be sent both during the handshake (d_0, d_1) and after the handshake ($d_2, d_3, \ldots$), and each of these messages has different security guarantees. The Noise specification [20] defines 3 levels of authenticity (A0-A2) and 6 levels of confidentiality (C0-C5). Figure 2 lists the security levels at each stage of our three protocol examples, and Appendix A lists them for all 59 Noise patterns.

Payload Authentication Properties. The three authentication levels are: **A0**: No authentication; **A1**: Sender authentication vulnerable to Key Compromise Impersonation (KCI) attacks; **A2**: Sender authentication without KCI attacks.

Consider a Noise protocol session between A and B, where B receives a message M at authentication level A2 (supposedly) from A. If B successfully decrypts this message, it has the guarantee that the message was indeed sent by A, unless the long-term static key of A (static Diffie-Hellman private key and/or PSK) has been compromised (i.e., leaked to the attacker) before the message was received. Authentication level A1 is weaker: it only guarantees message authenticity if the static keys of both A and B are uncompromised.

For a more formal illustration, in a prover like ProVerif, authentication level A1 would correspond to a security query written in terms of events triggered by the sender, receiver, and the adversary. The sender A triggers Sent(A,B,M) before sending a message; the receiver B triggers Recv(B,A,M) after processing the message; the adversary triggers LongTermCompromised(P) whenever it compromises the static keys of a principal P. (We assume that ephemeral keys are never compromised.) The resulting security query is written as follows:

```
query A:prin, B:prin, M:bitstring;
      event(Recv(B,A,M)) ⟹ event(Sent(A,B,M)
                              || event(LongTermCompromised(A))
                              || event(LongTermCompromised(B))
```

The query for authentication level A2 simply removes the last line (event(LongTermCompromised(B))). For reference, these queries correspond closely to the queries generated by a prior analysis of Noise in ProVerif [22].

For example, in NX, the initiator is never authenticated, so messages in the forward direction ($\rightarrow$) in Figure 2 always have authentication level A0. The responder is fully authenticated and so its messages to the initiator are at level A2. In X and IK, the first message is authenticated by the initiator, but authentication is based on static-static Diffie-Hellman (ss), which means that if the responders's static key is compromised, an attacker can impersonate the initiator to the responder, resulting in a KCI attack. Hence, the authentication level is A1 for forward messages ($\rightarrow$), until the third message when the static-ephemeral Diffie-Hellman (se) token strengthens the initiator's authentication level to A2.

Payload Confidentiality Properties. The six confidentiality levels, in increasing order of strength, are as follows: **C0**: No confidentiality; **C1**: Confidentiality only against passive adversaries; **C2**: Confidentiality against active adversaries, with weak forward secrecy against sender static compromise; **C3**: Weak forward secrecy against sender and receiver static compromise; **C4**: Strong forward secrecy unless sender static was compromised before message; **C5**: Strong forward secrecy.

Of these, the first two levels offer very weak confidentiality, in that an active network adversary can read a payload sent at level C0 or C1. Levels C2-C5 offer incremental degrees of forward secrecy, depending on which subset of static keys may be compromised and when. C2 offers confidentiality as long as the sender's ephemeral key and the recipient's static keys remain uncompromised. C3 additionally allows the receiver's static key to be compromised as long as the peer ephemeral public key at the sender corresponds to an uncompromised ephemeral private key at the recipient. C4 allows the sender and recipient's static keys to be compromised after the message is sent. C5 provides confidentiality even if the sender's static keys were compromised before the message was sent.

In a tool like ProVerif, encoding forward secrecy properties requires the use of phases to enforce an ordering between

protocol messages and compromise events. For example, we would run the target protocol session in phase 0 and transition to phase 1 at the end. We would then allow the attacker to compromise static keys in both phase 0 and phase 1, and state secrecy queries for each confidentiality level in terms of when these keys can be compromised. (As usual, we disallow the compromise of ephemeral keys.) Hence, to model level C4, we write a ProVerif query of the form:

```
query A:prin, B:prin, M:bitstring;
      (attacker_p1(M) && Sent(A,B,M)) ⟹
                          event(LongTermCompromised_p0(B))
                       || event(LongTermCompromised_p0(A))
```

That is, messages sent from A to B are confidential in phase 1 unless the static keys of A or B were compromised in phase 0. The query for C5 is stronger, it removes the last disjunct, and hence guarantees confidentiality even if A were compromised in phase 0 (when the session is still running.)

In Figure 2, X offers confidentiality at level C2 because there is no fresh ephemeral provided by the recipient. NX offers strong forward secrecy at level C5 for messages to the responder, but only level C1 for messages to the unauthenticated initiator. IKpsk2 provides level C5 confidentiality in both directions from the third message. However, the first message only offers level C2 (like X) and the second message only offers level C4 since an attacker who knows the responder's static private key and PSK will be able to forge the first message, record the second message, and later compromise the initiator's static key to obtain the session key and decrypt the payload.

We define an F* function that computes the authentication and confidentiality levels for each message in each handshake_pattern (see Appendix A). We confirm that it agrees with the Noise specification on the 38 protocols annotated in the document, and we also compute levels for the 21 PSK patterns not annotated in the specification. In Section V, we show how these security levels are mapped to precise security goals stated as trace properties and we prove that our protocol specification meets these goals.

D. A High-Level API for Noise

A full protocol implementation has to handle many security-critical details beyond message processing. For example, in the NX protocol, when the initiator receives the responder's static key in the second message, it has to *validate* this key. Otherwise, there is no guarantee it is talking to the intended responder and all authenticity and confidentiality guarantees are lost. Similarly, in X and IKpsk2, the initiator static key needs to be validated against some database of known initiators. In PSK-based protocols like IKpsk2, the responder does not know what PSK to use until it sees the initiator's static key; so we need a way for the responder to dynamically retrieve and validate a PSK based on a protocol message. An implementation that skips or incorrectly implements these key validation steps becomes vulnerable to serious attacks. However, none of these validation steps are documented in the Noise specification and so are left for the application layer to handle.

It is unrealistic to expect an application programmer who uses Noise to have the intimate knowledge needed about a specific Noise protocol in order to directly use the messaging functions, perform all the required validation steps, and know when it is safe to send or receive data.

We address this gap by formally specifying (and implementing) a high-level API that combines several layers: a session-based API that hides message-level protocol details, secure key storage with user-provided policies for key management, built-in validation steps and a defensive user-friendly interface that provides clear guidance on when it is safe to send or receive data over a Noise session. For example, sending secret application data after the first message of NX would be disastrous, but may be safe with IKpsk2. §IV describes our implementation of this high-level API in C.

III. Implementing a Noise Compiler in Low*

Our specification (§II) may run via the OCaml backend of F*, which we use for testing and spec-validation purposes. This execution path suffers, however, from slow performance: in F* specifications, integers compile as infinite-precision bignums; sequences compile to persistent functional lists; and execution relies on OCaml's runtime system and garbage collector.

We now set out to write a low-level, efficient implementation of Noise protocols that does not suffer from such performance shortcomings. This section focuses on a novel technique called "hybrid embeddings", a key technical ingredient that allows us to author low-level code that remains parametric over the choice of Noise pattern, in a fashion similar to the interpreter. With hybrid embeddings, we verify the low-level code once then generate for free any number of specialized implementations for any Noise patterns: doing so, we minimize the verification effort while still guaranteeing low-level performance.

A. Warm-Up: Low* Implementation of ss

For our efficient, low-level implementation of Noise protocols, we use Low*. Low* is a subset of F*; or, said differently, Low* is a *shallow embedding* of a well-behaved subset of C into F*. Thanks to F*'s powerful effect system, Low* defines a CompCert-like C memory model, which captures heap- and stack-based allocations. A set of distinguished types, combinators and libraries provides users with workingtools to operate on mutable arrays, machine integers, const pointers, and so on. Low* has been used for cryptographic libraries [26], [30], providers [15], protocol record layers [31], [32] and parsers [14].

In contrast to §II, where functions were pure, Low* functions use a new set of effects: Stack and ST. Consider the function that performs the required processing for the SS token.

```
inline_for_extraction
let send_ss (nc: iconfig) (ssdhi: ssdh_impls nc)
 (ssi: static_info) (initiator: bool) (is_psk: bool)
 (st: valid_send_token_hsm nc is_psk SS ssi):
 Stack error_code
 (requires (fun h →
   live h st.static ∧ live h st.remote_static ∧
   not (is_null st.static) ∧ not (is_null st.remote_static) ∧
```

```
    loc_disjoint (loc st.static) (loc st.remote_static) ∧ ... ∧
    sym_state_invariant st.sym_state ∧ nc.dh_pre ∧ ...))
  (ensures (
    let st0_v = eval_handshake_state h0 st ssi in
    let st1_v = eval_handshake_state h1 st (ssi_init_sk ssi) in
    let r_v = Spec.send_message_token initiator is_psk SS st0_v in
    match to_prim_error_code r, r_v with
    | CSuccess, Res (..., st1'_v) → st1_v == st1'_v ∧ ...
    | CDH_error, Fail DH → ⊤| _ → ⊥))
= ssdhi.dh_update ssi st.static st.remote_static st
```

Many of the parameters resemble the ones we saw earlier (§II). The iconfig, for *implementation* configuration, extends a spec-level config with low-level specific preconditions such as "our DH implementation requires AVX2" (nc.dh_pre). The ssdhi parameter contains our choice of implementation for cryptographic operations related to the symmetric state and DH; the Low* code is not only generic over the choice of algorithm (like the earlier specification), it is also generic over the choice of implementation. As an example, if the iconfig commits to Curve25519 for the DH algorithm, our code can operate either with HACL*'s Curve51 or Curve64 implementation. The ssi parameter stands for "state static-information"; it contains statically-known information, such as whether at this point of the handshake a symmetric key has been derived or not; and it also contains the nonce (sequence number) to be used for the cryptographic operations. Finally, initiator and is_psk are similar to the parameters we saw earlier (§II).

The function signature exhibits typical features of Low*. The st argument represents the low-level state of the protocol, which can be reflected in a given heap h0 as a high-level state, using eval_handshake_state h0 st ssi. The Stack return effect indicates that the function is valid vis-à-vis the C memory model *and* only performs stack allocations (this latter restriction can be lifted by using the ST effect). The pre-condition covers spatial (disjointness) and temporal (liveness) preconditions; as well as functional correctness requirements, such as the symmetric state invariant and the implementation-specific preconditions. In the post-condition, we elide memory-related predicates (e.g., only the protocol state is modified by a call to this function) for clarity. We focus instead of functional correctness: st0_v reflects low-level state st as a spec-level state before calling send_ss; similarly, st1 reflects st after calling the function. If we execute the interpreter on st0 and obtain st1', then both st1' and st1 coincide, i.e., if the specification guarantees success, so does the low-level implementation with the same result; if the specification errors out, so does the low-level implementation; no other outcome is allowed.

B. A Generic Low* Implementation

Inspired by the generic spec-level interpreter, we now write an *even more generic* low-level function that not only works for any choice of algorithm, implementation, responder and PSK, but also works for any Noise token.

```
inline_for_extraction
let send_message_token nc ssdhi ssi initiator is_psk
  tk st out: (rtype (send_token_return_type ssi is_psk tk))
= match tk with
  | S → send_s nc ssdhi ssi initiator is_psk st out
  | E → send_e nc ssdhi ssi initiator is_psk st out
  | ... → ... (* identical for SS, EE, SE, ES, PSK *)
```

The send_message_token function above attains the same level of genericity as the specification. Even the return type of the function is generic: send_token_return_type captures the fact that SS returns an error code (for DHs that compute to 0), whereas S does not, by reducing at compile-time to error_code or unit, respectively. (Here, our specification is more precise than the Noise specification, which leaves it up to the user to determine whether a DH that computes 0 is an error.) Our function can thus be used for *all* Noise protocols: the initial match acts as an interpreter, examines the Noise token, then dispatches execution to a suitable set of Low* functions.

Our style saves a tremendous amount of verification effort: rather than replicating the effort for 59 protocols, we extract the commonality, capture it with dependent types, and proceed to write send_message_token once and for all. The challenge now remains to ensure that the function generates valid C code that eliminates all runtime checks on the nature of the token.

To that end, we rely on implicit staging and compile-time partial evaluation via F*'s normalizer. The first six parameters of the function are compile-time parameters: once a Noise protocol is chosen, their concrete value is statically known; and the F* compiler is capable of performing enough partial evaluation at compile-time that all uses of these parameters disappear *before* the code is even extracted to C.

Consider, for instance, the X protocol we saw earlier. At compile-time, we pick concrete values for the choice of algorithms (nc) and implementations (ssdhi). For the first handshake message, we call send_message_token, with ssi.has_key = false, ssi.nonce = 0, initiator = true, is_psk = false and of course tk = E. Thanks to the "inline for extraction" keyword, F* aggressively reduces the definition of send_message_token; the match reduces away, leaving only a call to send_e. This latter function itself further reduces: for instance, any statement of the form if is_psk disappears, meaning we ignore the symmetric key generation induced by PSK patterns. Once partial evaluation is done, the code contains only the bare minimum set of operations needed for the first token E of the X protocol, and all compile-time parameters are gone.

C. Hybrid Embeddings

Looking back at §II, we can think of our earlier specification as an interpreter for Noise patterns; or, dually, as an evaluator defining the semantics of a deeply embedded domain-specific-language (DSL), in our case the language of Noise patterns. Unlike shallow embeddings, deep embeddings operate on a *representation* of the target language within the host language; doing so, they enjoy a great deal of flexibility since they are not confined to the syntax of the host language.

The match in the above function is a compile-time match that operates on the deeply embedded representation of Noise patterns, and gets partially evaluated away. We dub this style a hybrid embedding: the code evaluated at compile-time operates over a deep embedding (the Noise patterns), but after partial

evaluation, all that is left is a shallow embedding (the Low* code, which executes at runtime).

The hybrid style allows us to stage and automate the production of Low* code; rather than writing Low* code by hand, we embed a protocol compiler that executes on F*'s compile-time reduction facilities. This style is already useful for send_message_token; but there is no reason to limit ourselves to simple matches and ifs. We now show how to execute arbitrary pure F* code at compile-time, including recursion, to completely automate the production of a specialized Noise protocol instance.

```
[@@ strict_on_arguments [5]] inline_for_extraction
let rec send_message_tokens (nc: iconfig) ssi initiator
  is_psk tokens st outlen out =
match tokens with
| [] → success _
| tk :: tokens' →
 [@inline_let] let tk_outlen = token_message_vs nc ssi tk in
 let tk_out = sub out 0ul tk_outlen in
 let r1 = send_message_token ssi initiator is_psk tk st tk_out in
 if is_success r1 then
  let outlen' = outlen −! tk_outlen in
  let out' = sub out tk_outlen outlen' in
  [@inline_let] let ssi' = send_token_update_ssi is_psk tk ssi in
  let r2 = send_message_tokens ssi' initiator
    is_psk tokens' st outlen' out' in
  compose_return_type ssi is_psk true tokens' tk r2
 else
  compute_return_type ssi is_psk true tk tokens' r1
```

The function above now operates over a *list* of tokens; that is, it generates Low* code for an entire Noise handshake message. Naturally, the function cannot extract as-is: operating over pure, persistent lists in low-level efficient, idiomatic C is a no-go. The goal is to ensure that the subset of send_message_tokens that performs a (pure) recursion over the argument pattern (denoted in bold) is always evaluated away at compile-time when applied to constant arguments. To this end, we allow F* to unfold recursive definitions (elided); to prevent infinite compile-time recursion, we restrict the unfolding to applications where the fifth argument (tokens) is concrete, via the strict_on_arguments keyword. The inline_let attribute indicates pure computations to be inlined at extraction-time. (We use extraction-time and compile-time interchangeably.) We use the keyword for compile-time parameters or constants computed from such parameters.

The function is verified once and for all, meaning that we now have a verification statement for *any* list of noise tokens. At extraction-time, the user applies the function to five concrete arguments. If pattern is [E; ES; S; SS], then after a few steps of reduction, we obtain:

```
let r1 = send_message_token ... E ... in
if is_success r1 then ...
  let r2 = send_message_tokens ... [ ES; S; SS ] ... in ...
```

As computing E always succeeds, is_success r1 reduces to true, in turn eliminating the else branche entirely. Partial evaluation then continues until all compile-time code has disappeared; structural recursion over the list of tokens is over; and all that is left is a sequence of efficient Low* calls that implements the specification for the given list of tokens.

We use this style of hybrid embedding all throughout our low-level protocol code implementation, which allows us to substantially reduce the verification effort. The following section (§IV) shows how to extend this style to generate the entire state machine of a Noise protocol.

D. Hybrid Type Definitions And Function Signatures

We use hybrid embeddings further to optimize internal type definitions and user-facing functions.

For type definitions, we insist on generating C code that contains no superfluous fields. This is useful not only in case the code's internals are audited; but also to ensure that no extra space is consumed in e.g., the internal state of the handshake. To that end, our types reduce at compile-time; consider, for instance:

```
type handshake_state_t nc ssi ... is_psk ... = { ...
  psk: if is_psk then lbuffer ... else unit; ... }
```

If the chosen Noise protocol requires it, the psk field is an array of bytes. If the Noise protocol does not use a PSK, the generic type reduces to unit, which is then guaranteed to be eliminated by KReMLin [25], the Low*-to-C compiler. This eliminates an always-NULL, superfluous field.

For user-facing functions, we apply a similar design pattern and ensure that no "dummy" arguments are ever offered in the public API: such arguments cause user confusion, make code reviews more difficult, and generally diminish trust in our API. Anticipating slightly, consider this initialization function that we present as part of our user-facing API (§IV):

```
let session_p_create (idc: idconfig) (initiator: bool) ...
  (dvp: device_p idc) (peer_id: opt_pid_t idc initiator): ST ... = ...
```

As mentioned in §II, we may not immediately know a peer's identity: whether peer_id is needed at initialization-time depends on the protocol. Rather than rely on an implicit invariant that peer_id will be ignored for some patterns, we instead rely on a generic type opt_pid_t. In the case of XX, the type opt_pid_t becomes unit. In the case of IKpsk2 for the initiator, the type becomes lbuffer uint8. KreMLin guarantees that function arguments of type unit are eliminated: this means we offer a custom API for each Noise protocol. This directly supports our goal of generating robust user-facing APIs that leave no room for user error.

IV. A Complete Verified Noise Library Stack & API

Section III describes the core handshake actions, as captured by the Noise Protocol Framework. Yet, this forms only a small, core part of a Noise library. We now review the remainder of our Noise Protocol implementation and describe the many APIs and library features we wrote in order to provide a complete, self-contained, user-proof, verified Noise protocol stack.

A Generic State Machine. The core handshake actions (§III) each implement a single line of a Noise Pattern. We now tie together these individual protocol actions into two

state machines: one for the initiator and the responder. These basic state machines are trivially induced by the steps of the handshake: they are linear, and each valid transition advances the initiator or responder to their next step.

The send_message_tokens function from §III takes many run-time parameters; we group them in a single type definition, dubbed state_t. The state also holds the current step in the handshake, i.e., the current state of the machine. Continuing with hybrid embeddings, a generic function state_t_handshake_write advances the state machine, and returns a fresh state_t, for any choice of pattern, step i, or initiator *vs.* responder.

```
(* The low-level state machine type: encapsulates keypair, chaining
   hash state, symmetric state, current handshake step, psk, etc. *)
val state_t: isconfig → initiator:bool → Type0
(* Simplified signature *)
val state_t_handshake_write (isc: isconfig) (ssi: static_info)
  (i: nat { i < isc.pattern.messages })
  (payload_len: size_t) (payload: lbuffer uint8)
  (st: state_t isc (i%2=0) { ... })
  (outlen: size_t) (out: lbuffer uint8):
  Stack (s_result_code (st:state_t isc (i%2=0) { ... }))
```

The signature of the function is familiar; the earlier iconfig is now wrapped in an "implementation state config" isc, which contains the entire noise pattern, along with compile-time parameters that determine the shape of the final C struct (§III-D). The function is once again written in the hybrid embedding style; the compile-time parameter i allows the caller to specialize the function for the i-th step of the handshake; this in turns allows us to compute, at compile-time, whether the message originates from the initiator (i%2=0) or the responder (i%2=1). The compile-time parameters also determine the nature of the return type, which is derived from the series of return types for each token. The function *returns* a fresh state st1 under the successful Res case. In a fashion similar to send_message_tokens, the low-level stateful function coincides with the outcome st1'_v of the spec-level interpreter. (Full definitions can be found in [33].)

The parameter i represents the current step of the handshake at compile-time; but this information is also carried at run-time within the state st. A static precondition requires the compile-time i to be consistent with the step stored at run-time within st. This key technical trick enables compile-time computations over the step i, which allows us to write a single transition function. The function can be specialized at compile-time for any step i; doing so produces a Low* function that can only be called when the current run-time step coincides with the compile-time i.

Equipped with this extremely generic function, we now use the hybrid embedding style to generically program state machine management: at compile-time, we generate a series of run-time tests for each (statically-known) possible state of the handshake; if a run-time test succeeds, the code proceeds to execute state_t_handshake_write, specialized at compile-time for the specific step of the handshake. The result is a higher-level function that can generate the state machine of either the initiator or the responder, for *any* Noise pattern. We have effectively embedded at compile-time within F* a compiler that, from a deeply embedded Noise pattern, generates the corresponding shallowly-embedded Low* state machine.

A User-Proof State Machine. As it stands, the state machine cannot be exposed to the user. First, it returns a new state, rather than modifying a heap-allocated state through a pointer; second, it does not record stuck states, meaning that the user can make a mistake by ignoring the Failure and calling the function a second time.

We now transform this low-level state machine into a user-proof one. In the process, we also enrich the API with features for device, peer and key management. We dub this second API layer the "device API". We encapsulate the earlier state_t in a device state dstate_t, which handles Low* region-based memory management and ownership (elided), holds session and peer names (provided by the user), and maintains a device state for peer management.

```
[@CAbstractStruct] type dstate_t idc =
  | Initiator: state:state_t idc.isc true → session_name:name_t
    → peer_name: name_t → device: device_t → ...
    → dstate_t ...
  | Responder: state:state_t idc.isc false → (* similar *)
type dstate_p ... = B.pointer_or_null dstate_t
```

Introducing dstate_p, a potentially-null pointer, serves several purposes: the C code becomes more idiomatic, now manipulating a pointer to a structure instead of passing structures by value; we can now have a NULL case which accounts for errors, e.g., a point at infinity showing up at initialization time; and we can introduce a modicum of abstraction, by using the CAbstractStruct keyword which instructs KreMLin to only emit a typedef in the generated header, thus preventing clients from directly allocating or accessing a dstate_t.

Unlike state_t_handshake_write, this state machine from the device layer is safe to use from C. If an error happens during the handshake, we modify the step number to a special value that indicates that the machine is stuck, before returning an error. Any further attempt to use this state will leave the machine in the error (stuck) state.

Device API and Session Management. In addition to the state machine, the device state dstate_t also encapsulates device management. A device holds a set of peers, along with a table that indexes them by a unique (integer) identifier; it also holds the local static identity, and provides a high-level API which enables the user to add, lookup, update or remove peers. Each peer contains detailed information, such as their remote static and pre-shared keys. The library is written from scratch, since the existing Low* libraries for e.g., linked lists were proof-of-concept-quality and not intended to be used within a large development. The result is a relatively simple API, wherein the user provides a private key, an implementation-specific prologue and a C string for the device name.

```
device_t *device_create(uint32_t prologue_len,
  uint8_t *prologue, const char *name, uint8_t *spriv);
peer_t *device_add_peer(device_t *dvp,
  const char *name, uint8_t *rs, uint8_t *psk);
```

Given a device, the user can create a new *session* with a chosen peer, in the role of either the initiator or the responder.

```
session_t *create_IKpsk2_initiator(device_t *d, uint32_t peer_id);
session_t *create_IKpsk2_responder(device_t *d);
```

We mention at the end of §II that different Noise protocols handle identity management very differently; and that mishandlings can lead to serious vulnerabilities. We rule out these errors by construction in our API, using hybrid embeddings (§III-D) to generically program the signature of the API functions. For instance, IKpsk2 demands a peer identity at initiator-creation time; this is reflected by the presence of the peer_id (a unique identifier)argument above. Conversely, for XX, both parties learn the remote's identity during the handshake, and the peer_id argument is absent from the C function signature.

This in turn begs the question of what should be an acceptable *policy* to deal with receiving a peer's public static key over the network, when the key is currently unknown to the device. The answer varies, and generally requires application-specific error handling. For instance, in the case of WireGuard, an unknown user simply cannot connect and the handshake is aborted. For WhatsApp, conversely, the application registers the peer with the device, and proceeds with the conversation.

In Noise*, we delegate these decisions to the user of our library via a *policy function* and a *certification function*. The former is a constant in practice, and simply determines whether unknown keys may be accepted. The latter receives the decrypted payload of the message which should contain a certificate for the key, and from it determines whether to certify or invalidate a key. This behavior is triggered upon receiving an S token without a corresponding entry in the peer table.

Long-Term Key Storage. To make sure our library is self-contained and ready to be used, Noise* incorporates a verified long-term (e.g., on-disk) key storage feature. Concretely, the device state can be serialized and deserialized, which includes peer list and static key. We use an AEAD construction, with the device and peer names as authenticated data. In order to avoid nonce reuse, each serialization generates a fresh nonce to be fed into the AEAD construction; the nonce is stored on disk, so that it can be reloaded at decryption-time. Our implementation comes with proofs of correctness for the parser and serializer, namely that they are the inverse of each other. Whether on-disk storage is enabled is up to the user; should they enable it via a compile-time parameter, the resulting C code will contain, among other things, a create_device_from_secret that takes an encryption key, encrypted data, and returns a fresh device (or NULL if decryption failed). We delegate the handling of the on-disk encryption key to the user of our library.

A High-Level API with Message Encapsulation. To provide an industrial-grade, error-proof Noise library, there remains one last issue to address: right now, the user might inadvertently send messages at a lower level of confidentiality or authenticity than intended. This may happen either because the user has misunderstood the guarantees provided by a given Noise pattern; or because they sent early data in the handshake, before the full guarantees were established (Figure 2).

We revisit the Noise confidentiality levels (Figure 2) and expose an informative subset of them to the user: "public" (C0), "known remote replayable" (C2), "known remote weak forward" (C3) and "known remote strong forward" (C5). Then, we abstract away the type of messages and impose that the user go through a constructor and a destructor. These not only require the user to specify a level, but also to commit to a session and a peer, which rules out improper handling of data.

```
encap_message_t *pack_with_conf_level(
  uint8_t requested_conf_level,
  const char *session_name, const char *peer_name,
  uint32_t msg_len, uint8_t *msg);
bool unpack_message_with_auth_level(
  uint32_t *out_msg_len, uint8_t **out_msg,
  char *session_name, char *peer_name,
  uint8_t requested_auth_level, encap_message_t *emp);
```

Encapsulated messages can then be sent through an API that wraps handshake_write and takes care of packing and unpacking. When sending, we check that the session sn has reached *at least* the desired confidentiality level; when receiving, we check that the requested authentication level is *at most* the session's current level. The high-level rcode captures both state machine errors (stuck), and authenticity or confidentiality errors.

```
rcode session_write(encap_message_t *input, session_t *sn,
  uint32_t *out_len, uint8_t **out);
rcode session_read(encap_message_t **out, session_t *sn,
  uint32_t *inlen, uint8_t *input);
```

This concludes our tour of our Noise protocol implementation. From the protocol actions of §III, we derived a state machine implementation that properly handles failures and is generically programmed. We extend this state machine with runtime support for peer and device management, peer authentication policies, and on-disk long-term key storage. We expose the API via safe functions that perform confidentiality and authenticity run-time checks at the API boundary to rule out errors from unverified C clients. We obtain the first verified implementation for a full secure channel protocol stack, complete from cryptographic primitive to its user-facing API.

V. Symbolic Security Proofs for Noise*

As explained in Section II, the Noise specification [20] describes the expected security guarantees for each Noise protocol in terms of authentication (A0-A2) and confidentiality levels (C0-C5). Several analyses have shown that various Noise protocols meet these guarantees against classic Dolev-Yao-style active network adversaries [34], using symbolic analysis tools like ProVerif [22] and Tamarin [21]. Although these analyses provide comprehensive results for the protocol messaging code, they do not cover important details like message formats, protocol state machines, or key management, which are crucial to the security of full Noise implementations. In this section, we close this gap by proving the symbolic security of our F* Noise specification, relying on a framework called DY* [27].

A. Background on DY*

DY* Framework. DY* is a set of F* libraries that enables the symbolic security verification of protocol code written in F* [27]. In effect, we take our Noise protocol specification from Section II and replace all calls to concrete cryptography, random number generation, and state storage with the symbolic libraries provided by DY*, to obtain a *symbolic security specification* in F* that is functionally equivalent to our original specification. We then use the proof patterns provided by DY* to prove that our specification satisfies the security guarantees expected by Noise. Our proofs account for an unbounded number of protocol sessions and an active Dolev-Yao adversary [34].

DY* has previously been used to verify various protocols (including Signal [27]) but a key novelty of our approach is that we build a generic security proof for a Noise protocol interpreter to obtain security guarantees for *all* Noise protocols in one go. This kind of parameterized inductive proof is out of reach of tools like ProVerif and Tamarin, which instead have to rely on per-instance verification of each Noise protocol [22], [21]. The trade-off is that DY* is not as automated as these tools, and it does not yet support the verification of equivalence properties, needed to state goals like identity privacy.

We refer the reader to the DY* paper [27] and public code repository [35] for its detailed presentation. Below, we briefly discuss the main elements used in our Noise security proof.

Trace-Based Semantics. A DY* program consists of a set of stateful protocol functions (e.g., session_create, handshake_write) that can be executed by each protocol participant or *principal* (e.g., "alice","bob") to initiate or continue any number of protocol sessions, where each session is locally identified by an integer sid. Each principal can store session-specific state as well as long-term state shared between sessions.

The interleaved distributed execution of protocol sessions across multiple principals is modeled by an append-only global trace that records every message sent between principals, every freshly generated random value, every long-term and session state stored by each principal, and every security event triggered by a principal to mark the progress of a protocol session. The index of an entry in the global trace can be seen as a unique immutable timestamp, so we can state for example, that an event was triggered at a particular trace index (event_at i (Send A B M)) and that this occurred *before* another event (event_at j (Recv B A M) $\wedge$ i $<$ j).

For example, in a run of the Noise IKpsk2 protocol between I and R, after I sends the first message, the global trace contains an entry for the generation of I's ephemeral key (x), the message from I to R, and I's handshake state after this message. Once R processes the first message and responds with the second message, the trace is extended by another entry for the responder ephemeral, the second message, and the handshake state stored at R. When the handshake is complete, both parties discard their session-specific handshake states and store new session states containing the final cipher states.

The attacker is modeled as an F* program that acts as a global scheduler: it drives the execution of all protocol sessions by calling protocol functions at different principals. It has all the capabilities of an active network attacker: it can read and write messages between any two principals in the global trace, it can generate its own fresh random values, and it can call cryptographic functions using values it has learned. The attacker can also dynamically compromise any state stored at any principal to obtain its contents. Hence, by compromising the long-term state at a principal (indicated by the event corrupt_principal i "alice"), the attacker can learn the principal's static Diffie-Hellman and pre-shared keys. Alternatively, by compromising a session state corresponding to an ongoing Noise session at a principal (corrupt_session i "alice" sid), the attacker can learn the current handshake state, including any private ephemeral keys. However, the attacker cannot guess random values, or invert encryption unless it either has the key or has explicitly compromised it. The attacker's knowledge at a particular timestamp in the global trace is formalized by an inductive predicate: attacker_knows_at i m.

B. Formalizing Payload Security Goals as Trace Properties

We formalize each of the 3 authentication levels (A0-A2) and 6 confidentiality levels (C0-C5) of Noise as *trace properties*, i.e., predicates over the global trace.

Authentication Goals. Level A0 provides no guarantees.

```
let trace_property_A0 = ⊤
```

For A1, suppose that before sending an authenticated payload, each Noise participant A triggers an event of the form AuthSent A B M L indicating that it is sending a message M to B at authentication level L. After successfully processing an authenticated payload M in a session sid, the recipient B triggers an event AuthReceived B sid A M L. Then, the authentication goal for messages sent at Noise authentication level A1 can be written as a trace property:

```
let trace_property_A1 =                                                    1
    ∀i sid A B M. event_at i (AuthReceived B sid A M 1) ⟹                  2
          (∃ j. j < i ∧ event_at j (AuthSent A B M 1)) ∨                   3
          (∃ k. k < i ∧ (corrupt_principal k A ∨                           4
                         corrupt_session k B sid ∨                         5
                         corrupt_principal k B))                           6
```

This trace property says that whenever B accepts a message M from A at time i (with authentication level A1), either this must be an authentic message sent by A at time $j < i$, or else the long-term state of A, the session state at B or the long-term state of B must have been compromised before i. Note that corrupt_session k B sid actually implies corrupt_principal k B, so the conjunct on line 5 is actually redundant: we leave it only to make it clear that the trace_property_A1 above implies the trace_property_A2 below.

The disjunct on line 6 indicates the possibility of a KCI attack: i.e., the loss of message authenticity when the recipient B's static key is compromised.

To obtain the trace property for authentication level A2, we simply remove this disjunct (line 6) to require the absence of KCI attacks:

```
let trace_property_A2 =
    ∀i sid A B M. event_at i (AuthReceived B sid A M 2) ⟹
        (∃ j. j < i ∧ event_at j (AuthSent A B M 2)) ∨
        (∃ k. k < i ∧ (corrupt_principal k A ∨
                       corrupt_session k B sid)
```

Level A0 provides no guarantees:

```
let trace_property_A0 = ⊤
```

Confidentiality Goals. Confidentiality guarantees are stated as predicates over the global trace that describe the conditions in which a protocol secret may become part of the attacker's knowledge. Suppose that each Noise participant A triggers an event ConfSent A sid B sid' M L before sending a fresh random secret message M at confidentiality level L to B, where sid and sid' are the session indexes at A and B (note that sid' is retrieved from B's ephemeral key by mean of a ghost function - an abstract function that can only be used in specifications). Then, the confidentiality level C4 is written as the following:

```
1 let trace_property_C4 =
2     ∀i j sid sid' A B M.
3         (event_at i ConfSent A sid B sid' M 4 ∧
4          attacker_knows_at j M ∧ i ≤ j) ⟹
5         (∃ k. k < i ∧ (corrupt_principal k A ∨
6                        corrupt_principal k B)) ∨
7         (∃ l. l ≤ j ∧ (corrupt_session l A sid ∨
8                        corrupt_session l B sid')
```

This predicate says that if a secret message M sent at time i (and confidentiality level C4) from a session sid at A to a session sid' at B, and M subsequently becomes known to the adversary at time j, then either the static key of A or the static key of B was compromised before the message was sent at i or else one of the two ephemeral protocol session states (sid, sid') was compromised before j.

The strongest variant of forward secrecy provided by Noise (C5) limits static key compromise to the recipient; that is, we drop the disjunct on line 5 (corrupt_principal k A) allowing the sender A's static key to be compromised at any time without affecting the confidentiality of M:

```
let trace_property_C5 =
    ∀i j sid sid' A B M.
        (event_at i ConfSent A sid B sid' M 5 ∧
         attacker_knows_at j M ∧ i ≤ j) ⟹
        (∃ k. k < i ∧ corrupt_principal k B) ∨
        (∃ l. l ≤ j ∧ (corrupt_session l A sid ∨
                       corrupt_session l B sid')
```

The trace properties for levels C1-C3 provide weaker forward secrecy guarantees than C4 by restricting the compromise scenarios in which confidentiality is guaranteed. In these scenarios, the sender does not know if the peer ephemeral public key it is using actually belongs to some recipient session sid' of B; instead this public key may have been provided by the attacker. So we use a different event ConfSentEph A sid B eph M L, where instead of the peer session sid', A marks the (possibly attacker-controlled) peer ephemeral key which it used to derive the encryption key.

The confidentiality guarantee of C1 then states that the message is secret only if this peer ephemeral key is confidential:

```
let trace_property_C1 =
    ∀i j sid eph A B M.
        (event_at i ConfSentEph A sid B eph M 1 ∧
         attacker_knows_at j M ∧ i ≤ j) ⟹
        (∃ l. k ≤ j ∧ (corrupt_session k A sid ∨
                       (∃ sk. eph = PK(sk) ∧
                              attacker_knows_at k sk)))
```

That is, we have no confidentiality if the attacker actively interferes with the session to provide its own public key eph. Note that this means that confidentiality is lost even if none of the recipient B's keys have been compromised.

C2 provides a stronger guarantee that links the confidentiality of the message to static key compromise at the recipient B:

```
let trace_property_C2 =
    ∀i j sid eph A B M.
        (event_at i ConfSentEph A sid B eph M 2 ∧
         attacker_knows_at j M ∧ i ≤ j) ⟹
        (∃ k. k ≤ j ∧ (corrupt_session k A sid ∨
                       corrupt_principal k B))
```

That is, we have no confidentiality if the attacker compromises the recipient's static key or the sender's ephemeral key (before or after the message is sent), but the compromise of the sender's static key does not affect security.

C3 provides weak forward secrecy, which combines the guarantees of C1 and C2 to link message confidentiality to both the peer's ephemeral key and recipient's static key:

```
let trace_property_C3 =
    ∀i j sid eph A B M.
        (event_at i ConfSentEph A sid B eph M 3 ∧
         attacker_knows_at j M ∧ i ≤ j) ⟹
        (∃ k. k ≤ j ∧ (corrupt_session k A sid ∨
                       (∃ sk. eph = PK(sk) ∧
                              attacker_knows_at k sk ∧
                              corrupt_principal k B)))
```

That is, we have no confidentiality if the attacker first actively interferes with the session to provide its own ephemeral public key and then also compromises the recipient's static key (before or after the protocol message is sent).

C0 provides no guarantees:

```
let trace_property_C0 = ⊤
```

Deriving Security Goals for each Noise Protocol. The overall security goal for our Noise specification is to prove that every global execution trace for every Noise protocol satisfies the 7 trace properties corresponding to A1-A2 and C1-C5. Hence, for each payload in a Noise protocol, we can look up the confidentiality and authentication level (from Appendix A) and map it to the corresponding trace property to obtain the precise security guarantee at sender and recipient.

Our way of encoding security goals as trace properties (sometimes called correspondence assertions [36]) is similar to how these goals are usually stated in protocol verification tools like ProVerif and Tamarin. Notably, these trace properties

are defined independently of a specific Noise protocol or its F* code and only refer to events triggered during protocol execution. This allows our security goals to be independently audited and compared with other formulations. Indeed, the corresponding ProVerif query for authentication level A2 in prior work [22] (see Section II-C) is almost identical (modulo syntax) to our trace property. However, the ProVerif queries for forward secrecy (C2-C5) look different from our trace properties since they use phases (instead of timestamps) to enforce an order between messages and compromise events.

C. Security Proof for Noise*: Overview

Having stated our (trusted) security goals by mapping *levels* to trace properties, the next step is to prove that our security-oriented specification preserves a global trace invariant that implies these trace properties. This symbolic security proof in DY* relies on two kinds of (untrusted) annotations: secrecy labels and authentication predicates. These must be provided by the programmer and are then verified by typechecking. Note that *labels* are not the same as the *levels* previously introduced: *levels* are a way of revealing high-level security properties to the user and are used by the API to perform dynamic checks, while *labels* are annotations we insert for the security proofs.

Secrecy Labels. Each bytestring (key, message, constant) used in the protocol must be annotated with a *secrecy label* that indicates which *sessions* of which *principals* are allowed to read them. For example, a static (long-term) Diffie-Hellman private key belonging to a principal named "alice" is given a label CanRead [P "alice"], indicating that it can be read by all sessions of "alice", whereas an ephemeral private key that is only meant to be used in session sid is labeled CanRead [S "alice" sid]. A long-term pre-shared key between "alice" and "bob" is given the label CanRead [P "alice"] ⊔ CanRead [P "bob"], where the join (⊔) operator indicates the union of the two labels. For succinctness, we can also write the above label as CanRead [P "alice"; P "bob"]. Constants and public bytestrings are labeled with Public, indicating that they can be read by any session, including by the attacker.

Secrecy labels are related by a reflexive, transitive relation can_flow i l1 l2 which says that a label l2 is stronger (more restrictive) than label l1 at a timestamp i. For example, the label Public can always flow to any other label, and CanRead [P p; P p'] can always flow to CanRead [P p]; but CanRead [S p sid] can only flow to Public at timestamp i if the event Compromise p sid occurs before i in the global trace.

The DY* cryptographic API manipulates these labels and imposes a strict discipline on their usage, ensuring that secret data never flows to a public location. In particular, AEAD encryption returns a ciphertext labeled Public, but requires as a pre-condition that the label of the payload must flow to the label of the key. Computing a Diffie-Hellman shared secret between two private keys with labels l and l' yields a key with label $l \sqcup l'$, indicating that any session that can read one of the two private keys can know the shared secret. Calling a key derivation function (KDF) with two keys with labels l and l' yields a key with label $l \sqcap l'$, where the meet ($\sqcap$) operator indicates an intersection; only sessions that can read both inputs may read the result. Hence, KDF strengthens the label of a key by mixing in additional key material.

As a consequence of the secret labeling discipline, DY* provides a generic secrecy lemma stating that a secret with label l can only be obtained by the adversary at timestamp i if can_flow i l Public holds. This lemma can be instantiated to obtain strong protocol-specific security guarantees. For example, a Diffie-Hellman shared secret x with label CanRead[S "alice" sid] ⊔ CanRead[S "bob" sid'] is *forward secret*: an attacker can only obtain it if it specifically compromises sid (at alice) or sid' (at bob) before these sessions end and their state is deleted. Notably, compromising the long-term keys of alice or bob, or any other sessions at these principals does not help the adversary obtain x. Labels like these allow us to prove trace properties like strong forward secrecy (C5).

Authentication Predicates. DY* also defines a set of authentication predicates that can be instantiated for each protocol to enable the propagation of security invariants through cryptographic calls and events. For example, AEAD encryption has a pre-condition ae_pred that is intended to specify the conditions under which a message is allowed to be encrypted; this predicate becomes a post-condition for AEAD decryption. For Noise, we instantiate ae_pred to require that, either the encryption key is compromised, or the sender (who is thus honest) must have triggered the AuthSent and ConfSent events, and consequently obtain the corresponding authentication guarantee at the recipient. Similarly, an event predicate event_pred states when an event may be triggered; we instantiate it to encode our authentication goals, requiring that the event AuthReceived can only be triggered if the corresponding authentication property holds. By instantiating these predicates and verifying that our protocol code still satisfies the resulting preconditions, we link protocol session state invariants with cryptographic guarantees to prove the target trace invariants for our Noise specification.

Structure of the Proof. We structure the symbolic security proof for our Noise specification in several steps:

- **Security Levels to Trace Invariants**: we write a generic function that maps any step of any Noise pattern to its corresponding *level*, as described in Figure 2, the full version of which is in Appendix A. We then extend the global trace invariant with the corresponding authentication or confidentiality trace properties for every Noise message sent and received at each level.
- **Security Levels to Key Secrecy Labels**: we map each payload security level to a predicate over the *secrecy label* of the AEAD key used to encrypt the payload. We show that, for each confidentiality and authentication level, the AEAD key secrecy label, the properties of AEAD encryption, and the generic secrecy lemmas of DY* together imply the global trace invariant.
- **Handshake State Invariant**: to each state of the handshake, we associate a label and we prove that in all runs of the protocol code, the resulting state matches its target

label. We then prove that the label of the handshake state at a given protocol stage is always stronger than the target key secrecy label for that stage of the protocol.
- **High-Level API security**: our high-level API always preserves the handshake state invariant. In combination with the above sequence of proof steps, this allows us to prove that all reachable traces of our Noise protocol specification satisfy the level-based authentication and confidentiality guarantees of Noise. In particular, we prove that these security guarantees are correctly propagated all the way up to the user-facing API where they are exposed as understandable security guarantees.

To achieve the proof above, we build a new security-oriented specification of Noise that is provably equivalent to our original specification, but is annotated with labels and logical invariants that enable us to prove our security goals. The full proof development is in F*; we now describe each of the proof steps.

D. Security Proof: Handshake State Invariant

Labeling the Handshake State. In our security spec, we annotate every element of the handshake state with a secrecy label. The cipher_state and symmetric_state types are now parameterized by a timestamp i and a label l for the chaining key ck and AEAD key k:

```
type cipher_state (i:nat) (l:label) = {
  k: option (aead_key i l);
  n: nat;}
type symmetric_state (cfg:config) (i:nat) (l:label) = {
  h : hash cfg i Public;
  ck : chaining_key cfg i l;
  c_state : cipher_state i l}
```

The full handshake state for a session sid at a protocol participant p is annotated with a security index. For each participating principal in the protocol, the index includes the name of the principal (p), its local session identifier (sid), the name of the peer (peer), and the secrecy label associated with the ephemeral key of the peer (peer_eph_label). Of these, the last two are optional, since they may only be available in later stages of protocols.

```
type index = {
  p: principal;
  sid: nat;
  peer: option principal;
  peer_eph_label: option label}
```

Notably, the index does not contain the peer's local session identifier, since this value is unknown to p. All p knows is the remote ephemeral public key, and so we state our security properties in terms of what p knows about the security of this key, which is encapsulated in peer_eph_label.

Each handshake state is annotated with the current index idx and the current label l encoding the secrecy of the current chaining key and cipher state. Hence, in each run of a protocol at a principal p, we have an index and a label describing the current security guarantees.

```
type handshake_state (cfg:config) (i:nat) (l:label) (idx:index) = {
 sym_state : symmetric_state nc i l;
 static : option (keypair cfg i (CanRead [P idx.p]));
 ephemeral : option (keypair cfg i (CanRead [S idx.p idx.sid]));
 remote_static : option (public_key cfg i (CanRead [P idx.peer]));
 remote_ephemeral : option (public_key cfg i idx.peer_eph_label);
 preshared : option (preshared_key cfg i idx.p idx.peer);}
```

In the handshake state, the local static and ephemeral keypairs have secrecy labels related to the current principal and session. Once we have validated the remote static key (see the certification function below), it is labeled with CanRead [P idx.peer]. However, the relationship between the remote ephemeral key label (idx.peer_eph_label) and the peer's identity is unknown. The pre-shared key, if it exists, has a label indicating that it is shared between the principal and its peer.

Computing Target Secrecy Labels. Given a Noise protocol (described as a handshake_pattern), and an index describing the current run, we can compute the target secrecy label for the handshake state at the initiator and responder at each stage of the protocol. Note that since the initiator and responder have different (partial) views of their peer's protocol state, the computed labels at the two ends may be different. In total, we compute four labels at each stage, two for‘ the initiator, and two for the responder:

- l_i: the current label at I;
- $l_i^{\leftarrow}$: the last label at which I received a message from R;
- l_r: the current label at R;
- $l_r^{\rightarrow}$: the last label at which R received a message from I.

The last labels at which I or R received a message are used for authentication through the peer ephemeral invariant: receiving a message from a peer gives you confirmation that he was able to advance in the protocol up to this message, providing guarantees about the secrecy of his ephemeral key (see **Establishing the Peer Ephemeral Invariant**). When we wish to refer to the label at a particular stage n, we write $l_i[n]$ or $l_r[n]$. The sequence of computed labels for our three example Noise protocols X, NX, and IKpsk2 are shown in Figure 3, the full version of which is in Appendix B.

The target label computation faithfully follows the sequence of cryptographic operations. Every time new key material is added to the handshake state, the new label is a meet (or $\sqcap$) of the old label and the new key material. If the key material is a Diffie-Hellman secret, its label is a join (or $\sqcup$) of the labels of the two Diffie-Hellman private keys. Each participant knows the labels of its own static key (CanRead [P idx.p]) and its own ephemeral key (CanRead [S idx.p idx.sid]). After public key validation, it also knows the label of the peer's static key (CanRead [P idx.peer]), but it typically does not know the label of the peer's ephemeral key (idx.peer_eph_label). Hence, Diffie-Hellman operations involving the peer's ephemeral key result in labels that use idx.peer_eph_label as an opaque label.

Computing Target Labels for X. The protocol X has a single message with four tokens. At the initiator point of view, the token e does not affect the label; es changes the label to the

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$
X	$\leftarrow$ s	pre				
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p]$\sqcup$$\mathsf{idx}_r$.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$
	$\rightarrow$ $[d_1, d_2, \ldots]$	2	$= l_i[1]$	Public	$= l_r[1]$	$= l_r[1]$
NX	$\rightarrow$ e $[d_0]$	1	Public	-	Public	Public
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p]$\sqcup$$\mathsf{idx}_r$.peer_eph_label)	Public
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$
	$\leftrightarrow$ $[d_3, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$
IKpsk2	$\leftarrow$ s	pre				
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$
	$\leftarrow$ e, ee, se, psk $[d_1]$	2	$l_i[1] \sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p]$\sqcup$$\mathsf{idx}_i$.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[1] \sqcap$ (CanRead [S idx_r.p idx_r.sid]$\sqcup$$\mathsf{idx}_r$.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$

Fig. 3. Target Security Labels Computed for Three Example Noise protocols (X, NX, and IKpsk2)

secrecy label of the ephemeral-static Diffie-Hellman shared secret (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]); s does not affect the label; ss changes the label to the meet of the previous label and the label of the static-static Diffie-Hellman shared secret (CanRead [P idx_i.p; P idx_i.peer]). Hence the label l_i after the first message is a meet of the labels of the two Diffie-Hellman shared secrets.

From the responder's point of view, the label l_r looks a bit different. Since the responder does not know the label of the initiator's ephemeral key, the label it computes for the ephemeral-static shared secret is of the form (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label), where idx_r.peer_eph_label is the label of the peer's ephemeral private key. The label for the static-static shared secret is the same. Hence, for the responder, the key after the first message is only partially authenticated (level 1 in Noise terminology)

The last received label $l_i^{\leftarrow}$ is null since the initiator has not received any message, and $l_r^{\rightarrow}$ is the same as l_r.

Computing Target Labels for NX. For NX, the label computation is similar, except that the labels at the initiator and responder are even more asymmetric, since the initiator is unauthenticated. Hence, at the end of the protocol, the initiator has a precise label linking its session to the peer's identity (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]), but the responder only has a weak label linking its session to *some* (potentially compromised) peer ephemeral key (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label).

Computing Target Labels for IKpsk2. The computation of labels for IKpsk2 follows the same pattern as X and NX except that both parties are authenticated and their labels get stronger with each stage. Notably, at the end of the second message, the responder's label $l_r[2]$ has reached the maximum label for this pattern (it never changes thereafter). However, at this point, the last received label $l_r^{\rightarrow}[2]$ is still quite weak (since R has not yet received a message protected under the newest key, confirming that I was able to complete the handshake). It is only when the responder receives a subsequent (data) message from the initiator that the two labels l_r an $l_r^{\rightarrow}$ coincide. It is this quirk of IKpsk2 that leads to the responder obtaining a slightly weaker forward secrecy guarantee (Noise level 4) at the end of the second message, and strong forward secrecy (level 5) after the third message.

Hence, for instance, after the second IKpsk2 message, the target handshake state label at an initiator with index idx_i is computed as follows:

```
(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓
(CanRead [P idx_i.p; P idx_i.peer]) ⊓
(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label]) ⊓
(CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label]) ⊓
(CanRead [P idx_i.p; P idx_i.peer])
```

Each line of the label corresponds to some key material that has been mixed into the chaining key: ephemeral-static, static-static, ephemeral-ephemeral, and static-ephemeral Diffie-Hellman secrets, followed by a pre-shared key.

Proving the Handshake Secrecy Invariant. Our main secrecy invariant for the handshake state is that at each stage of the protocol its label must match the computed target label. We prove that the messaging functions in our Noise specification preserve this invariant whenever they modify the handshake state. For example, the type of our labeled send_message_tokens function is as follows:

```
val send_message_tokens (cfg:config) (initiator is_psk:bool)
  (tokens:list token) (i:nat) (l:label) (idx:index)
  (st:handshake_state cfg i l idx) :
  (result (ciphertext:msg i Public &
   handshake_state cfg i (updt_label l idx tokens initiator) idx))
```

The result type says that the new handshake state label (after the message is sent) can be computed from the old label, the index, the list of sent tokens, and the message direction. Separately, we show that this updated label corresponds exactly to the target label computed for this stage of the handshake pattern.

The type for receive_message_tokens is a bit more complicated since the index of the handshake state may change in the course of the function, if the message contains the peer's static or ephemeral key. Other than this detail, we again prove that it updates the handshake label in the same way from the prior label and received tokens. Hence, we prove that all our messaging functions preserve the handshake labeling invariant.

Establishing the Peer Ephemeral Invariant. The label of peer ephemeral key (idx.peer_eph_label) in the handshake state is (as yet) unrelated to the peer's identity. It means that the keys in the handshake state are linked to an untrusted remote ephemeral key, and hence are not forward secret. To obtain stronger forward secrecy guarantees, we need to establish an authentication invariant on the handshake state.

As described above, in addition to the target secrecy labels (l_i, l_r) for each handshake state at the initiator and responder, we also keep track of the label at which each participant received its last message ($l_i^{\leftarrow}$, $l_r^{\rightarrow}$). We then prove that if this last receive label is uncompromised at i (i.e., it does not flow to Public) then the remote ephemeral key label at i (idx.peer_eph_label) must be of the form CanRead [S idx.peer sid'] for some session sid' at the peer. In other words, the last received message conditionally attests to the authenticity of the peer ephemeral key. If the payload received with this message was protected with a strong label, we get a strong authentication guarantee for the peer ephemeral.

To obtain an authentication guarantee for the peer ephemeral key, we rely on the global AEAD predicate (ae_pred, mentioned in §V) to enforce that every encrypted handshake payload sent in each direction contains a transcript hash in the associated data, which uniquely captures all the ephemeral keys exchanged so far. Using this AEAD predicate at each decryption, the receive_message functions can establish and maintain the peer ephemeral invariant in the recipient's handshake state.

E. Security Proof: Handshake State Invariant to Trace Properties

Our next goal is to show that the handshake state invariant implies the trace properties corresponding to our authentication (A0-A2) and confidentiality goals (C0-C5). This proof is in three steps: (1) we map each authentication and confidentiality level to predicates on the secrecy label of an AEAD key; (2) we show that the handshake state invariant guarantees that the current AEAD key in the handshake state satisfies these key secrecy predicates; (3) we show that each key secrecy predicate implies the trace property for the corresponding level.

Mapping Levels to Key Secrecy Predicates. We define a series of security predicates in F*, one for each payload security level, stated in terms of the current global timestamp (i), security index (idx), and a handshake state label (l). The confidentiality predicates should be read from the viewpoint of the sender, whereas the authenticity predicates are from the viewpoint of the recipient. Each predicate has the same shape, represented by the predicate type below:

```
type security_pred = i:nat → idx:index → key_label:label → Type
```

The three authentication predicates are as follows:

Level	Authentication Predicate (over i, idx, and l)
A0	⊤
A1	can_flow i (CanRead [P idx.p; P idx.peer]) l
A2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Each authentication predicate is stated in terms of the strength of the current key label l; that is, the conditions under which the current key may be known to the adversary. This in turn implies the conditions under which the messages received by idx.p may have been forged or tampered with.

For level A0, there are no authentication guarantees, and so the predicate is always ⊤.

For level A1, we require that l is at least as strong as the (static-static) label CanRead [P idx.p; P idx.peer], which means that the current key can only be known to the adversary if one of the two static keys (at idx.p or idx.peer) were currently known to the adversary.

For level A2, we strengthen the requirement by requiring that l is at least as strong as the (ephemeral-static) label CanRead [S idx.p idx.sid; P idx.peer]. This predicate requires that the AEAD key in the cipher state should be known only to the principal (idx.p) and its peer (idx.peer), and should be bound to the current session sid at idx.p. So, even an adversary who compromises a principal's static key cannot obtain the session key; the adversary must compromise either the principal's ephemeral key or the peer's static key. In particular, this forbids KCI attacks, since compromising the long-term keys of the principal idx.p does not break authentication.

The six confidentiality predicates are depicted in Figure 4, again stated in terms of the timestamp i, index idx, and the current handshake state label l.

As with authenticity, level C0 provides no guarantees.

For level C1, we require that the handshake state label l is at least as strong as the ephemeral-ephemeral label (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) which means that the recipient (peer) is unauthenticated and hence could be played by an active attacker. This level only protects against passive adversaries. Note that idx.peer_eph_label is actually an optional value, so the predicate definition implicitly says that this value must not be empty, otherwise the confidentiality predicate is false.

For level C2, we require that l is as strong as the ephemeral-static label CanRead [S idx.p idx.sid; P idx.peer], so we have confidentiality unless the sender's ephemeral key or the peer's static key are compromised.

For level C3, our requirement is a little stronger in that we require the current label to be stronger than both the ephemeral-static label (from 2) and the ephemeral-ephemeral label (from 1). This level provides weak forward secrecy, since the attacker can actively interfere with the session to insert its own ephemeral key.

For level C4, we strengthen the forward secrecy guarantee of level 3 by adding conditions under which the peer ephemeral key is known to be secure. Unless the attacker has actively compromised one of the two static keys (before the session is complete), the peer_eph_label must be of the form CanRead [S idx.peer sid'] for some peer session sid'. Hence, we have forward secrecy if both static keys are uncompromised during the session.

Level C5 provides strong forward secrecy: the attacker must compromise either the sender's ephemeral key or the recipient's

Level	Confidentiality Predicate (over i, idx, and l)
C0	$\top$
C1	can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l
C2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
C3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l
C4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l $\wedge$ (compromised_before i (P idx.p) $\vee$ compromised_before i (P idx.peer) $\vee$ ($\exists$sid'. peer_eph_label == CanRead [S idx.peer sid']))
C5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l $\wedge$ (compromised_before i (S idx.p idx.sid) $\vee$ compromised_before i (P idx.peer) $\vee$ ($\exists$sid'. peer_eph_label == CanRead [S idx.peer sid']))

Fig. 4. Confidentiality Predicates for each Noise Confidentiality Level

static key before the session is complete. This predicate is as follows:

```
can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧
can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧
(compromised_before i (S idx.p idx.sid) ∨
compromised_before i (P idx.peer) ∨
   (∃ sid'. peer_eph_label == CanRead [S idx.peer sid']))
```

The first line of this predicate says that the handshake secrets should be readable only by the (authenticated) peer (idx.peer) and the current session idx.sid at idx.p. The second line says that the handshake secrets must also be bound to some peer ephemeral key. The last two lines provide strong forward secrecy: they say that unless the peer's long-term keys and the specific session idx.sid of idx.p was compromised (before the session is complete), the peer ephemeral key must have a label of the form CanRead [S idx.peer sid']. Since the key label is bound to specific sessions at both ends, compromising long-term keys after the session has no effect on key secrecy.

Handshake Invariant to Key Secrecy Predicates. Given the handshake state invariant (including the secrecy invariant and the peer ephemeral invariant), we prove that in each reachable handshake state the current handshake label satisfies the authenticity and confidentiality predicates described above for the security levels at the current stage of the protocol. In other words, we show that the secrecy label annotating the handshake state (and hence the label of its current AEAD key) is always stronger than the label expected by the Noise payload security level at the current stage of the protocol.

Key Secrecy Predicates to Trace Invariants. Message authenticity and confidentiality guarantees in secure channel protocols directly rely on the secrecy of the corresponding encryption key. In DY*, the AEAD encryption function only allows the encryption of a message under a key whose label is stronger than the message, and so the key label expresses an upper bound on the secrecy of messages. For our confidentiality proofs, we assume that the application always sends messages labeled with the current handshake state label, which is the same as the current AEAD key label. Then, from using the confidentiality predicate on key labels for each level, we derive the confidentiality trace invariant for messages sent at this level as a corollary of the generic secrecy lemmas of DY*.

For authentication, we instantiate the ae_pred pre-condition of AEAD encryption to ensure that each call to the AEAD encryption function is preceded by a security event AuthSent with the appropriate parameters. We also instantiate the event_pred pre-condition for security events to ensure that the AuthReceived function can only be called if the corresponding authentication trace property is satisfied. These predicates, along with the properties of AEAD encryption and decryption, and the authentication predicates on the key secrecy label allow us to prove the trace invariant for each authentication level.

This completes the security proof for the protocol code. In summary, we combine the handshake state invariant with key secrecy predicates to show that every reachable handshake state preserves the global trace invariant, which includes the confidentiality and authentication goals for each level.

F. Security Proof: High-Level API security

The final step of our proof involves propagating the protocol security guarantees up through the stack all the way to the high-level API. For most of our code, this propagation is relatively straightforward: we prove that our code does not accidentally break the labeling discipline, by storing a secret value in a public location, or mixing up data from different sessions.

The main security-critical step in this proof is the static key validation function provided by the device API. We assume that the certification function can take a potential public key, along with a (possibly-empty) certificate, and verify that it is indeed a static public key belonging to a given principal:

```
val certification_function: i:nat → rs:bytes → rcert:bytes →
 option (peer:principal{is_public_key rs i (CanRead [P peer])})
```

We also propagate our secrecy labels through the device management API, by annotating all remote static and pre-shared keys stored in the device with the appropriate labels

Component	F* spec	Low* code	DY* proof
Core Protocol (§III)	1,095	15,506	1,792
Device Management (§IV)	315	6,410	475
Session API (§IV)	1,106	13,184	3,681

Fig. 5. Size of the Noise* codebase, *excluding* whitespace and comments. The total size of the codebase is 43kLOC.

Pattern	Noise*	Custom	Cacophony	NoiseExpl.	Noise-C
X	6677	N/A	2272	4955	5603
NX	5385	N/A	2392	4046	5065
XX	3917	N/A	1593	3149	3577
IK	3143	N/A	1357	2459	2822
IKpsk2	3138	3756	1194	2431	N/A

Fig. 6. **Performance Comparison**, in handshakes / second. Benchmark performed on a Dell XPS13 laptop (Intel Core i7-10510U) with Ubuntu 18.04.

and ensuring that these labels are respected by the data structure and by the encrypted storage mechanism.

After all these steps, we obtain a high-level API that guarantees that each application message sent or received with the API meets high-level security properties expressed using a subset of the Noise security levels.

VI. Evaluation and Comparison with Related Work

Size of the Codebase. Figure 5 measures the size of the F* codebase for our Noise protocol implementation. This covers everything described in this paper. The core protocol code contains the Noise messaging functions. Device management includes long-term key storage and validation, including the encrypted storage and verified in-memory data structures, such as a linked list and an imperative map. Session API includes the two successive state machines and the high-level user-facing API code. For each component, we list the size of the high-level specification, the Low* code, and the DY* proof. All of the code listed here was written for the purposes of this paper. The total size is 43kLOC excluding whitespace and comments. As a point of comparison, HACL* itself is 97kLOC, making Noise* the second largest F* project in the literature. All this code is open-sourced [33].

The Compiled C Library. Using the Noise* compiler, we compile several specialized C implementations for each of the 59 Noise protocols. Representative code sizes are: 6,400 lines of C code for IKpsk2, 5,900 LoC for XX, and 4,900 LoC for X. Each Noise Protocol admits several implementations, depending on the choice of primitives (e.g., SHA2-256 vs. Blake2b), and the degree of optimization (e.g., Blake2b-portable vs. Blake2-AVX2). As a proof of concept, we ran a batch job that produced 472 implementations, out of several thousand possible choices [33]; the result totals 3.2M lines of C code.

In practice, a typical user would choose a Noise protocol, a set of primitives and a choice of optimization level, then would download the corresponding C implementation from Noise*, along with a custom distribution of HACL* containing the relevant cryptographic primitives for the target platform, to obtain a small high-performance protocol implementation. Advanced users can extend our code-base and compile it in different ways, to obtain any combination of Noise patterns.

Proof Overhead. A popular way of measuring the human effort of verification is the proof-to-code ratio: how many lines of Low* code did we write for each line of C that we produced. If we were to consider all 59 Noise patterns, this ratio would drop to 0.2, without even taking into account all the ciphersuite specializations we support. Conversely, if we only ever wanted generated code for a single Noise protocol, then the ratio jumps to nearly 7. A more realistic estimate is a proof-to-code ratio of 1, based on the 44kLOC of C code produced for the five patterns we actively test and benchmark. This is on par with (or even better than) mature F* verification projects like HACL*.

Feature Comparison. We compare other Noise implementations in Figure 7.

Noise* generates specialized code, and is a compiler (C). WireGuard and Brontide are specialized, built-in (B) implementations for the purposes of a single application. Other implementations are interpreted (I). We count all patterns, even those that do not appear in the Noise Protocol Framework.

An implementation offers a *Lean API* if it establishes a clear abstraction boundary that strives to prevent user mistakes. Details vary; here, the presence of a state machine with abstract send and receive functions is enough to qualify as a "Lean API". WireGuard and Brontide are omitted, since they use a single Noise protocol and therefore leverage that fact to traverse abstraction boundaries.

An implementation successfully handles *Early Data* if it allows the user to use early message payloads, while preventing confidentiality issues one way or another. WireGuard uses a custom scheme that has been carefully audited; Brontide prevents sending payloads altogether before the handshake is finished.

An implementation with *Key Validation* provides a way of validating keys upon receiving them, e.g., by calling a user-provided function. Finally, an API with *Key Storage* provides a long-term, secure way of storing and retrieving preshared or remote static keys.

Code sizes vary according to the feature set and the language used. For Noise*, we list the average size of a *single*, specialized C implementation. Noise* is larger than e.g., Cacophony or Noise Explorer, because of a more verbose language (C) and a larger feature set. Noise* is smaller than Noise-C or Noise Java. Our choice of generating C code will, we hope, facilitate integration in existing codebases. We remark that not all implementations support the same cipher suites; this depends on the choice of the underlying cryptographic library. We are here limited by e.g., the absence of Curve448 in HACL*; fortunately, none of the applications we studied require it (this includes WhatsApp, not counted in this table).

Performance comparison. We compare the speed of our code with other Noise implementations in Figure 6. We compiled the C code for Noise-C and Noise* using gcc 7.5.0. We used QEMU to run WireGuard for benchmarking, the Criterion 0.3.3 crate to benchmark the Rust code and the Criterion 1.5.9.0 package to benchmark the Haskell code. We observe that our Noise* implementations beat all other general Noise libraries for handshakes per second. Of these, Noise-C is the closest in speed to Noise* and the performance difference is

Implementation	Type	Language	Patterns	Lean API	Early Data	Key Valid.	Key St.	Verif.	Code Size
Noise*	C	C	59	Y	Y	Y	Y	Y	~5000
Cacophony	I	Haskell	59	Y	N	N	N	N	~1800
Noise-C	I	C	40	N	N	N	Y	N	~9000
WireGuard	B	C	1	-	-	Y	N	N	~2700
Snow	I	Rust	59	N	N	N	N	N	~3400
Noise Explorer	I	Rust/Go	50	Y	N	N	N	N	~900
Brontide	B	Go	1	-	-	N	N	N	~750
Noise Java	I	Java	40	N	N	N	N	N	~8000

Fig. 7. **Noise Implementations Comparison.** For Type: C = compiler, I = interpreter, B = builtin, i.e., a custom implementation.

Project	Model	Tool	# Patterns	Code Gen	Model Size	Verif. Time
Noise*	S	F*	59	Y	~680	10s
Vacarme [21]	S	Tamarin	53	N	~270	1.4 days
Noise Explorer [22]	S	ProVerif	57	Y	~650	0.5-24h
fACCE [23]	C	Manual	8	N	-	-
Dowling et al. [37]	C	Manual	1	N	-	-
WireGuard-CV [9]	C	CryptoVerif	1	N	~5000	1.5h

Fig. 8. **Noise Analysis Comparison.** For Model: S = symbolic, C = computational. Model Size and Verification Time are per pattern.

dominated by DH computations. The IKpsk2 implementation from WireGuard, which also uses HACL* for DH, incorporates careful kernel-level optimizations and is consequently 20% faster than the user-space Noise* code.

Security Analysis Comparison. Figure 8 compares our symbolic security analysis with prior formal proofs of Noise protocols. The closest related works are Noise Explorer and Vacarme, which both analyze (almost all) Noise protocols against Dolev-Yao attackers. Noise Explorer [22] compiles each handshake pattern to a ProVerif model and verifies it against a series of reachability queries corresponding to the different Noise secrecy and authenticity levels. The analysis of each protocol takes between 30 minutes and 24 hours. Vacarme generates Tamarin models for each protocol, and analyzes it against the strongest threat model supported by the protocol. Analyzing 53 protocols takes a total of 74 CPU days.

The key difference in our approach is that we verify a generic executable Noise specification using a modular, semi-automated proof technique based on dependent types. Hence, we are able to verify the whole protocol specification in about 9 minutes, which amounts to 10s per pattern. Furthermore, our proofs are for an executable specification of the whole Noise protocol stack, whereas Noise Explorer and Vacarme only focus on the protocol messaging code. Conversely, the protocol-level verification results of Vacarme are stronger than ours, since Tamarin can handle equivalence properties like anonymity and has a more precise model of Diffie-Hellman.

The security proof overhead for Noise* can be estimated by the ratio between our DY* proof and the functional specification, which is 2.4. Note however, that this is a proof for all 59 patterns, and is still just a fraction of the effort of developing the Low* implementation.

Figure 8 also notes other work on the computational analysis of Noise protocols: [23] defines a new security model for cryptographically analyzing multiple Noise protocols using pen-and-paper proofs; [37] describes a manual proof of WireGuard, including an analysis of IKpsk2; [9] presents a mechanized cryptographic proof of WireGuard using CryptoVerif. These works use a more precise cryptographic model than the symbolic models in our work or in Vacarme. However, the work needed to prove each protocol is significantly higher. Linking our verified implementations to computational proofs is an interesting direction for future work.

Other Related Work. Apart from work on Noise, prior works have investigated the automatic generation of protocol code from verified high-level protocol specifications, yielding implementations in Java [38], OCaml [17], [39], and F# [40], [41]. Each of these tools has been applied to a handful of protocols; the generated protocol code is tuned for correctness rather than performance and relies on unverified cryptographic libraries. In contrast, by relying on the F* ecosystem, we generate high-performance C code that is provably correct, memory safe, and linked to a verified cryptographic library. Furthermore, the flexibility and succinctness of the Noise specification language enables us to automatically generate verified implementations for 59 distinct protocols, yielding a comprehensive protocol library. Other prior works have focused on efficient code generation for specialized cryptographic constructions like multi-party computation and zero-knowledge proofs; we refer the reader to [11], [42] for a survey of this line of work. Finally, a long line of work has investigated techniques for directly verifying cryptographic protocol implementations written in F# [43], [44], [45], [8], F* [46], [16], Java [47], [19], and C [48], [49], [50]. In these settings, each protocol implementation must be verified independently, whereas our compiler-based approach allows us to verify a large class of protocol implementations once and for all.

VII. Conclusion

We have presented a Noise Protocol Compiler embedded within F*. Our compiler is verified once; then, for any choice of Noise Protocol and matching cryptographic implementations, it produces an efficient, low-level implementation in C. We generate not only protocol transitions, but also the entire protocol stack, including state machine, device and session management, user-configurable key policies, long-term key storage, and dynamic security levels. At all layers, we guard against user error by providing robust APIs. We go beyond the usual trifecta of memory safety, functional correctness and side-channel resistance, by connecting our verified stack to a symbolic security proofs based on the DY* framework. These extensive verification results come at no cost to performance; indeed our C code beats most existing Noise implementations.

Acknowledgment

This work was partially supported by the European Research Council (ERC) through Grant CIRCUS-683032, the Office of Naval Research (ONR) through Grant N000141812618 and the DST-INSPIRE Faculty Grant.

REFERENCES

[1] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *IEEE Computer Security Foundations Symposium*, 2016, pp. 164–178.

[2] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," in *Foundations and Trends in Privacy and Security*, vol. 1, no. 1-2, 2016, pp. 1–135.

[3] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, ser. LNCS, vol. 8044, 2013, pp. 696–701.

[4] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of tls 1.3," in *ACM Conference on Computer and Communications Security*, 2017, p. 1773–1788.

[5] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *IEEE Symposium on Security and Privacy*, 2017, pp. 483–502.

[6] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *IEEE European Symposium on Security and Privacy*, 2017, pp. 435–450.

[7] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, p. 193–207, Oct. 2008.

[8] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *ACM Conference on Computer and Communications Security*, 2011, p. 341–350.

[9] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the wireguard virtual private network protocol," in *IEEE European Symposium on Security and Privacy*, 2019, pp. 231–246.

[10] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *IEEE Symposium on Security and Privacy*, 2013, pp. 445–459.

[11] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *IEEE Symposium on Security and Privacy*, 2021, pp. 777–795.

[12] National Vulnerability Database, "Heartbleed bug," CVE-2014-0160 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, Apr. 2014.

[13] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A Messy State of the Union: Taming the Composite State Machines of TLS," in *IEEE Symposium on Security and Privacy*, 2015, pp. 535–552.

[14] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: verified secure zero-copy parsers for authenticated message formats," in *USENIX Security Symposium*, 2019, pp. 1465–1482.

[15] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet *et al.*, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *IEEE Symposium on Security and Privacy*, 2019, pp. 634–653.

[16] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the tls 1.3 record layer," in *IEEE Symposium on Security and Privacy*, 2017, pp. 463–482.

[17] D. Cadé and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications," *Journal of Computer Security*, vol. 23, no. 3, pp. 331–402, 2015.

[18] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in webassembly," in *IEEE Symposium on Security and Privacy*, 2019, pp. 1256–1274.

[19] R. Küsters, T. Truderung, and J. Graf, "A Framework for the Cryptographic Verification of Java-like Programs," in *IEEE Computer Security Foundations Symposium*, 2012, pp. 198–212.

[20] T. Perrin, "The Noise Protocol Framework," http://noiseprotocol.org/noise.html, 2018.

[21] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. Basin, "A spectral analysis of Noise: A comprehensive, automated, formal analysis of diffie-hellman protocols," in *USENIX Security Symposium*, 2020.

[22] N. Kobeissi, G. Nicolas, and K. Bhargavan, "Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols," in *IEEE European Symposium on Security and Privacy*, 2019, pp. 356–370.

[23] B. Dowling, P. Rösler, and J. Schwenk, "Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework," in *Public-Key Cryptography*, vol. 12110, 2020, pp. 341–373.

[24] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *ACM Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[25] J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 17:1–17:29, 2017.

[26] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *ACM Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[27] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY* : A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code," in *IEEE European Symposium on Security and Privacy*, Virtual, Austria, Sep. 2021.

[28] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*. Springer, 2012, pp. 159–176.

[29] R. Barnes, K. Bhargavan, B. Lipp, and C. Wood, "Hybrid public key encryption," IRTF Internet-Draft draft-irtf-cfrg-hpke-12, 2021.

[30] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin, "HACLxN: Verified generic SIMD crypto (for all your favourite platforms)," in *ACM Conference on Computer and Communications Security*, 2020, p. 899–918.

[31] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué, "Implementing and proving the TLS 1.3 record layer," in *IEEE Symposium on Security and Privacy*, 2017.

[32] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou, "A security model and fully verified implementation for the ietf quic record layer," in *IEEE Symposium on Security and Privacy*, 2021, pp. 1162–1178.

[33] "Noise* Code Repository," https://github.com/Inria-Prosecco/noise-star.

[34] D. Dolev and A. Yao, "On the security of public key protocols," in *IEEE Transactions on Information Theory*, vol. 29, no. 2, 2006, p. 198–208.

[35] "DY* Code Repository," https://github.com/REPROSEC/dolev-yao-star.

[36] T. Y. C. Woo and S. S. Lam, "Authentication for distributed systems," in *Computer*, vol. 25, no. 1, 1992, p. 39–52.

[37] B. Dowling and K. G. Paterson, "A cryptographic analysis of the wireguard protocol," in *Applied Cryptography and Network Security*, 2018, pp. 3–21.

[38] A. Pironti and R. Sisto, "Provably correct Java implementations of spi calculus security protocols specifications," *Journal of Computer Security*, vol. 29, no. 3, p. 302–314, 2010.

[39] J. A. McCarthy, S. Krishnamurthi, J. D. Guttman, and J. D. Ramsdell, "Compiling cryptographic protocols for deployment on the web," in *International Conference on World Wide Web*, 2007, p. 687–696.

[40] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer, "A secure compiler for session abstractions," *Journal of Computer Security*, vol. 16, no. 5, p. 573–636, 2008.

[41] C. Fournet, G. L. Guernic, and T. Rezk, "A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms," in *ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 432–441.

[42] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *IEEE Symposium on Security and Privacy*, 2019, pp. 1220–1237.

[43] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 2, pp. 8:1–8:45, 2011.

[44] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *ACM Principles of Programming Languages*, 2010, pp. 445–456.

[45] M. Backes, C. Hritcu, and M. Maffei, "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations," *Journal of Computer Security*, vol. 22, no. 2, pp. 301–353, 2014.

[46] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin, "Probabilistic relational verification for cryptographic implementations," in *ACM Principles of Programming Languages*, 2014, pp. 193–206.
[47] M. Avalle, A. Pironti, D. Pozza, and R. Sisto, "Javaspi: A framework for security protocol implementation," *International Journal of Secure Software Engineering*, vol. 2, no. 4, pp. 34–48, 2011.
[48] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," *Journal of Computer Security*, vol. 22, no. 5, pp. 823–866, 2014.
[49] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from C protocol code by symbolic execution," in *ACM Conference on Computer and Communications Security*, 2011, pp. 331–340.
[50] S. Chaki and A. Datta, "ASPIER: an automated framework for verifying security protocol implementations," in *IEEE Computer Security Foundations Symposium*, 2009, pp. 172–185.

APPENDIX A
59 NOISE PROTOCOLS AND THEIR AUTHENTICATION AND CONFIDENTIALITY GOALS

Protocol Name	Message Sequence	Payload Security Properties			
		←		→	
		Auth	Conf	Auth	Conf
N	(premessages) → e, es [d_0] → [$d_1, d_2, \ldots$]	- -	- -	A0 A0	C2 C2
K	(premessages) → e, es, ss [d_0] → [$d_1, d_2, \ldots$]	- -	- -	A1 A1	C2 C2
X	(premessages) → e, es, s, ss [d_0] → [$d_1, d_2, \ldots$]	- -	- -	A1 A1	C2 C2
NN	→ e [d_0] ← e, ee [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A0 A0	C0 C1 C1	A0 A0 A0	C0 C0 C1
KN	(premessages) → e [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A0 A0 A0	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C1 C1
NK	(premessages) → e, es [d_0] ← e, ee [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A2 A2	C0 C1 C1	A0 A0 A0	C2 C2 C5
KK	(premessages) → e, es, ss [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C4 C4 C5	A1 A1 A2 A2	C2 C2 C5 C5
NX	→ e [d_0] ← e, ee, s, es [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A2 A2	C0 C1 C1	A0 A0 A0	C0 C0 C5
KX	(premessages) → e [d_0] ← e, ee, se, s, es [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C5 C5
XN	→ e [d_0] ← e, ee [d_1] → s, se [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A0 A0 A0	C0 C1 C1 C5	A0 A0 A2 A2	C0 C0 C1 C1
IN	→ e, s [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A0 A0 A0	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C1 C1
XK	(premessages) → e, es [d_0] ← e, ee [d_1] → s, se [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C1 C1 C5	A0 A0 A2 A2	C2 C2 C5 C5
IK	(premessages) → e, es, s, ss [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C4 C4 C5	A1 A1 A2 A2	C2 C2 C5 C5
XX	→ e [d_0] ← e, ee, s, es [d_1] → s, se [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C1 C1 C5	A0 A0 A2 A2	C0 C0 C5 C5
IX	→ e, s [d_0] ← e, ee, se, s, es [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C5 C5

Protocol Name	Message Sequence	←		→	
		Auth	Conf	Auth	Conf
Npsk0	(premessages) → psk, e, es [d_0] → [$d_1, d_2, \ldots$]	- -	- -	A1 A1	C2 C2
Kpsk0	(premessages) → psk, e, es, ss [d_0] → [$d_1, d_2, \ldots$]	- -	- -	A1 A1	C2 C2
Xpsk1	(premessages) → e, es, s, ss, psk [d_0] → [$d_1, d_2, \ldots$]	- -	- -	A1 A1	C2 C2
NNpsk0	→ psk, e [d_0] ← e, ee [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A1 A1	C0 C1 C1	A1 A1 A1	C0 C0 C1
NNpsk2	→ e [d_0] ← e, ee, psk [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A1 A1	C0 C1 C1	A0 A0 A1	C0 C0 C1
NKpsk0	(premessages) → psk, e, es [d_0] ← e, ee [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A2 A2	C0 C1 C1	A1 A1 A1	C2 C2 C5
NKpsk2	(premessages) → e, es [d_0] ← e, ee, psk [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A2 A2	C0 C1 C1	A0 A0 A1	C2 C2 C5
NXpsk2	→ e [d_0] ← e, ee, s, es, psk [d_1] ↔ [$d_2, d_3, \ldots$]	A0 A2 A2	C0 C1 C1	A0 A0 A1	C0 C0 C5
XNpsk3	→ e [d_0] ← e, ee [d_1] → s, se, psk [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A0 A0 A1	C0 C1 C1 C5	A0 A0 A2 A2	C0 C0 C1 C1
XKpsk3	(premessages) → e, es [d_0] ← e, ee [d_1] → s, se, psk [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C1 C1 C5	A0 A0 A2 A2	C2 C2 C5 C5
XXpsk3	→ e [d_0] ← e, ee, s, es [d_1] → s, se, psk [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C1 C1 C5	A0 A0 A2 A2	C0 C0 C5 C5
KNpsk0	(premessages) → psk, e [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A1 A1 A1	C0 C3 C3 C5	A1 A1 A2 A2	C0 C0 C1 C1
KNpsk2	(premessages) → e [d_0] ← e, ee, se, psk [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A1 A1 A1	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C1 C1
KKpsk0	(premessages) → psk, e, es, ss [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C4 C4 C5	A1 A1 A2 A2	C2 C2 C5 C5
KKpsk2	(premessages) → e, es, ss [d_0] ← e, ee, se, psk [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C4 C4 C5	A1 A1 A2 A2	C2 C2 C5 C5
KXpsk2	(premessages) → e [d_0] ← e, ee, se, s, es, psk [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A2 A2 A2	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C5 C5
INpsk1	→ e, s, psk [d_0] ← e, ee, se [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A1 A1 A1	C0 C3 C3 C5	A1 A1 A2 A2	C0 C0 C1 C1
INpsk2	→ e, s [d_0] ← e, ee, se, psk [d_1] → [d_2] ↔ [$d_3, d_4, \ldots$]	A0 A1 A1 A1	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C1 C1

Protocol Name	Message Sequence	←		→	
		Auth	Conf	Auth	Conf
IKpsk1	(premessages)				
	$\rightarrow$ e, es, s, ss, psk $[d_0]$	A0	C0	A1	C2
	$\leftarrow$ e, ee, se $[d_1]$	A2	C4	A1	C2
	$\rightarrow$ $[d_2]$	A2	C4	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
IKpsk2	(premessages)				
	$\rightarrow$ e, es, s, ss $[d_0]$	A0	C0	A1	C2
	$\leftarrow$ e, ee, se, psk $[d_1]$	A2	C4	A1	C2
	$\rightarrow$ $[d_2]$	A2	C4	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
IXpsk2	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, se, s, es, psk $[d_1]$	A2	C3	A0	C0
	$\rightarrow$ $[d_2]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
NK1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, es $[d_1]$	A2	C1	A0	C0
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	A2	C1	A0	C5
NX1	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ es $[d_2]$	A0	C1	A0	C3
	$\leftarrow$ $[d_3]$	A2	C1	A0	C3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	A2	C1	A0	C5
X1N	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ s $[d_2]$	A0	C1	A0	C1
	$\leftarrow$ se $[d_3]$	A0	C3	A0	C1
	$\rightarrow$ $[d_4]$	A0	C3	A2	C1
	$\leftrightarrow$ $[d_5, d_6, \ldots]$	A0	C5	A2	C1
X1K	(premessages)				
	$\rightarrow$ e, es $[d_0]$	A0	C0	A0	C2
	$\leftarrow$ e, ee $[d_1]$	A2	C1	A0	C2
	$\rightarrow$ s $[d_2]$	A2	C1	A0	C5
	$\leftarrow$ se $[d_3]$	A2	C3	A0	C5
	$\rightarrow$ $[d_4]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
XK1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ s, se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
X1K1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ s $[d_2]$	A2	C1	A0	C5
	$\leftarrow$ se $[d_3]$	A2	C3	A0	C5
	$\rightarrow$ $[d_4]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
X1X	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ s $[d_2]$	A2	C1	A0	C5
	$\leftarrow$ se $[d_3]$	A2	C3	A0	C5
	$\rightarrow$ $[d_4]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
XX1	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ es, s, se $[d_2]$	A0	C1	A2	C3
	$\leftarrow$ $[d_3]$	A2	C5	A2	C3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	A2	C5	A2	C5
X1X1	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ es, s $[d_2]$	A0	C1	A0	C3
	$\leftarrow$ se $[d_3]$	A2	C3	A0	C3
	$\rightarrow$ $[d_4]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
K1N	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ se $[d_2]$	A0	C1	A2	C1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A0	C5	A2	C1

Protocol Name	Message Sequence	←		→	
		Auth	Conf	Auth	Conf
K1K	(premessages)				
	$\rightarrow$ e, es $[d_0]$	A0	C0	A0	C2
	$\leftarrow$ e, ee $[d_1]$	A2	C1	A0	C2
	$\rightarrow$ se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
KK1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, se, es $[d_1]$	A2	C3	A0	C0
	$\rightarrow$ $[d_2]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
K1K1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
K1X	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
KX1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, se, s $[d_1]$	A0	C3	A0	C0
	$\rightarrow$ es $[d_2]$	A0	C3	A2	C3
	$\leftarrow$ $[d_3]$	A2	C5	A2	C3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	A2	C5	A2	C5
K1X1	(premessages)				
	$\rightarrow$ e $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ se, es $[d_2]$	A0	C1	A2	C3
	$\leftarrow$ $[d_3]$	A2	C5	A2	C3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	A2	C5	A2	C5
I1N	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ se $[d_2]$	A0	C1	A2	C1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A0	C5	A2	C1
I1K	(premessages)				
	$\rightarrow$ e, es, s $[d_0]$	A0	C0	A0	C2
	$\leftarrow$ e, ee $[d_1]$	A2	C1	A0	C2
	$\rightarrow$ se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
IK1	(premessages)				
	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, se, es $[d_1]$	A2	C3	A0	C0
	$\rightarrow$ $[d_2]$	A2	C3	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
I1K1	(premessages)				
	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
I1X	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s, es $[d_1]$	A2	C1	A0	C0
	$\rightarrow$ se $[d_2]$	A2	C1	A2	C5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
IX1	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, se, s $[d_1]$	A0	C3	A0	C0
	$\rightarrow$ es $[d_2]$	A0	C3	A2	C3
	$\leftarrow$ $[d_3]$	A2	C5	A2	C3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	A2	C5	A2	C5
I1X1	$\rightarrow$ e, s $[d_0]$	A0	C0	A0	C0
	$\leftarrow$ e, ee, s $[d_1]$	A0	C1	A0	C0
	$\rightarrow$ se, es $[d_2]$	A0	C1	A2	C3
	$\leftarrow$ $[d_3]$	A2	C5	A2	C3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	A2	C5	A2	C5

APPENDIX B
AUTHENTICATION AND CONFIDENTIALITY TARGET SECURITY LABELS FOR 59 NOISE PROTOCOLS

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties $\leftarrow$ Auth	$\leftarrow$ Conf	$\rightarrow$ Auth	$\rightarrow$ Conf
NN	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	1	0	1
KN	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	3	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	5	2	1
NK	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	0	5
KK	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	$\leftarrow$ e, ee, se $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
NX	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	0	5
KX	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
XN	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ s, se $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	1	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	0	5	2	1
IN	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	3	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	5	2	1
XK	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	$\rightarrow$ s, se $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
IK	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	$\leftarrow$ e, ee, se $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
XX	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ s, se $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
IX	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
N	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	-	-	0	2
	$\rightarrow$ $[d_1, d_2, \ldots]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	0	2
K	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	$\rightarrow$ $[d_1, d_2, \ldots]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
X	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	$\rightarrow$ $[d_1, d_2, \ldots]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
NNpsk0	$\rightarrow$ psk, e $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	0
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	1	1	1	0
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	1	1	1
NNpsk2	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	1	1	0	0
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	1	1	1
NKpsk0	$\leftarrow$ s	pre								
	$\rightarrow$ psk, e, es $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer]) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	1	2
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	1	2
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	1	5
NKpsk2	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	$\leftarrow$ e, ee, psk $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	1	0	2
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	1	5
NXpsk2	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	1	0	0
	$\leftrightarrow$ $[d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	1	5
XNpsk3	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ s, se, psk $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[3]$	0	1	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	1	5	2	1
XKpsk3	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	$\rightarrow$ s, se, psk $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
XXpsk3	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ s, se, psk $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
KNpsk0	$\rightarrow$ s	pre								
	$\rightarrow$ psk, e $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	0
	$\leftarrow$ e, ee, se $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	1	3	1	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
KNpsk2	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	1	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
KKpsk0	$\rightarrow$ s	pre								

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties $\leftarrow$ Auth	$\leftarrow$ Conf	$\rightarrow$ Auth	$\rightarrow$ Conf
	$\leftarrow$ s	pre								
	$\rightarrow$ psk, e, es, ss $[d_0]$	1	(CanRead [P idx$_i$.p; P idx$_i$.peer]) ⊓ (CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	0	0	1	2
	$\leftarrow$ e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer])	$= l_r[1]$	2	4	1	2
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, ss $[d_0]$	1	(CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	0	0	1	2
KKpsk2	$\leftarrow$ e, ee, se, psk $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	2	4	1	2
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
KXpsk2	$\leftarrow$ e, ee, se, s, es, psk $[d_1]$	2	(CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	$= l_i[2]$	(CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	2	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
	$\rightarrow$ e, s, psk $[d_0]$	1	(CanRead [P idx$_i$.p; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	0	0	1	0
INpsk1	$\leftarrow$ e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer])	$= l_r[1]$	1	3	1	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
INpsk2	$\leftarrow$ e, ee, se, psk $[d_1]$	2	(CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	$= l_i[2]$	(CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	1	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, s, ss, psk $[d_0]$	1	(CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	0	0	1	2
IKpsk1	$\leftarrow$ e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer])	$= l_r[1]$	2	4	1	2
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	0	0	1	2
IKpsk2	$\leftarrow$ e, ee, se, psk $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	2	4	1	2
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
IXpsk2	$\leftarrow$ e, ee, se, s, es, psk $[d_1]$	2	(CanRead [S idx$_i$.p idx$_i$.sid] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [P idx$_i$.p] ⊔ idx$_i$.peer_eph_label) ⊓ (CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	$= l_i[2]$	(CanRead [S idx$_r$.p idx$_r$.sid] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [S idx$_r$.p idx$_r$.sid; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	2	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
	$\leftarrow$ s	pre								
Npsk0	$\rightarrow$ psk, e, es $[d_0]$	1	(CanRead [P idx$_i$.p; P idx$_i$.peer]) ⊓ (CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label)	$= l_r[1]$	-	-	1	2
	$\rightarrow$ $[d_1, d_2, \ldots]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
Kpsk0	$\rightarrow$ psk, e, es, ss $[d_0]$	1	(CanRead [P idx$_i$.p; P idx$_i$.peer]) ⊓ (CanRead [S idx$_i$.p idx$_i$.sid; P idx$_i$.peer]) ⊓ (CanRead [P idx$_i$.p; P idx$_i$.peer])	-	(CanRead [P idx$_r$.p; P idx$_r$.peer]) ⊓ (CanRead [P idx$_r$.p] ⊔ idx$_r$.peer_eph_label) ⊓ (CanRead [P idx$_r$.p; P idx$_r$.peer])	$= l_r[1]$	-	-	1	2

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties $\leftarrow$ Auth	$\leftarrow$ Conf	$\rightarrow$ Auth	$\rightarrow$ Conf
	$\rightarrow [d_1, d_2, \ldots]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
Xpsk1	$\leftarrow$ s	pre								
	$\rightarrow$ e, es, s, ss, psk $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	$\rightarrow [d_1, d_2, \ldots]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
NK1	$\leftarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\leftrightarrow [d_2, d_3, \ldots]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	0	5
NX1	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ es $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	1	0	3
	$\leftarrow [d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	1	0	3
	$\leftrightarrow [d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	1	0	5
X1N	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	0	1	0	1
	$\leftarrow$ se $[d_3]$	4	$l_i[3]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	3	0	1
	$\rightarrow$ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	0	3	2	1
	$\leftrightarrow [d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	0	5	2	1
X1K	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	$\rightarrow$ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	2	1	0	5
	$\leftarrow$ se $[d_3]$	4	$l_i[3]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	5
	$\rightarrow$ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	$\leftrightarrow [d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
XK1	$\leftarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ s, se $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow [d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
X1K1	$\leftarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	2	1	0	5
	$\leftarrow$ se $[d_3]$	4	$l_i[3]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	5
	$\rightarrow$ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	$\leftrightarrow [d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
X1X	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	2	1	0	5
	$\leftarrow$ se $[d_3]$	4	$l_i[3]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	5
	$\rightarrow$ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	$\leftrightarrow [d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
XX1	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ es, s, se $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	1	2	3
	$\leftarrow [d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow [d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
X1X1	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ es, s $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	1	0	3
	$\leftarrow$ se $[d_3]$	4	$l_i[3]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	3
	$\rightarrow$ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	$\leftrightarrow [d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
K1N	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ se $[d_2]$	3	$l_i[2]$ $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	1	2	1
	$\leftrightarrow [d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	0	5	2	1

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
K1K	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
	$\rightarrow$ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	$\leftarrow$ e, ee $[d_1]$	2	$l_i[1] \sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1] \sqcap$ (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	$\rightarrow$ se $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
KK1	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
K1K1	$\rightarrow$ s	pre								
	$\leftarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ se $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
K1X	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ se $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
KX1	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	$\rightarrow$ es $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	3	2	3
	$\leftarrow$ $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
K1X1	$\rightarrow$ s	pre								
	$\rightarrow$ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ se, es $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	1	2	3
	$\leftarrow$ $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
IK1	$\leftarrow$ s	pre								
	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
I1K1	$\leftarrow$ s	pre								
	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ se $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
I1X	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	$\rightarrow$ se $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	$\leftrightarrow$ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
IX1	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	$\rightarrow$ es $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	3	2	3
	$\leftarrow$ $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow$ $[d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
I1X1	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ se, es $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$	$= l_r[3]$	0	1	2	3

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties $\leftarrow$ Auth	$\leftarrow$ Conf	$\rightarrow$ Auth	$\rightarrow$ Conf
			(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])		(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)					
	$\leftarrow [\overline{d_3}]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow [d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5