
Karthikeyan Bhargavan*, Abhishek Bichhawat†, Quoc Huy Do‡, Pedram Hosseyni‡,
Ralf Küsters‡, Guido Schmitz‡, and Tim Württele‡

*INRIA
Paris, France
karthikeyan.bhargavan@inria.fr

†Carnegie Mellon University
Pittsburgh, PA, USA
and IIT Gandhinagar
Gandhinagar, Gujarat, India
abhishek.b@iitgn.ac.in

‡University of Stuttgart
Stuttgart, Germany
{quoc-huy.do, pedram.hosseyni, ralf.kuesters,
guido.schmitz, tim.wuerttele}@sec.uni-stuttgart.de

Abstract—We present DY*, a new formal verification framework for the symbolic security analysis of cryptographic protocol code written in the F' programming language. Unlike automated symbolic provers, our framework accounts for advanced protocol features like unbounded loops and mutable recursive data structures, as well as low-level implementation details like protocol state machines and message formats, which are often at the root of real-world attacks.

Our work extends a long line of research on using dependent type systems for this task, but takes a fundamentally new approach by explicitly modeling the global trace-based semantics within the framework, hence bridging the gap between trace-based and type-based protocol analyses. This approach enables us to uniformly, precisely, and soundly model, for the first time using dependent types, long-lived mutable protocol state, equational theories, fine-grained dynamic corruption, and trace-based security properties like forward secrecy and post-compromise security.

DY* is built as a library of F' modules that includes a model of low-level protocol execution, a Dolev-Yao symbolic attacker, and generic security abstractions and lemmas, all verified using F'. The library exposes a high-level API that facilitates succinct security proofs for protocol code. We demonstrate the effectiveness of this approach through a detailed symbolic security analysis of the Signal protocol that is based on an interoperable implementation of the protocol from prior work, and is the first mechanized proof of Signal to account for forward and post-compromise security over an unbounded number of protocol rounds.

1. Introduction

Since the early authentication protocols of Needham and Schroeder [48] and the key exchange protocols of Diffie and Hellman [31], the design and analysis of cryptographic protocols has come a long way. Modern protocol standards like Transport Layer Security (TLS) support multiple authentication modes, key exchange mechanisms, and encryption schemes, yielding dozens of possible combinations [52]. Messaging protocols like Signal (used in WhatsApp) invoke five Diffie-Hellman exchanges before even sending the first message, seeking to protect messages against powerful adversaries who can dynamically compromise phones and servers [46]. Protocols like these, with specifications that sometimes run to hundreds of pages, form the cornerstone of Internet security, and any flaw in their design or implementation could have a catastrophic effect. The comprehensive security analysis of such protocols requires automated tools.

Mechanized Cryptographic Protocol Analysis. The research community has developed several formal analysis techniques and (semi-)automated tools to verify cryptographic protocols (see [3, 19] for detailed surveys). Broadly, these methods can be divided into two categories. The first approach is to identify the cryptographic core of a protocol and to formally prove its (probabilistic) security based on precise computational assumptions on the underlying cryptographic primitives. However, building and maintaining computational proofs requires significant manual effort, and even with the aid of mechanized verification tools, it is infeasible to cover all protocol features and attack vectors for large protocols.

An alternative is to build comprehensive models of protocols and their threats, but to analyze them under simpler, stronger, symbolic assumptions on the cryptographic primitives. In this paper, we focus on the symbolic approach, but both methods are complementary and can be used side-by-side to get stronger assurances about protocol security (see, e.g., [11] and Section 6).

Automated Symbolic Protocol Analysis. The study of symbolic methods for protocol analysis was initiated by Needham and Schroeder [48] and formalized by Dolev and Yao [32]. The first high-profile success of this approach was Lowe’s attack and fix for the Needham-Schroeder public key authentication protocol (NS-PK) [45].

Symbolic analysis techniques have since evolved by leaps and bounds. Most notably, automated provers like ProVerif [20] and Tamarin [47] can quickly analyze all possible execution traces of protocols and find attacks like Lowe’s in a matter of seconds. Recent advances allow for the symbolic analysis of cryptographic primitives like Diffie-Hellman [42, 54] and XOR [33, 41] that require equational theories, of protocols that rely on mutable state [40], of Web-based security protocols like OAuth 2.0 and OpenID Connect that require new attacker models [34, 35], of stronger confidentiality and privacy properties
The problem is even more acute for protocols that inherently have an unbounded recursive structure or manages recursive stateful data structures. For example, consider the Diffie-Hellman (DH) ratcheting protocol depicted in Figure 1, which is inspired by the Signal Double Ratchet protocol [49]. The protocol starts with an established key $K_0$ shared by both parties. The initiator also knows the responder’s last known DH public key $g^{h_0}$ and uses it to generate a fresh DH secret $g^{h_0} g$ and mixes it with $K_0$ to obtain a new session key $K_1$ (this is called a ratcheting step). Upon receiving $g^{1}$, the responder in turn invokes its own ratcheting step to generate $g^{1+1}$ and compute $K_2$. This process continues in an unending loop over the lifetime of the messaging conversation (which can last months). Since each key $K_{n+1}$ is recursively dependent on the previous key $K_n$, the analysis complexity grows with each round.

Analyzing such protocols for an arbitrary number of rounds requires induction. Tamarin supports some inductive reasoning by relying on user-supplied lemmas, a technique that has been used to analyze group key agreement [53], but the proof requires many manual proof steps and hence is no longer fully automated, and still does not offer the full flexibility of general-purpose proof frameworks like Coq [58] or LF [57]. Notably, Signal has not been mechanically analyzed for an arbitrary number of rounds before. The ProVerif analysis of the Signal protocol [38] was limited to two messages (three ratcheting rounds), at which point the analysis already took 29 hours. (With CryptoVerif, the analysis of Signal has to be limited to just one ratcheting round [38].)

**Executable Protocol Code** Even for protocols that can be analyzed with tools like Tamarin, there remains a significant gap between the high-level protocol models analyzed by the symbolic provers above and the low-level protocol details specified in the standards or the implementations deployed in practice.

For clarity and ease of verification, protocol models are often succinct; they focus on core protocol features and ignore rarely-used or obsolete protocol modes. So they can miss attacks that rely on features like export ciphersuites [1] or session renegotiation [12]. Protocol models also ignore details like message formats and parsing, error handling, and state machines, which are often a key source of protocol bugs [9]. All these features could potentially be precisely modeled in (say) ProVerif, either by hand or by extracting models from reference implementations [15, 38, 55], but the resulting model becomes so large that automated analysis may take hours or not even terminate [13, 38].

Finally, as protocol models get larger, it becomes harder to be confident that the models themselves are correct. Hence, having executable models or analyzing protocol implementations is even more attractive, since these models can then be systematically tested against expected protocol traces.

**Dependent Type Systems for Protocol Code.** A different line of work, starting with RCF [8], seeks to address the limitations of automated protocol verifiers using dependent type systems that support the modular verification of cryptographic protocol implementations. The main drawback is that building security proofs with these type systems requires manual intervention and hence is less automated than provers like ProVerif and Tamarin.
Prior works on symbolic protocol analysis using RCF-like type systems for symbolic cryptographic models [36]. Even with intersection types [2], support for relational reasoning [4], the RCF type system in various ways: adding union and intersection types [2], support for relational reasoning [4], and computational cryptographic models [36]. Even with these advances, using RCF-like type systems for symbolic protocol verification has several limitations:

**External Lemmas** As noted above, typecheckers like F7 do not explicitly model the global trace-based runtime semantics, and so rely on external security arguments that need to be proved by hand. This increases the risk of accidentally introducing unsoundness, especially in large protocols, and reduces confidence in the proofs.

**Implicit Security Goals** Another disadvantage of not modeling the global trace is that protocol security goals are written in terms of dependent types that need to be manually interpreted as trace properties. This makes it difficult to express and prove properties about dynamic compromise, such as forward secrecy and post-compromise security, which depend on the precise order of events in the global trace.

**Equational Theories** Prior works on symbolic protocol verification using RCF-like type systems do not model cryptographic primitives like Diffie-Hellman or XOR that require equational theories.

**Mutable Protocol State** Most variants of RCF have limited support for stateful code with mutable data structures and hence can not be easily applied to verify protocols that rely on such features.

Due to these limitations, dependent type systems have lagged behind tools like ProVerif and Tamarin, and have never been used to symbolically verify (say) stateful Diffie-Hellman protocols like Signal with advanced security goals like post-compromise security.

**Our Approach.** The goal of our work is to build a mechanized symbolic verification framework that closes the gap between dependent types and dedicated provers like ProVerif and Tamarin, and by this, combines many of the mentioned benefits of both approaches.

We build a new framework, called DY* (as depicted in Figure 2), for the mechanized symbolic verification of protocol code written in \( F^* \) [56, 57]. \( F^* \) is a full-fledged programming language with a powerful dependent type system that supports user-defined effects for stateful code. The \( F^* \) type-checker can prove that programs meet their specifications using a combination of SMT solving and interactive proofs. While \( F^* \) has been used as a basis for computational protocol analysis (see Section 6), it has not been used for symbolic protocol analysis before this work.

To encode a symbolic cryptographic model within \( F^* \), we follow a radically different approach compared to prior work like F7. We explicitly model the global runtime semantics in terms of a mutable (append-only) global trace that tracks the interleaved distributed execution of an arbitrary number of protocol sessions. Hence, any property we need about the global trace, such as the attacker’s knowledge, secrecy lemmas about random bytestrings, assumptions about cryptographic primitives, or constraints on the order of protocol events, can be formulated in a natural way and proven sound within our framework, without relying on external assumptions or manual proofs.

Unlike previous approaches based on dependent types, the explicit treatment of global traces also allows us to express and prove security properties involving features like (long-lived) mutable state, dynamic compromise, forward secrecy, and post-compromise security. Finally, we extend this model to account for *equational theories*, which we then use to verify Diffie-Hellman-based protocols.

In \( F^* \), we have a full-fledged functional programming language at our disposal, so protocols can be modeled in
detail, including implementation features like session state storage that are usually left out in other approaches like ProVerif or Tamarin. Our protocol models are executable, and hence, testable using (say) test vectors from protocol specifications. This helps us to avoid creating faulty models, and also allows us to implement and test attacks. With F’’s powerful dependent type system and expressive proof environment, we are able to use induction-based proofs to model and verify unbounded and recursive protocols with complex data structures. Our approach is modular: we structure the DY* library and protocol code as independent modules with clean interfaces, and verify each module independently. As a result, verification time grows roughly linearly with code size, and we can build reusable libraries of generic protocol patterns and security lemmas that can be verified once and for all.

Contributions. Our main contribution is the design and implementation of DY*, realized as a library of 9 verified F* modules (see Figure 8) that together provide a domain-specific framework for modeling, executing and symbolically verifying security protocols. These modules essentially form two layers (see Figure 2). The lower layer, the symbolic runtime model (described in Section 2), provides libraries for symbolic cryptography, storage, and networking, implemented in terms of a global trace. It also models adversarial behavior by implementing an API for use by the (Dolev-Yao) symbolic attacker.

In principle, protocols can directly be implemented and verified on top of this layer. However, to ease verification and allow for succinct protocol proofs, DY* provides another layer, the labeled security library (described in Section 3), which factors out common protocol patterns, reusable security abstractions and invariants, and generic security lemmas that are proved sound once and for all with respect to the low-level trace-based semantics. The labeled layer provides a high-level API that imposes a security-oriented coding discipline using secrecy labels, authentication predicates, and usage constraints for key material. Obeying this discipline enables succinct protocol security proofs (see Section 5 for a detailed evaluation).

To illustrate our framework, we present the first symbolic analysis of the Signal protocol (in Section 4) that accounts for an unbounded number of ratcheting rounds. Our executable protocol model is based on an interoperable implementation of the protocol taken from prior work [51]. This analysis also offers the first type-based formulation and proof of post-compromise security for any protocol.

DY* is designed to be used by others; all the code and proofs for the framework are developed and documented in an open source repository, and can be used by other programmers to verify their own protocol code [10].

2. The DY* Symbolic Runtime Model

The DY* framework is meant to model a distributed system that consists of principals executing protocol code and exchanging messages over an untrusted network which is under the control of a Dolev-Yao adversary.

A central component of our model is the global (execution) trace. Among others, it records the history of the states of all principals at any time throughout the run of a system. A principal’s state may contain arbitrary information. For example, it can contain long-lived keys, such as the principal’s public and private keys. Also, principals may be involved in an unbounded number of sessions at the same time. Hence, a principal’s state also contains the current session state in all of its sessions.

At each step in a protocol, a principal first retrieves its current state from the global trace, possibly reads a message from the network, performs its computation, sends messages back to the network, and at the end of the invocation saves its new state in the global trace.

The global trace records all messages sent on the network by principals. It also records the nonces generated by principals and documents whether principals or their sessions (even versions of sessions, see below) are corrupted by the adversary, who can corrupt principals dynamically in a fine-grained way.

The trace determines the attacker’s knowledge at any point in a run: the attacker knows all messages sent on the network thus far as well as the state of corrupted principals or corrupted sessions of principals. This knowledge in turn determines which messages the attacker can send to (sessions of) principals. An attacker can only construct and send messages it can derive from its knowledge. In particular, it cannot simply guess secrets.

Modeling the global trace allows us to explicitly define and reason about the attacker’s knowledge and dynamic compromise within the proof framework in a sound way. In previous dependently-typed approaches, this was not possible, and the programmer had to rely on manual arguments and weaker security guarantees.

The global trace also allows us to naturally and explicitly express security properties, such as secrecy properties and authentication/integrity properties, involving features like (long-lived) mutable state, dynamic compromise, forward secrecy, and post-compromise security that require reasoning about the adversary’s knowledge and the precise order of events in the global trace. In order to prove such properties, we would typically formulate global invariants over the global trace which the code of every principals should preserve. The invariants should be strong enough to then imply the security properties we care about.

2.1. Encoding the Symbolic Runtime in F*

Next, we describe how the main components of our symbolic runtime model are encoded in F*.

An Explicit Global Trace. Formally, a global trace is defined as an array of certain entries in DY*:

```
records type entry =
| RandGen: b.bytes → l.label → u:usages → entry
| SetState: p.principal → v:versions → s:sessions → entry
| Message: s:principal → r:principal → m:bytes → entry
| Event: p:principal → (string × list bytes) → entry
| Corrupt: p:principal → session_id:nat → version:nat → entry

type trace = array entry
```

Each entry records one of five protocol actions. A record RandGen b l u indicates a fresh random bytestring b annotated with some metadata l and u has been generated, where l and u are used in the labeling layer (see Section 3). Similarly, SetState, Message, and Event record that a principal updated its internal state, sent a message on the network, or recorded a protocol event. We use protocol
events to annotate a trace to ease analysis. For example, a protocol event can be used to indicate that a new protocol session with a certain set of principals has been started or completed.

Each SetState entry stores a principal state grouped in so-called sessions, each annotated with a version identifier. Sessions can store long-term keys, such as a principal’s public and private keys, but also, as the name suggests, the principal’s states of arbitrary many ongoing protocol sessions. This partitioning is used by the entry Corrupt p session_id version that states that the attacker has obtained a specific version of a session (session_id) stored at principal p. This is a particularly fine-grained notion of compromise: it allows an attacker to dynamically compromise both long-term keys and specific ephemeral protocol states. By this type of corruption, we can model that only a subset of data stored in a principal’s state leaks to the adversary, allowing us, for instance, to analyze forward secrecy and post-compromise security. The attacker, however, is not restricted by this model as he can corrupt as many versions and sessions as it likes.

In our DY* implementation, we define one global variable that holds an append-only trace: each principal can add new entries at the end of the trace, but it cannot remove or modify existing entries. The length of the trace monotonically increases and hence can be used as a global symbolic timestamp.

The symbolic runtime layer provides (honest) principals with an API of basic functions, where underneath some of them read from and extend the trace; dishonest principals/the adversary can manipulate the trace in arbitrary ways subject to certain constraints (see later). For example, to generate a nonce principals would call the function gen which adds a RandGen entry to the trace and returns a fresh random bytestring; the function set_state adds a SetState entry while get_last_state reads the last SetState entry stored by a principal; send adds a Message entry to the trace, while receive_i reads the Message entry at a particular trace index, with the latter typically provided by the adversary to determine which message the principal is supposed to receive from the network. There are also functions to construct and parse messages (see later).

Global Trace Invariants. When verifying/type-checking the (stateful) code of a principal, we want to guarantee that we are in the context of a “reasonable” trace, i.e., not a trace that is completely arbitrary, but follows some generic rules. The symbolic runtime layer therefore contains a generic trace invariant to ensure basic constraints on traces. By so-called attacker-typability (explained later), we ensure that the invariant is sound in the sense that it does not overly restrict the adversary. The generic trace invariant is parameterized so that it can be enriched for proving desired security properties of a protocol, as discussed later.

An Algebraic Model of Crypto. The symbolic runtime layer defines an abstract type of bytestrings called bytes along with a series of conversion and cryptographic functions for constructing and parsing bytestrings. It also defines a type principal (alias for string) to represent the names of protocol participants (e.g. *alice*).

To model security assumptions on the cryptographic primitives, we define bytes as an algebraic datatype with multiple constructors, similar to the modeling style of [14]:

```haskell
type literal =
  | String of string | ByteSeq of seq FStar.UInt8.t | Nat of nat

type bytes =
  | Literal: lit:literal → bytes
  | Concat: b1:bytes → b2:bytes → bytes
  | Rand: index:nat → llabel → u:usage → bytes
  | PK: secret:bytes → bytes
  | PKEnc: pub_key:bytes → msg:bytes → bytes
  | DH_PK: priv_key: bytes → bytes
  | DH: priv_key: bytes → pub_key:bytes → bytes
```

The type literal defines constants like strings, natural numbers, and concrete bytearrays that are treated as opaque. The constructor Literal models the conversion of a literal to a bytestring; in concrete code, this conversion may use some encoding functions (e.g. UTF-8). Similarly, Concat models the concatenation of bytestrings. The constructor Rand represents a fresh random value that was generated at some index in the global trace with metadata l and u (see also Section 3).

Each cryptographic primitive is modeled in terms of constructors and functions that use pattern matching to break down bytestrings. For example, the constructor PK models the conversion of a private decryption key to a public encryption key, and PKEnc models public key encryption. These are used to define the three functions that define public-key encryption:

```haskell
let pk dk = PK dk
let pke_enc ek msg = PKEnc ek msg
let pke_dec ctx = match ctx with
  | PKEnc (PK dk) ms →
    if dk = dk then Success msg
    else Error "decryption_failed"
```

The functions pk and pke_enc simply call the symbolic constructors, while pke_dec uses pattern matching to look inside a symbolic ciphertext and checks if the decryption key matches the encryption key before returning the plaintext. We similarly define models for AEAD encryption, signatures, hashing, and MACs.

Note that the algebraic definition of the bytes type is only visible within the symbolic crypto library; it is kept abstract from the protocol code, which can only use this type via functions like string_to_bytes, gen, etc.

Encoding Equational Theories. For Diffie-Hellman, we define two constructors (DH_PK, DH) and associated functions: dh_pub generates the public key (g^y) from a private key, and dh computes a shared secret (g^xy), from a private key (x) and public key (g^y). To capture the mathematical properties of exponentiation, we need to define additional equations between bytes. There are many symbolic equational theories for Diffie-Hellman that have been considered in prior work (see [28] for more discussion.) Here, we encode the “standard” equation that is used in ProVerif, and has been used in prior analyses of protocols like Signal [11] and TLS 1.3 [11]. The rule says that (g^xy) is equal to (g^yx), as reflected in the dh_shared_secret_lemma:

```haskell
val dh_shared_secret_lemma: x:bytes → y:bytes →
  Lemma (((dh x (dh_pub y)) == (dh y (dh_pub x)))
```

By choosing a set of equations for a particular primitive, we are symbolically modeling the concrete computational semantics of the underlying cryptographic algorithm
at a particular level of precision. For protocols (like those in this paper) that rely on standard Diffie-Hellman operations over a large prime-order group, the equation above is enough to capture most known attacks, but for protocols that rely on other operations like addition and multiplication over exponents, we may need to add further equational rules to capture other potential attacks [44].

In our framework, we can use the full flexibility of F* to add any number of equational rules (for DH, XOR, etc.). The main cost is that we need to prove that the encoding of the equational theory is an equivalence (reflexive, symmetric and transitive) and that it is respected by all the functions and predicates in our symbolic model and in the protocol code, which can be a tedious verification task.

Sometimes, we can encode an equational rule in a way that eases these equivalence and preservation proofs. For the Diffie-Hellman equation above, we define a total order \texttt{le\_bytes} over the \texttt{bytes} type, and use it to ensure that the constructor application \texttt{DH x y} always has \texttt{x} less than \texttt{y}. If inputs are given in the other order (e.g. \texttt{DH y x}) we reorder them to maintain the invariant. This encoding allows us to prove that the function defining the equational theory coincides with logical equality (=) in F* and hence is a congruence.

Modeling Adversarial Behavior. We model an active network attacker, in the tradition of Dolev and Yao [32], who can intercept, modify, and block all messages sent on the network, can compromise any session state, and can call any cryptographic function (using messages it on the network, can compromise any session state, and can call any cryptographic function (using messages it receives), and can schedule any part of a protocol, i.e., functions that model the behavior of honest parties (see below). By using these capabilities, the attacker can grow its knowledge as the global trace is extended and it can try to break the protocol’s security goals.

Essential for an attacker’s behavior is its knowledge. The attacker’s knowledge at each index \texttt{i} in the global trace is logically characterized using a set of derivation rules defined as a recursive predicate \texttt{attacker\_can\_derive \_i \_steps \_b} that specifies whether the attacker can derive a bytestring \texttt{b} at index \texttt{i} within a certain number of derivation steps (see Appendix B). For example, the attacker can derive literals, can read any message sent on the network, and read previously compromised states in 0 steps. At each derivation step, it applies a cryptographic function to bytestrings it has already derived to obtain a new bytestring. Importantly, an attacker cannot guess a freshly generated random value (generated by some honest party); it can only learn a generated value by deriving it from network messages or compromised states.

The predicate \texttt{attacker\_knows\_at \_i} says that an attacker knows a bytestring \texttt{b} at trace index \texttt{i} if he can derive \texttt{b} at \texttt{i} in some (finite, but unbounded) number of steps:

\[
\begin{align*}
\text{let attacker\_knows\_at \_i \_b} & \equiv \\
\exists \_steps \_nat. \text{attacker\_can\_derive \_i \_steps \_b} \\
\text{type pub\_bytes \_i} & \equiv \_b\_bytes \_i \_attacker\_knows\_at \_i \_b
\end{align*}
\]

The type \texttt{pub\_bytes \_i} represents bytestrings known to the attacker at index \texttt{i}. Of course, these values are also known to the attacker at all subsequent indexes \texttt{j > i}.

We further implement a (typed) API for the use of the attacker (e.g., to implement an actual attack) by wrapping each function in the symbolic runtime layer with a attacker-friendly version that works for \texttt{pub\_bytes}. For example, the attacker API for public key encryption is as follows:

\[
\begin{align*}
\text{val pke\_enc \_i} & \equiv \_i \_nat \rightarrow \text{enc\_key\_pub\_bytes \_i} \\
\text{msg\_pub\_bytes \_i} & \rightarrow \text{pub\_bytes \_i} \\
\text{val pke\_dec \_i} & \equiv \_i \_nat \rightarrow \text{dec\_key\_pub\_bytes \_i} \\
\text{ctxt\_pub\_bytes \_i} & \rightarrow \text{result \_pub\_bytes \_i}
\end{align*}
\]

We typecheck our implementation of this attacker API to prove that as long as an attacker calls each function with derivable bytestrings, the function succeeds and returns derivable bytestrings. Furthermore, by showing that all the cryptographic and trace functions can be wrapped in this way, we demonstrate that the attacker has enough passive and active capabilities to construct and destruct all public bytestrings and to perform all the protocol actions (on behalf of a compromised principal) that an honest participant would be able to enact. This guarantee is sometimes called attacker typability for which we provide a machine-checked proof here using F*.

Specifying Protocols. A protocol is written as a set of functions, each of which defines one protocol step performed by a principal. These functions can be called by the adversary in arbitrary order. The parameters of these functions allow the adversary to specify which session of the protocol is to be invoked and which message the principal is supposed to read from the network. In particular, we have no restrictions on the number of principals or sessions in a protocol run.

When called, a function parses the principal’s state as well as the network message into some semantically rich data type (we provide protocol-dependent parsing and serializing functions). Next, the function performs the computation of the respective protocol step, serializes its results (a new state for this principal and possibly new network messages), and places these results on the trace (by storing the new state and sending the network messages). Since with F* we have a full-fledged functional programming language at our disposal, the functions can perform arbitrary computation and, in combination with global traces, easily deal with recursive, mutable, and long-lived state, unlike previous approaches.

We note that in principle, it is sufficient to have just one function for invoking a protocol since by reading the session state, the function can determine the current state of and stage of the protocol session. However, it is often more convenient to use several functions.

Symbolic Execution. For each protocol, we write a scheduler function which calls the above described protocol functions in the expected order. This scheduler essentially describes a run of the protocol and can be seen as a test case. We can then compile the scheduler together with the DY* framework and the protocol implementation to OCaml and execute this code to print out a symbolic trace of a protocol run. This way, we can implement test cases, inspect symbolic runs, and check our model for errors, something not possible in tools like Tamarin and ProVerif. We can also implement and check known attacks for unfixed protocol code.

Security Properties. Having the global trace explicitly formulated in our framework, unlike previous type-based approaches, we can easily and naturally formulate and then prove security properties about the trace in terms...
of $F^*$ lemmas. As already mentioned, this includes stating secrecy properties as well as integrity/authorization properties with a natural and fine-grained treatment of long-lived and mutual state, dynamic corruption as well as forward secrecy and post-compromise security. We provide concrete examples in Sections 2.2 and 4.

To prove such properties, we formulate appropriate trace invariants, show that honest programs satisfy these invariants (under arbitrary adversarial behavior), and then prove that these invariants imply the desired properties.

While trace-based security properties sketched above are expressive enough to state many classic protocol security goals, they however, do not capture stronger secrecy and privacy guarantees that can be expressed using program equivalence. In the future, we plan to extend our methodology to specify and verify properties written in terms of diff-equivalence [21].

2.2. Example: Implementing NS-PK with $DY^*$

In this section, we demonstrate our methodology using the well-known Needham-Schroeder Public Key protocol (NS-PK) [48] depicted in Figure 3. We begin by writing a detailed executable specification, or reference implementation, of NS-PK in $F^*$, relying on $DY^*$’s symbolic runtime layer. In Section 3, we lift this implementation to the labeled security layer to analyze its security.

Messages. Figure 4 depicts an algebraic datatype message that defines three constructors (Msg1, Msg2, Msg3), corresponding to the three messages in the NS-PK protocol. The function serialize_message uses a string-to-bytes conversion function and concatenation to encode each message unambiguously as bytes; parse_message does the inverse. The return type of parse_message is result message which means that it can either be of the form Success m for some m of type message, or it can be Error s with some error string s. We prove a lemma part_message_correctness_lemma (see Figure 4) that says that the parsing is correct, i.e. it correctly inverts serialization, which is an important functional requirement for protocol code.

Session States. As mentioned in Section 2.1, a principal can use the storage API to save its long-term data items, such as private and public keys, and current states of all ongoing sessions as an array, denoted sessions, of (serialized) session states, which comes along with an array versions (called a version vector) that maps each session to a version number. Initially, all versions are 0. When a session state is updated, its version may optionally be incremented to indicate a new protocol phase.

In our NS-PK implementation, each session state has type session_st (Figure 4), denoting one of six forms: its own long-term private key (SecretKey), the name and public key for a peer (PublicKey), the initiator $I$ and responder $R$’s intermediate protocol states ($ISentMsg1$, $RSentMsg2$, $RReceivedMsg3$), and their final protocol states ($ISentMsg3$, $RReceivedMsg3$). Each protocol state stores the (claimed) identity of the peer and further material, such as nonces, needed to continue the protocol session. Like with messages, we define a serialization and a parsing function for session_st and prove a correctness lemma.

Our model of protocol messages and state is more detailed than other formal analyses, such as those carried out in ProVerif and Tamarin, which typically ignore persistent state storage, and do not consider message or state serialization. However, these details are needed to implement protocols in practice, and a bug in serialization or parsing can easily break the protocol security goals. Our correctness lemmas prevent these kinds of bugs.

Initiator Code. The code for the NS-PK initiator is depicted in Figure 5 and is divided into two functions: the function initiate begins an NS-PK session and sends the first message, whereas initiate_complete processes the second message and sends the third message, completing the initiator’s role.

The function initiate begins by calling get_last_state to retrieve the last state (i.e. sessions, versions, and timestamp) stored by the principal i. It then calls find_public_key to search the sessions array for a PublicKey entry containing the responder $r$ and its public key pk_r. Our $DY^*$ framework supports, by an appropriately defined $F^*$ monad, implicit error propagation: if a function call like find_public_key fails and returns an error, the whole function fails and returns the same error.
Next, initiate calls gen to generate a fresh random bytesting $n_i$; gen takes three arguments with metadata that describe who generated the bytesting, who can read that value, and how $n_i$ is intended to be used. This metadata will be later used for proofs in the labeling layer (see Section 3).

The initiate function then constructs the first message $msg_1$, serializes it to bytes, and encrypts the result with the public key of $r$ to obtain the ciphertext $c_{msg_1}$. It creates a new session state $iS_{sentMsg1}$, $r_{n_i}$, serializes it to obtain $s_{st1}$, extends the previous state of $i$ with this new session state (at version $0$) and calls the set_state function to store the new state for $i$.

Finally, the initiator logs an event initiated indicating that it has started a new protocol session before calling the networking function send to put the message $c_{msg_1}$ on the network; besides the message, send takes the identities of the sending and receiving principals, here $i$ and $r$, respectively. The function initiate returns the index of the new session (length sessions), which is treated as a session handle and which the adversary or a scheduler can use to invoke and continue this session later on.

The code for initiate_complete is a bit more complicated; it is given a session handle session_id for an ongoing session, and a handle (msg_id) for a message that has been received over the network. The function retrieves its session state by calling get_last_state, receives the message by calling receive, decrypts the message with the initiator’s private key, verifies that the received nonce $n_i$ matches the nonce $n_i$ in the stored state, and then constructs and encrypts $msg_3$ for the responder. The function finishes by updating its session state to $iS_{sentMsg3}$ (hence deleting the previous state), logging an event InitiatorDone indicating that its role in the protocol session is complete, and sending the third message to the responder.

**Responder Code.** The code for the responder (included in [10]) is the dual of the initiator; it consists of two functions, respond and respond_complete, that store session states ($R_{sentMsg2}$, $R_{receivedMsg3}$), and log events (Responded, ResponderDone) each with three parameters [string_to_bytes in_in_r].

**Implementing Lowe’s Attack on NS-PK.** Using the attacker API and the protocol API (the above mentioned protocol functions for NS-PK), we can implement potential attacks on the NS-PK protocol as $F^*$ programs and symbolically execute them on the protocol code. In particular, we implement Lowe’s man-in-the-middle attack on NS-PK [45]. Lowe’s attack is based on mixing two sessions: (1) Alice as initiator (intentionally) talking to the attacker (as responder) and (2) the attacker pretending to be Alice (as initiator) talking to Bob (see Figure 9 in Appendix A). We symbolically execute the attack and it succeeds, resulting in a symbolic trace with 22 entries, at the end of which the attacker is able to learn the nonce $n_r$, which was meant to be known only to $I$ and $R$.

Lowe proposes a simple fix to NS-PK that adds $R$’s identity to $msg_2$ and requires $I$ to check this identity. We implement this fix and retry the attack code, which now fails as expected. The failure of a single attack however is not a formal proof of security. So the next step is to formalize and prove the security for the fixed Needham-Schroeder-Lowe (NSL) protocol.

**Security Properties.** For the NSL protocol, we are particularly interested in the secrecy of the nonce $n_r$ as well as authentication goals that link logged events. We here sketch how such properties can be expressed in $DY^*$. To express the secrecy of the nonce $n_r$ used in NSL, we ask whether it is possible for the attacker to interact with an initiator $I$ and responder $R$ in a way that it can eventually learn $n_r$, without compromising the long-term keys or the stored session states of $I$ or $R$. We state this in $DY^*$ by the following lemma, which says that if the $n_r$ value logged by $i$ in an initiator event InitiatorDone $r_{n_r}$ is known to the adversary at test_id, either $i$ or $r$ must have been compromised before test_id.

```plaintext
let n_r.secrecy_condition_at_i = 
vtest_id.ev_idx i r n_i r.ev_idx < test_id ∧ entry_at.ev_idx (Event i "InitiatorDone", |string_to_bytes r n_i n_i)) ∧ attacker_knows_at test_id n_r ⇒ (3 ∧ comp_idx.v s.comp_idx ≤ test_id ∧ entry_at_comp_idx (Corrupt i v s)) ∨ entry_at_comp_idx (Corrupt r v s))
```

We can write a similar lemma to express the secrecy goal for $n_r$ value stored in a responder’s state. The compromise condition in our secrecy invariant above is relatively broad: it gives no guarantees if any session of $i$ or $r$ is compromised. This is because any adversary who compromises the long-term keys of $i$ or $r$ can decrypt $n_r$ from the second or third message (NSL does...
not provide forward secrecy) and similarly the attacker can learn \(n_r\) by compromising the protocol states in which \(n_r\) is stored. For other protocols, however, it is possible to state secrecy goals with tighter compromise conditions (see Section 4).

In addition to secrecy, we also state authentication goals as properties of the trace that link logged events. For NSL, responder authentication at an initiator \(i\) states that if \(i\) logs an event \texttt{InitiatorDone} with parameters \(r, n, i\), and \(n_r\), then there must be a prior event \texttt{ResponseLogged} by \(r\) with matching parameters \(i, n, i\), and \(n_r\), unless one of \(i\) or \(r\) has been compromised. A similar property states initiator authentication at \(r\).

In the next section, we will see how we can prove these properties using the labeled layer of \(\text{DY}^*\).

**Comparison with Other Approaches.** In 340 lines of \(\text{F}^*\) code, we have formally specified the NS-PK protocol at an unprecedented level of detail. The core protocol code is quite short (less than 100 lines of code); the remainder accounts for low-level implementation details like message formats, the protocol state machine, long-term storage for public keys, and session storage for intermediate states. Furthermore, our code is executable and hence serves as a reference implementation of \(\text{NS-PK}\). Our security properties systematically allow for any stored state to be compromised, including intermediate states like \texttt{ISentMsg1}.

For comparison, formal models of NS-PK are included with both the ProVerif [20] and Tamarin [47] distributions. These models are written in about 110 lines in domain-specific protocol specification languages, but they do not account for message parsing, session state, or key compromise, and are not executable or testable. A model of key or state compromise can be added manually to each model in about 10-20 lines but precisely modeling session state would take significantly more work.

The size of our specification and the effort written to write it is comparable to other prior work on verifying symbolic protocol implementations. Models of the Otway-Rees protocol, which has a similar level of complexity to NS-PK takes 148 lines of \(\text{F}\#\) code in [15], and 234 lines in [14], but neither of these implementations account for session storage and state compromise. Adding these features would expand these implementations to roughly the same size as ours. In summary, it is possible to write more succinct formal models of NS-PK that ignore low-level implementation details, but ours is the first to account for all these details and can still be written succinctly with little modeling effort.

### 3. The \(\text{DY}^*\) Labeled Security Library

One may try to directly prove in the symbolic runtime layer that the global traces generated by a protocol (in parallel with an attacker) satisfy a certain security property or preserve a certain trace invariant. For example, in NSL, we would need to prove that the secrecy invariant for \(n_r\) is preserved by all four functions in the protocol API. However, in order to prove this invariant, it becomes necessary to strengthen it to include secrecy conditions for \(n_i\) and the private keys of \(i\) and \(r\). Establishing this extended invariant across all the functions of the protocol APIs may be laborious, even with the help of \(\text{F}^*\!'s dependent type checker.

Instead, we prove security properties about the low-level global traces with the help of a higher security abstraction layer, the labeled security library (or labeled layer for short), to facilitate proofs and obtain more succinct proofs (see Figure 2). The labeled layer factors out generic security abstractions and invariants, which we mechanically prove sound in \(\text{F}^*\) w.r.t. our lower-level trace-based runtime semantics; a once and for all effort.

At the heart of our methodology is a security-oriented coding discipline for protocol code written in terms of secrecy labels and usage constraints. Labels allow us to proactively track knowledge of secrets. Whenever some secret is generated (e.g., a nonce), we annotate this secret with a label that states who is allowed to know this secret. To enforce that the labeling always holds true, we add a global trace invariant that describes a valid trace. For example, in a valid trace, messages sent to the network must always be publishable (according to their labeling) and principals only store terms in their states that they are allowed to know.

Usage constraints complement labeling: We annotate key material with a usage, for example: a key may only be used for signing but not for encryption (which rules out decryption oracles). Moreover, the annotation can also express that a key may only be used for cryptographic operations with certain payloads, e.g., that some key is only ever used to sign specific messages. This allows us to (by local type checking) even reason about the behavior of other honest principals.

We refine the types of each function on the symbolic runtime layer to help preserve labeling and usage constraints, including the valid trace invariant. For example, the messages passed to the \texttt{send} function are required to be publishable according to the labeling. By this, it is guaranteed that messages put on the network are publishable, as postulated by the valid trace invariant. In turn, this means that protocol code which only uses the labeled library and type checks correctly, obeys labeling and thus is automatically proven to never reveal terms that are labeled to be kept secret to an adversary.

This rich set of invariants and properties allows us to state and prove generic and common security properties, so that they do not have to be re-done time and time again. As mentioned, this approach is mechanically verified based on the low-level trace-based runtime semantics in \(\text{DY}^*\) itself, without resorting to an external, manual meta theory. We emphasize that, as also mechanically verified, all of these invariants do not restrict adversarial behavior. In particular, all actions of the Dolev-Yao adversary (running on the lower layer) preserve the higher-level valid trace invariant.

We now present key elements of the labeled layer in more detail before illustrating this layer with our running example, the NSL protocol.

**Secrecy Labels and Usage Constraints.** A secrecy label (see Figure 6 for syntax) indicates which sessions of which principals are allowed to read a bytestring: a label is either \texttt{Public}, scoped to a list of session state identifiers (\texttt{Can\_Read [id1;id2;...]}), a \texttt{Meet} (intersection) of labels, or a \texttt{Join} (union) of labels. A session state identifier can refer to a principal and all of its sessions (\(P\ p\)), to a specific session \(s\) of \(P\ s\), or even to a specific version \(v\) of that session (\(V\ p\ v\)). Usages describe how a bytestring may be used: it
may be a secret Nonce, an unique identifier (Guid), or a key
for public key encryption, AED encryption, signatures,
MACs, key derivation functions or Diffie-Hellman. Each
of the uses takes a string argument that is used to
distinguish the different types of keys (e.g., a PKE
long-term key PKE.Key "id" is different from an ephemeral
PKE key PKE.Key "onetime") and to define the usage of
derived secrets.

The predicates has_label and has_usage define a set of
inductive rules that assign labels and usages to bytestrings.
For example, a bytestring containing a literal (Literal l)
is given the label Public with no specific usage, whereas
a generated random value (Rand idx u) has the label l
and usage u. In this manner, these predicates define
labeling and usage rules for all constructors of the bytes
data type. Not all bytestrings are well-labeled, but well-
labeled bytestrings have a unique label and usage. The
type bytes refers to a bytestring with label l at timestamp i,
and secret l refers to a secret bytestring having label
l with usage u at timestamp i.

The labeling rules rely on a predicate can_flow, which
specifies when a bytestring with label l can flow into
a function that expects a bytestring with label l2. This
predicate is reflexive and transitive, and it allows data
to flow from less restrictive labels to more restrictive
labels. For example, a secret nonce with label Can_Read [P ; P]
can only be encrypted under a key with a label it can flow
to (e.g. Can_Read [P i]).

The type msg l refers to bytestrings whose labels can
flow to l at timestamp i. The predicate is publishesable holds
for bytestings that can flow to Public (and hence can safely
be sent over the network.)

**Labeled Crypto API.** The core of the labeled layer
is a labeled crypto API that provides labeled wrappers
for all the crypto functions on the symbolic runtime
layer and internally enforces labeling and usage rules.

For example, the labeled version of string_to_bytes always
returns a Ibytes i Public; concat takes two inputs of type
msg i l and also returns a msg i l; split does the reverse.

The labeled types of the crypto functions are more
subtle; the public key encryption functions are declared as
follows:

```
type private_dec_key (i: nat) (l: label) (s: string) =
  b:bytes(has_label i b l \∧ has_usage i b (PKE.Key s))
type public_enc_key (i: nat) (l: label) (s: string) =
  b:bytes(\exists (sk:private_dec_key i l s). b == sk)
```

The type private_dec_key refers to private decryption
keys with a label l, whereas public_enc_key refers to
bytestings obtained by applying the pk function to a
private_dec_key. Each function takes implicit arguments
i, l and s indicating the trace index, the label of the
private decryption key and the string to identify the type
of PKE.Key. When calling the function, these arguments
can be omitted if they are obvious from the context.

The encryption function pke_enc enforces two con-
straints. First, it requires that the input message m can
flow to the label l of the private decryption key. This
means that we cannot encrypt a more-secret message with
a public encryption key whose private key is less-secret; a
labeling constraint that is necessary to guarantee message
secrecy. Second, it requires that the input message m must
satisfy a protocol-specific usage predicate pke_pred i msg.
Each protocol defines this predicate to specify the kinds
of messages it is willing to encrypt with a key of type
public_enc_key i l's. If these two constraints are met, pke_enc
returns a Public ciphertext that can be safely sent over the
public network.

Conversely, the decryption function pke_dec takes a
private decryption key with label l, a public ciphertext,
and returns a decrypted message of type msg with the
additional guarantee that this message is either publishable
(it may have come from the attacker) or it must satisfy
pke_pred (it has been produced according to the constraints
on pke_enc).

Each DH private key has the type dh_priv_key i l's
indicating that it has a secrecy label l and that the shared
secret generated from this private key should have the
usage defined by the function dh_secret_usage that takes
as parameter the string s. The corresponding public keys
have type dh_pub_key i l's.

```
type dh_secret_usage: string \→ usage
type dh_pub: #i:nat \→ #i:label \→ #s:string \→
dh_priv_key i l's \→
dh_pub_key i l's
```

The function dh takes a private key with type
dh_priv_key i l's and a public key with type
dh_pub_key i l's to compute a shared secret with label
Join i l 12 and usage defined by dh_secret_usage given the
string s. The label Join i l 12 means that the shared secret may be used in any

```
val dh_secret_usage: string \→ usage
val dh_pub: #i:nat \→ #i:label \→ #s:string \→
dh_priv_key i l's \→
dh_pub_key i l's
val dh: #i:nat \→ #i1:label \→ #i2:label \→ #s:string \→
dh_priv_key i l 11 s \→
dh_pub_key i l 12 s \→
  b:bytes(i:Join i l 12)\has_usage i b (dh_secret_usage s))
```

Figure 6. Secrecy labels, usage constraints, and labeling predicates

```
type has_label: i:nat \→ b:bytes \→ #l:label \→ pred
val has_usage: i:nat \→ b:bytes \→ #u:usage \→ pred
type bytes (i:nat) (l:label) = b:bytes(has_label i b l)
let has_flow_to: i:nat (b:bytes) (l:label) =
  \exists (l' has_flow_to i b l').

type msg (i:nat) (l:label) = b:bytes(has_flow_to i b l)
let is_publicable: (i:nat) (b:bytes) =
  \exists l (has_flow_to i b l) (usage = b:bytes(has_label i b l \∧ has_usage i b u))
```
session covered by $t_1$ or $t_2$. We define several other variants of $\text{dn}$ function, including for cases where the peer’s public key is untrusted.

The types for the rest of the cryptographic API are similar (see Appendix C). In each construction, the arguments must satisfy some protocol-specific usage predicate $(\text{aead\_pred}, \text{sig\_pred}, \ldots)$, and in all encryption functions, we ask that messages must flow to the labels of the decryption keys.

As mentioned before, unlike previous approaches, we prove the soundness of the labeled API within $F^*$ w.r.t. our low-level runtime semantics. This was impossible in previous type-based approaches due to the lack of a low-level runtime semantics within the frameworks themselves. We note that our soundness proof of course uses the $F^*$ type system and verification framework. The soundness of $F^*$ itself is outside the scope of our work.

Global Trace Invariants and Stateful API. As mentioned, we define a generic trace invariant $\text{valid\_trace}$ that holds in all global traces generated by protocol code that follows the labeling rules. It enforces three properties: i) any message $b$ that is sent on the network must satisfy $\text{is\_publishable}$; ii) any state (sessions, versions) that is stored by an honest principal $p$ at index $i$ must satisfy the protocol-specific state invariant $\text{state\_inv}_i p$ versions sessions; furthermore, for each session $s$, the session state sessions.$[s]$ must flow to the label $\text{Can\_Read} [V p s \text{versions}.[s]]$; iii) any event $e$ with parameters $pl$ logged by principal $p$ at index $i$ must satisfy the protocol-specific predicate $\text{event\_pred}_i p e pl$.

We then define labeled wrappers for all the stateful functions in the symbolic runtime layer: gen, set state, get state, etc. and extend their pre-conditions and post-conditions to ensure that they preserve the $\text{valid\_trace}$ invariant. We also prove that all functions in the attacker API preserve $\text{valid\_trace}$, and hence, we obtain that all reachable global protocol traces satisfy this invariant.

Generic Security Lemmas. Given the $\text{valid\_trace}$ invariant, we obtain many security lemmas for free (as mentioned before, proven in the framework itself). For example, we prove that $\text{valid\_trace}$ implies that a nonce that is labeled with $\text{Can\_Read}$ readers can only be known to the adversary if one of the session states included in readers is compromised. This lemma is encapsulated in a stateful function that can be called by protocol code to establish a protocol specific secrecy goal:

```
val secrecy_lemma: b:bytes -> readers:list st_id -> DY unit
  (requires (\lambda ti -> valid_trace ti \land
             has_label (len ti) b (Can_Read readers)))
  (ensures (\lambda ti . ti -> ti == ti \land
             (attacker_knows_at (len ti) b ->
               contains_comprised_st_id readers)))
```

Similarly, the $\text{valid\_trace}$ invariant relates each logged event to a protocol specific $\text{event\_pred}$ precondition, which a protocol can instantiate to obtain its own authentication goal. In NSL, we use $\text{event\_pred}$ to specify initiator and responder authentication: the initiator code is allowed to log $\text{InitiatorDone}$ only if we can prove that either a matching $\text{Responded}$ has been logged previously or one of the two principals is compromised.

Example: Symbolic Security Theorems for NSL. We continue our example from Section 2.2 and illustrate how the security of NSL can be proven using the labeled security API. To prove the NSL security goals, we first need to port our NSL implementation to use the labeled API instead of the symbolic runtime layer.

The first step is to annotate all the keys and nonces used in the protocol with labels and usages. (We already did this in our example code in Section 2.) In particular, the private key of each principal $p$ is given the type $\text{private\_dec\_key}_i p$ “NSL”, and the public key has the corresponding public $\text{enc\_key}$ type; the two nonces $n_i$ and $n_r$ are given labels $\text{Can\_Read} [P i; P r]$, indicating their intended secrecy. We define a protocol state invariant $\text{state\_inv}$ that specifies the labels of all bytestrings stored in the session state. We then typecheck the code against the labeled API to prove that it obeys all the labeling and usage rules. Finally, we can call the generic secrecy lemma provided before for $n_r$ to obtain our secrecy goal: if $n_r$ is known to the attacker, then one of the two principals $i$ or $r$ must be compromised.

To prove authentication, we define the $\text{event\_pred}$ to specify the order in which the four protocol events (Initiated, Responded, InitiatorDone, RespondedDone) may be logged in each session, and how their parameters must be related. We also need to define $\text{pk\_pred}$ to constrain what bytestrings may be encrypted by $i$ and $r$. We then typecheck the code to prove that each event and public-key encryption satisfies these predicates. Finally, we obtain initiator and responder authentication as a corollary of the $\text{valid\_trace}$ invariant.

The proof effort consists of specifying the event predicate and state invariant and typechecking the full protocol implementation, adding type annotations and lemmas where needed. Choosing these definitions and annotations requires some insight into the proof, but getting them wrong cannot affect soundness: an incorrect lemma will not be provable, and an incorrect predicate or annotation will not be allow us to prove the protocol security goals.

Comparison with Other Approaches. In total, we add about 188 lines of proof-related code to verify NSL. In contrast, tools like ProVerif and Tamarin can automatically verify the core NSL protocol for trace properties like secrecy and authentication with no additional annotations. Furthermore, verification with $F^*$ takes roughly 0.5 minutes, whereas ProVerif can verify the protocol in a few seconds. However, as we add more low-level protocol details, verification time with tools like ProVerif grows exponentially. For example, a comparably-sized ProVerif model of Otway-Rees generated from an F# implementation takes over 8 minutes to analyze [15]. Hence, for symbolically analyzing small protocols like NSL, automated provers require significantly less effort, but as protocols get larger or detailed, the benefits of modular verification methods like ours become more pronounced.

The proof effort required in $DY^*$ is significantly lower than the overhead in prior analyses based on dependent typechecking. For example, the F7 typechecker requires 255 lines of annotations to verify Otway-Rees, an overhead of over 100% on top of the protocol specification [14]. Furthermore, as we shall see in our Signal case study, our framework is able to model fine-grained compromise and stateful protocol implementations, which are out of the reach of these prior works.
4. Verifying the Signal Messaging Protocol

Despite their popularity, Diffie-Hellman (DH) protocols have traditionally been a challenging target for protocol verification. A precise model of DH requires an equational theory which can complicate proofs. Furthermore, DH protocols usually seek to provide advanced guarantees like forward secrecy that require reasoning about fine-grained dynamic compromise. Using DY*, we are able to mechanically verify, for the first time using dependent types, sophisticated DH protocols for strong security properties. Appendix D discusses our verification of a simple authenticated DH protocol.

The Signal protocol, used in popular messaging applications like WhatsApp and Skype, is notable for its sophisticated use of DH computations to obtain strong security guarantees against adversaries who can compromise both short-term and long-term secrets. Its innovative multi-round (or ratcheted) protocol design has inspired a line of work on new security definitions and proof techniques for properties like post compromise security [7, 23, 24, 37, 50]. Signal has also been analyzed with mechanized provers, both in the symbolic and the computational model, but these analyses had to severely restrict the protocol to tame its verification complexity [38]. Prior to this work, modular verification techniques like dependent types have not been used to analyze Signal.

Our verification target is an interoperable implementation of Signal in F* developed in prior work [51]. This implementation was verified for correctness against a purely functional protocol specification (also in F*), but the security of this protocol specification was not verified in F*. We close this gap by extending this specification code to a full DY* protocol model and proving that it achieves the secrecy and authentication goals of Signal, even for an unbounded number of rounds. We refer the reader to [51] and the F* source code for the full implementation details. Here, we will focus on how we model and verify the novel features of Signal.

X3DH: Initial Key Exchange. Each messaging conversation in Signal begins with the X3DH protocol depicted in Figure 7, which performs four DH operations involving two keys known to the initiator—a long-term identity key $k_i$ and an ephemeral key $e_i$—and three keys known to the responder—a long-term identity key $k_r$, a medium-term signed pre-key $s_r$, and a one-time pre-key $o_j$. (The one-time key is optional, but for simplicity we will assume it is always present.) The results of all four DH operations are fed into a key derivation function to obtain a root key $K_0$, which is then used to derive message encryption keys.

Each of the DH keys in X3DH serves a different purpose. The identity keys ($k_i, k_r$) are used to authenticate the two parties. The signed pre-key ($s_r$) is changed regularly and protects against the compromise of the responder’s identity key ($k_r$). The one-time pre-key ($o_j$) and ephemeral key ($e_i$) are specific to a single X3DH session and are deleted immediately after $K_0$ is generated, in order to provide forward secrecy for $K_0$ even if all other keys are subsequently compromised. In combination, the X3DH protocol seeks to provide defense-in-depth for $K_0$ against various combinations of key compromise. We show next how we can formalize and prove the precise secrecy goal for $K_0$ in our model.

![Figure 7. Extended Triple Diffie-Hellman (X3DH): a one-message key agreement protocol for asynchronous messaging.](image_url)
When a session’s version is incremented, old keys \((e, d)\) that were associated with previous versions can no longer be stored in this (or any other) session. Hence, by labeling ephemeral keys with specific versions, we ensure that they are discarded at the end of the protocol run.

By applying the labeling rules for DH and KDF, we obtain a label of the following form for \(K_0\):

\[
\text{Meet}(\text{Join}(\text{Can}_i, \text{Read}(\{s, 0\})), \text{Can}_r, \text{Read}(\{p, r\}))
\]

However, this is the \textit{ideal} label of \(K_0\) when both parties know the labels of all keys. In the presence of an active adversary however, each participant only has an incomplete view of the labels of its peer’s public keys.

In the initiator code, for example, we prove that \(K_0\) has a label \(l_0\) such that:

\[
\text{can}_\text{flow}_{l_0} (\text{Can}_i, \text{Read}(\{s, 0\}) \land \\
\text{can}_\text{flow}(\text{Join}(\text{Can}_i, \text{Read}(\{s, 0\})), \text{Can}_r, \text{Read}(\{p, r\}))_{l_0}
\]

In other words, \(K_0\) can be stored at version 0 in session \(sid_0\) and it is at least as secret as the label \(\text{Join}(\text{Can}_i, \text{Read}(\{s, 0\})), \text{Can}_r, \text{Read}(\{p, r\})\). Given this labeling, and applying the labeled secrecy lemma for \(K_0\), we obtain the above-stated secrecy goal for the initiator.

The proof for the responder’s secrecy goal is a bit more complicated: in addition to labeling, it also relies on the AEAD encryption of the X3DH message to authenticate the initiator’s ephemeral key, and hence to establish a stronger security invariant for \(K_0\). We note that most of the complexity of the protocol reasoning is succinctly represented by the labels given to different keys and the usage predicates for authenticated encryption. Verifying that each function respects the labels and usage predicates is relatively straightforward; most of the difficulty of using DH has already been abstracted away (in a provably sound way) once and for all at the labeling layer.

**DH Ratchet: Key Updates.** The root key \(K_0\) generated by X3DH is only the first in a tree of keys derived over the lifetime of a Signal session.

The DH ratchet (depicted in Figure 1 is executed whenever one of the parties receives a new ephemeral DH key \(v\) from its peer. The receiver generates a fresh DH key \(x\), computes \(g^x\) and mixes it with the old root key to obtain a new root key. This protocol can be executed indefinitely, resulting in a sequence of root keys \(K_0, K_1, K_2, \ldots\), where each root key \(K_n\) is deleted as soon as its successor \(K_{n+1}\) has been generated. The goal of this update mechanism is to protect long-running externalizing sessions against state compromise. We expect to provide secrecy guarantees for a root key \(K_{n+1}\) even when other root keys are compromised.

Signal also includes a second ratcheting mechanism called the KDF ratchet for deriving messaging keys from root keys. We do not detail this mechanism further in this paper, although it is implemented in our Signal model. We only observe that from each root key \(K_n\), the protocol derives an AEAD key that is used to authenticate the next ephemeral key \((x_{n+1})\). In other words, the root key \(K_n\) is used to authenticate the key material \(K_{n+1}\).

**Post Compromise Security for DH Ratchet.** Our implementation of X3DH establishes an initial root key \(K_0\) and stores it in an initiator session \(sid_0\) and responder session \(sid_0\) and sets both sessions to version 0. The code for DH Ratchet reuses these sessions; at each ratcheting step it computes and stores a new ephemeral key and root key, and increments the session version by 1.

The secrecy goal for root keys is defined as a recursive trace invariant that relates the secrecy of \(K_{n+1}\) to that of \(K_n\). Suppose a principal \(p\) has a Signal session \((sid_{p}, v)\) at version \(v\) with a peer session \((sid_{p}', v')\) at \(v'\), and suppose this session stores a root key \(K_{n+1}\) at time \(t\). Then, the secrecy goal states that there must have been a prior time \(t' < t\) when \(sid_{p}\) stored the previous root key \(K_{n}\) and \(K_{n+1}\) remains secret unless either \(K_n\) was made public before \(t\), or the current versions \(v\) and \(v'\) of \(sid_{p}\) and \(sid_{p}'\) are compromised.

In other words, if an active attacker compromises \(K_n\) before we compute \(K_{n+1}\), we get no guarantees for \(K_{n+1}\). Otherwise, we get a strong versioned secrecy guarantee for \(K_n\): compromising \(K_n\) later, or compromising any other session at \(p\) or \(p'\), or even compromising earlier or later versions of \(sid_{p}\) or \(sid_{p}'\) cannot affect the secrecy of \(K_{n+1}\). This secrecy invariant is our formulation of both the forward secrecy and post compromise security guarantees provided by the DH ratchet protocol.

The proof of the root key secrecy for the DH Ratchet protocol is similar to the proof of X3DH. By relying on the AEAD encryption, we prove that if \(K_n\) was not compromised before \(t\), then the new remote DH key \(y\) must have a label \(\text{Can}_r(V_{p' \text{sid}_{p} v'})\). We can then prove that \(K_{n+1}\) is at least as secret as the label \(\text{Join}(\text{Can}_i, \text{Read}(V_p \text{sid}_{p} v) ((\text{Can}_r(V_{p' \text{sid}_{p} v'}))))\). Finally, by applying the labeled secrecy lemma, we obtain the secrecy invariant for \(K_{n+1}\).

We note that our proof establishes an invariant for any \(n\) and does not depend on the number of ratcheting rounds, although the first round is a special case and has to be handled differently. We are therefore able to prove the security of DH Ratchet for any number of rounds. In contrast, even verifying the security of three DH ratcheting steps starts hitting the limits of automated symbolic tools like ProVerif [38].

Our full implementation of Signal composes X3DH with both ratcheting protocols and has many other details not detailed in this section, including detailed message formats and an application API. Using the proof techniques discussed here, we modularly verify the code for X3DH and DH Ratchet and use their security guarantees to prove authenticity and secrecy goals for the stream of messages exchanged in a Signal conversation.

## 5. Evaluation

The effectiveness of a mechanized protocol analysis framework like DY* can be measured along several axes: \textit{expressiveness} (i.e., what protocols and security properties it can verify), \textit{testability} (i.e., are the models correct), \textit{verification time} (i.e., how long does the tool take to verify a protocol), and \textit{human effort} (i.e., how much of the proof is automated). Figure 8 summarizes the verification results for the protocols considered in this paper. In this section, we discuss these results and compare them with prior work using dependent type systems like RCS, and with symbolic provers like ProVerif and Tamarin.
In terms of expressiveness, protocols like NSL and ISO-KEM can be analyzed by all prior frameworks, including symbolic provers and dependent type systems. Diffie-Hellman protocols like ISO-DH and Signal were, until this work, out of scope for RCF-like type systems since they do not support equational theories and cannot express the kinds of fine-grained compromise and forward/post-compromise secrecy guarantees we require in these protocols. Unbounded (looping) protocols like Signal, and protocols with mutable recursive data structures (e.g., ART [25]), are also out of scope for symbolic provers, without introducing artificial restrictions. For example, the only prior mechanized symbolic analysis of Signal was in ProVerif, and it had to be limited to two messages [38]. In contrast, we use the expressiveness of $\mathsf{F}^*$ along with the mechanized trace-based semantics of $\mathsf{DY^*}$ to model and verify Signal in full generality.

In terms of testability, all our models are executable, and so are prior models using dependent type systems, whereas models for symbolic provers are not. In terms of verification time, proofs using dependent types, including ours, are modular and so verification time grows roughly linearly with protocol size. This is in contrast with symbolic provers, where the verification time can grow exponentially with the number of messages considered in protocols like Signal. For example, in the ProVerif proof of Signal, going from one message to two messages increases verification time from 1 hour to 29 hours.

In terms of human effort, $\mathsf{DY^*}$ requires more manual proof annotations than symbolic provers, but we use the modular structure of $\mathsf{DY^*}$ to factor out common proof patterns into reusable libraries that only need to be verified once and for all. The generic $\mathsf{DY^*}$ library is written in about 2,880 lines of $\mathsf{F^*}$ (including 1,344 lines of proof-related code). Implementing small examples like NSL and ISO-DH requires about 400 lines of $\mathsf{F^*}$ code, distributed in 5 modules, of which the protocol logic takes about 100 lines and the rest of the code implements message formats, session state, and debugging code. Verifying the security of these small protocols requires about 150 lines of proof, which amounts to a similar proof burden as prior symbolic analyses using dependent type systems (see, e.g., [14]).

Our largest case study is Signal, which is itself broken up into modules implementing X3DH and Double Ratchet and shared modules for state management and messaging code, totalling 836 lines of code. Verifying this code requires 719 lines of proof annotations, most of which are for establishing the recursive trace invariants for the DH Ratchet. The proof overhead for Signal is higher than NSL, but this is primarily due to the complexity of the desired security properties, not due to the size of the protocol.

6. Related Work

We refer the reader to [3, 19] for comprehensive surveys of cryptographic protocol verification approaches. Here, we briefly discuss work closely related to $\mathsf{DY^*}$.

Automated Symbolic Provers. A long line of research on symbolic protocol verification has yielded tools like ProVerif [20] and Tamarin [47] that can automatically verify protocols like NSL in a few seconds. These tools offer more automation than $\mathsf{DY^*}$, but as discussed in Section 1, they offer limited support for modular analysis, looping protocols like Signal, recursive data structures, and executable models. Conversely, these tools can verify strong equivalence-based properties like indistinguishability, whereas $\mathsf{DY^*}$ currently only supports trace properties.

Dependent Types for Symbolic Analysis. Our work follows in the tradition of using dependent type systems like RCF to symbolically verify cryptographic protocol implementations [2, 4, 8, 14, 16, 17, 36, 57]. In comparison to these prior works, $\mathsf{DY^*}$ offers several improvements: fewer manual proofs, support for Diffie-Hellman protocols and mutable protocol state, and the ability to state and verify sophisticated trace properties like forward secrecy and post-compromise security.

Computational Provers. This paper focuses on symbolic verification, but several tools, such as EasyCrypt [5] and CryptVerif [18], have been developed for building game-based computational proofs for cryptographic constructions and protocols. There are also frameworks for cryptographically analyzing protocol implementations, like the CVJ framework [43]. In general, computational approaches offer stronger guarantees than symbolic frameworks like $\mathsf{DY^*}$ by relying on more realistic cryptographic assumptions. However, computational proofs require more manual effort and are not as automated as symbolic provers.

Dependent Types for Computational Analysis. Dependent type systems like $\mathsf{F}$ and $\mathsf{F^*}$ can also be used to help build computational proofs. Fournet et al. [36] show how to compose type-based protocol verification with manual cryptographic proofs to obtain computational security theorems. This methodology was used to verify a reference implementation of TLS 1.2 (called miTLS) using $\mathsf{F^*}$ [16, 17], and the record layer of TLS 1.3 using $\mathsf{F^*}$ [30]. However, the manual effort needed for protocol-specific cryptographic modeling and proofs can be significant, easily dominating the verification cost. For example, verifying the 3,600 line miTLS implementation requires 2,050 lines of type annotations for $\mathsf{F^*}$, as well as a large cryptographic proof that includes both manual arguments and a 3,000 line EasyCrypt proof [17]. This level of verification effort is probably only justified for multi-year case studies of important protocols like TLS.
7. Conclusion

We presented DY⋆, the first framework for symbolic protocol verification that combines the benefits of previous type-based approaches for reasoning using dependent types in a functional programming programming language with the kind of low-level trace-based guarantees one gets from automated provers, like ProVerif and Tamarin. DY⋆ supports modular analysis of large composite protocols, inductive reasoning for unbounded protocols, and accounts for low-level implementation details, which is outside of the scope of automated tools like Tamarin and ProVerif.

At the same time, DY⋆ explicitly incorporates and models low-level global traces. This allows us to naturally state and prove trace-based properties, and to account for advanced features like mutable state, dynamic compromise, forward secrecy, and post-compromise security, which was not possible with previous type based approaches. Also, unlike these prior approaches, DY⋆ does not rely on manual proofs, neither for proving protocols secure nor for proving the soundness of security abstractions.

Due to our treatment of equational theories, DY⋆ is also the first symbolic verification framework based on dependent types that can state and reason about Diffie-Hellman protocols and their intricate properties, as illustrated by our ISO-DH and Signal case studies. By virtue of the inductive reasoning supported by F⋆, we are also the first to provide a mechanized symbolic security proof of Signal for an unbounded number of protocol rounds. Furthermore, our analysis is based on an interoperable implementation of Signal. We believe that these kinds of verification results would have been unachievable with previous approaches.

The protocol code written for DY⋆ closely resembles real-world protocol implementations. We are working on building a concrete low-level library, based on the HACL⋆ verified crypto library [59], that can be used as a drop-in replacement for our symbolic runtime model to obtain interoperable protocol code. We plan to use this strategy to build verified reference implementations of sophisticated protocols like TLS 1.3 in future work.

DY⋆ still has many limitations. Proofs in DY⋆ are in a symbolic model, and hence are less precise than analyses based on computational cryptographic assumptions. We do not consider attacks outside our model, like timing side-channels. Finally, we only verify trace-based properties, and do not support equivalence-based security goals for confidentiality and privacy. We plan to improve on all these limitations in future work and apply DY⋆ to the verification of more sophisticated protocols and their implementations.

Acknowledgments

This work was partially supported by Deutsche Forschungsgemeinschaft (DFG) through Grant KU 1434/10-2 and the ERC Grant CIRCUS-683032.

References


Appendix A.

Lowe’s Attack on the Needham-Schroeder Public Key Protocol

Figure 9 illustrates Lowe’s man-in-the-middle attack on the NS-PK protocol [45]. The attack relies on mixing two sessions: (1) an honest initiator $I$ (intentionally) connects to the attacker $M$ (as responder), and (2) the attacker $M$ pretending to be $I$ (as initiator) talking to a responder $R$.

Lowe’s attack works because the second message (Msg2) of NS-PK (Figure 3) does not indicate the name of the responder, which allows a malicious $R$ to forward a Msg2 it received from some honest $R'$. A natural fix is to add $R'$’s identity to the second message, and for $I$ to check this identity before sending Msg3.

Appendix B.

Dolev-Yao Attacker Model

We model a network attacker who can read all Message events from the global trace and read any session state that has been corrupted. The attacker can also generate its own random values, call any function in the crypto API to construct and destruct bytestrings, and inject (unauthenti-
cated) messages from any sender to any receiver. Hence, the attacker’s knowledge and capability monotonically grows as the global trace is extended, as it learns more messages, compromises more states, and generates more random bytes.

Modeling Attacker Knowledge. To model the attacker’s knowledge at a certain trace index $i$, we define a recursive predicate $attacker\_can\_derive$ that specifies all the ways in which a attacker can derive a bytestring $b$ by applying
The predicate `attacker_knows_at` says that an attacker knows a bytestring `b` at trace index `i` if it can derive `b` at `i` in some (finite, but unbounded) number of derivation steps:

```haskell
let attacker_knows_at (i:nat) (b:bytes) = ∃ steps:nat. attacker_can_derive steps i b
```

The function returns `pub_bytes` indicating that the value must be logically derivable by the attacker at index `i`. Indeed, for each function in the attacker API, we prove (by typechecking in F#) that its symbolic implementation does not give the attacker any bytestring that it would not be able to derive using just the rules in the `attacker_knows_at` predicate.

We typecheck our implementation of the full attacker API against the low-level crypto and trace APIs, to obtain a soundness guarantee that is sometimes called `attacker typability`: it shows that the attacker has enough passive and active capabilities to construct and deconstruct all public bytestrings and to perform all the protocol actions (on behalf of a compromised principal) that an honest participant would be able to enact.

**Appendix C. Labeled Security API**

To enable security proofs for crypto protocol code, we define a higher-level API that constrains and tracks the flow of secret and authenticated data as it passes through session storage, cryptographic operations, and the untrusted network. In particular, we use secrecy labels to annotate and track the secrecy of bytestrings, _usage predicates_ to constrain the use of cryptographic keys, and _trace invariants_ for security properties.

The labeled security API can be seen as a coding discipline for well-behaved protocol code. Of course, protocols (and attackers) are free to directly use the low-level crypto and trace APIs, but by obeying the constraints imposed by...
the labeled API, the protocol can obtain strong trace-based secrecy and authentication guarantees with succinct proofs.

**Secrecy Labels and Data Flow.** The type label represents a secrecy annotation that can be applied to a bytestring:

```plaintext
type timestamp = nat

type principal = string

type st_id =
  | P: principal → st_id
  | S: principal → session:nat → st_id
  | V: principal → session:nat → version:nat → st_id

type label =
  | Public: label
  | Can: list st_id → label
  | Meet: label → label → label
  | Join: label → label → label

val can_flow: timestamp → label → label → pred
```

The label Public applies to bytestrings that are meant to be public, whereas Can_Read [id1:id2:id3:...:] is used to label bytestrings that are secrets shared by a list of principals, denoted by session state identifiers or st_ids [id1:id2:id3:...:]. An st_id may specify a principal (P p), and optionally a specific session of the principal (S p s), and optionally a specific version of that session (V p s v). For example in our protocols, we use the label Can_Read [P "alice"] for long term keys belonging to alice. They can be stored in any session belonging to alice and never have to be deleted. Conversely, we use the label Can_Read [V "alice" s v] for an ephemeral key that can only be stored in version v of session s of alice; once the session s is deleted or its version grows beyond v+1, the secret can no longer be stored and must be discarded.

The functions meet and join represent the intersection and union of labels. The predicate can_flow specifies under what conditions a bytestring with label l1 can flow into a function that expects a bytestring with label l2 at timestamp t. Informally, data is allowed to flow from less-secret labels to more-secret labels. We formally define can_flow as a recursive predicate, and prove that it is reflexive, transitive, and obeys several important rules:

- Public can flow to any label l;
- Can_Read [s1:s2:s3:...:] can flow to Public if one of the sessions identified by s1:s2:s3:... is compromised;
- Can_Read s1 can flow to Can_Read s1 if the st_ids in s2 are a subset of those in s1;
- Join l1 l2 can flow to l1 and to l2;
- l1 and l2 can both flow to Meet l1 l2.

Hence, the can_flow predicate can be seen as a way of tracking explicit information flow using label annotations.

**Constraining Cryptographic Usage.** The type usage is used as an annotation to describe how a bytestring should be used by well-behaved protocol code:

```plaintext
type usage =
  | Nonce: string → usage
  | Guid: string → usage
  | PK: string → usage
  | AEAD: string → usage
  | SIG: string → usage
  | MAC: string → usage
  | KDF: string → usage
  | DH: string → usage
```

For example, a bytestring may be used as a secret Nonce, as a globally unique identifier (Guid), as a key or nonce for public key encryption (PK KEY) or AEAD encryption (AEAD KEY) or signatures (SIG KEY), as a key for MACing (MAC KEY), or as a key used for key derivation (KDF KEY), or as a Diffie-Hellman (DH KEY) private key (DH KEY) that can in turn be used to compute a DH secret. The string parameter to the usage is used to distinguish usages of keys, and to derive the usage of shared secrets like a DH secret. All bytestrings, by default, can be used as payloads.

**Labeling Rules for Bytestrings.** We define two predicates, has_label and has_usage, that assign a unique label and an optional unique usage to each bytestring:

```plaintext
val has_label: l:nat → b:bytes → l:label → pred
val has_usage: l:nat → b:bytes → u:usage → pred
```

For example, any bytestring that is obtained from a literal using literal_to_bytes is given the label Public and has no specific usage (can be used only as payloads), while a fresh random value generated by calling gen l u is given the label l and usage u. Not all bytestrings are well-labeled; for example, it is forbidden to use a bytestring with a SIG_KEY usage as an AEAD KEY.

```plaintext
type bytes (i:nat) (l:label) = b:bytes [has_label i b l] 
let bytes_can_flow_to (i:nat) (b:bytes) (l:label) = 
  \exists l1 l2. has_label i b l1 ∧ can_flow i l1 l2
let is_publishable (i:nat) (b:bytes) = bytes_can_flow_to b l Public
```

We define refinement type abbreviations for different classes of bytestrings: the type bytes i l refers to bytestrings which have label l at timestamp i. The type msg i l refers to a bytestring whose label can flow to the label l at timestamp i. Intuitively, a value of type msg i l can be safely used in any function that expects a bytestring with label l. The predicate is_publishable holds for bytestrings whose labels can flow to Public; intuitively these values can be safely revealed to the attacker.

We also define abbreviations corresponding to all the usages that bytestrings may have in our library. For example, types for public key encryption and decryption keys are defined as:

```plaintext
type private_dec_key (i:nat) (l:label) (s:string) = b:bytes [has_label i b l ∧ has_usage i b l]
```

The type public_dec_key i l s refers to bytestrings that are obtained by applying the pk function to a value of type private_dec_key i l s, i.e. private decryption keys with label l and a string s identifying the usage at timestamp i.

**Labeled Crypto API.** We provide labeled versions of all the functions in the low-level crypto API; each function now has additional pre-conditions constraining the labels and usages of the inputs, but also provides more guarantees in its post-condition. Furthermore, for each function in the API, we provide lemmas guaranteeing that the bytestrings returned by the labeled versions are identical to those returned by the corresponding functions in the low-level crypto API.

We begin with literal_to_bytes, concat, and split functions:

```plaintext
val literal_to_bytes: #i:nat → literal → msg i Public
val bytes_to_literal: #i:nat → msg i Public → result literal
val concat: #i:nat → #l:label → msg i l → msg i l → msg i l
val split: #i:nat → #l:label → msg i l → result (msg i l ➔ msg i l)
```

The result of literal_to_bytes is always a Public payload: it can be freely used in the construction of any bytestring but cannot be used as a key or a nonce. The function concat
takes two bytestings that can both flow to a label l and returns a bytestring that also flows to l. The function split is the dual of concat; it splits a msg with label l into two.

The labeling rules for public key encryption are as follows:

val pk: #nat → #l:label → #s:string →
  private_dec_key i l s → public_enc_key i l s
val pke_pred: #nat → #l:label → msg i l → pred
val pke_enc: #nat → #l:label → #s:string →
  public_enc_key i l s → m:msg i l (pke_pred m) → msg i Public
val pke_dec: #nat → #l:label → #s:string →
  private_dec_key i l s → msg i Public →
  result (m:msg i l (is_publicable l m ∨ pke_pred m))

The function pk transforms a private_dec_key with label l and a string s identifying the type of PKE_Key usage at timestamp i to the corresponding public_enc_key.

The encryption function pke_enc enforces two constraints. First, it requires that the input message m at timestamp i must flow to the label l of the private decryption key. This means that we cannot encrypt a more-secret message with a public encryption key whose private key is less-secret; a labeling constraint that is necessary to guarantee message secrecy. Second, it requires that the input message m must satisfy a usage predicate pke_pred i l msg. This usage predicate is protocol-specific; each protocol defines this predicate to specify the kinds of messages it is willing to encrypt with a key of type public_enc_key i l s. If these two conditions are satisfied, pke_enc returns a Public message that can safely be revealed on a public network.

Conversely, the decryption function pke_dec takes a private decryption key with label l at timestamp i, a public ciphertext, and returns a decrypted message of type msgs i l with the additional guarantee that this message is either publishable (it may have come from the attacker) or it must satisfy pke_pred (i.e., it has been produced using pke_enc).

The types for the rest of the cryptographic API are similar; in each construction, the message must satisfy some protocol-specific usage predicate, and for encryption functions, messages must flow to the labels of the decryption keys.

Notably, while AEAD encryption produces a msg Public like public key encryption, the other functions in the API do not provide confidentiality. Functions like mac, hash, and sign preserve the label of the message input in the returned output. Finally, the labeled types for dh is as follows:

val dh_secret_usage: string → usage
val dh_pub: #nat → #l:label → #s:string → dh_priv_key i l s →
  dh_pub_key i l s
val dh: #nat → #l:label → #2:label → #s:string →
  dh_priv_key i l s → dh_pub_key i l s →
  b:bytes i (Join i 2 i has_usage i b (dh_secret_usage s))

The function dh_pub is similar to pk; it converts a private key to the corresponding public key. The function dh takes a DH private key with label l1 and a public key with label l2 and returns a bytestring with label Join l1 l2. Intuitively, if the private key belongs to one principal and the public key to another, then the shared secret is known to both. We also require that both DH keys must have the same usage DH_Key s, where s is the string identifying the usage of the shared secret given by the function dh_secret_usage.

Labeled Trace API and Valid Traces. If all honest principals use the labeled API, we can guarantee that all reachable global traces obey a strong security invariant called valid_trace:

let valid_trace (t:trace) = ∀ e. j < len t ∧ entry_at t e ==>
  (match e with
  | Message s r b → is_publishable j b
  | SetState p versions sessions →
    compromised_before j p ∨
    (state_inv j p versions sessions ∧
     (∀ s. s < len sessions ==> bytes_can_flow_to j sessions [s]
      Can_Read [V p versions [s]]))
  | Event p (e, pl) →
    compromised_before j p ∨
    event_pred j p e pl
  _ → ⊤)

The predicate valid_trace guarantees four properties:

- any message b that is sent on the network at timestamp i must be publishable (is_publishable j b);
- any message b that is sent over an authenticated channel from s to r at index i must satisfy the protocol-specific usage predicate auth_message preds i r b;
- any state sessions with version vector versions that is stored by principal p at index i must satisfy the protocol-specific state invariant state_inv i p versions sessions; furthermore, for each session s, the session state sessions [s] stored at i must flow to the label Can_Read [V p versions [s]]);
- any event e with parameters pl logged by principal p at index i must satisfy the protocol-specific event usage predicate event_pred i p e pl.

In other words, unless the issuing principal is compromised, all events and authenticated messages must satisfy usage predicates that are defined by each protocol. Similarly, the state stored by each principal should satisfy a protocol-specific state invariant. In addition, the labeling conditions on network messages and stored state safely limits the flow of secrets to the public network and compromisable storage device.

The valid_trace predicate trivially holds at index 0 (for empty traces) and we define a labeled version of the stateful API that guarantees that this invariant is preserved by all stateful functions. For example, the labeled version of gen is as follows:

("Labeled AEAD Encryption API")
val aead_enc: #nat → #l:label → #s:string →
  aead_key i l s →
  msg i Public → aead_option (msg i Public) →
  result m:msg i [can_floow i Public ∧ aead_option m]

("Labeled Signature API")
val vk: #nat → #l:label → #s:string →
  sign_key i l s →
  verify_key i l s
val sign_pred: #nat → #l:label → #k:l:label →
  msg i ml →
  sign: #nat → #k:l:label → #ml:label →
  #s:string →
  sign_key i k l s →
  m:msg i ml (sign_pred k l m) →
  msg i ml
val verify: #nat → #k:l:label → #ml:label →
  #s:string →
  verify_key i k l s → m:msg i ml →
  result m:msg i ml (sign_pred k l m)

("Labeled Hash Function API")
val hash: #nat → #l:label →
  m:msg i ml →
  result m:msg i ml

("Labeled MAC API")
val mac: #nat → #k:l:label →
  #ml:label →
  m:msg i ml →
  result m:msg i ml (mac_pred k l m) →
  msg i ml
The type of this function says that it has the DY effect, a special case of the F* State monad that only considers a single mutable variable (the global trace). Its pre-condition says that when it is called, the initial trace t0 should be valid. Its post-condition says that the final trace t1 is valid, and if the function succeeds, then the trace has been appended with one entry of the form RandGen v l u.

Secrecy and Authentication Guarantees. We verify the full labeled API, hence proving that it is sound with respect to the low-level APIs. Protocol code that is typechecked against the labeled API obtains some security lemmas for free. In particular, we prove, once and for all, that every labeled bytestring is secret as long as the principals in the label remain uncompromised. This guarantee is expressed through a stateful function called secrecy_lema:

val secrecy_lema : b:bytes -> readers:list_id -> DY unit
requires (λ t0 -> valid_trace t0 ∧ has_label (len t0) b (Can_Read readers))
ensures (λ t0 . t1 -> t0 == t1 ∧ (attacker_knows_at (len t0) b ==> contains_compromised_st_id_readers))

This function can be read as a challenge to the attacker: we ask the attacker to produce a bytestring b that was labeled with Can_Read readers and we prove that if the attacker can derive this bytestring in the current trace, then one of the sessions in readers must have been compromised in the past. In other words, the attacker cannot obtain a secret without explicitly compromising one of its intended readers.

In addition to this secrecy guarantee, our systematic treatment of usage predicates can be used to prove strong authentication guarantees. For example, signature verification and AEAD decryption provide post-conditions in terms of the usage predicates sig_pred and aead_pred, and we use these post-conditions to establish authentication invariants in protocol code. Furthermore, we use event_pred to encode protocol authentication goals as correspondence assertions between events logged by different protocol events.

Appendix D.
Signed Diffie-Hellman Key Exchange

In this section, we describe the verification of the signed Diffie-Hellman (ISO-DH) protocol depicted in Figure 11. Like in NSL, the initiator and responder exchange three messages and seek to authenticate each other and establish a session key. The code implementing these protocols has a similar structure to that of NSL, with one function for each step of the protocol at the initiator and responder. Here, we will focus only on the formal security analysis, leaving the reader to peruse the F* code for the implementation details.

Authentication using Signatures. The authentication goals are defined in terms of protocol events and require that both parties must agree on all session parameters. For example, if a principal i completes the session with parameters (i, r, g, y, k), then either the responder r is compromised, or it must have a session with matching parameters. The authentication guarantee provided by ISO-DH is stronger than NSL: each principal only relies on the honesty of the peer’s signature key, which means that, unlike NSL, ISO-DH is not vulnerable to Key Compromise Impersonation (KCI).

To prove our authentication goals, we rely on the guarantees provided by the labeled API for signatures. The signing key of each principal sk_p is given the label Can_Read [P p], indicating that it is a session independent (long-term) secret known only to p. We then define the usage predicate sign_pred to prescribe the use of signatures in ISO-DH: only messages matching the formats of MSG_2 and MSG_3 can be signed, and their contents must correspond to the session parameters at the sender. We prove that this predicate holds at all calls to sign, and then use the post-condition of verify to prove the authentication trace invariant at both parties.

Forward Secrecy vs. Signature Key Compromise. Each ISO-DH protocol run results in two local sessions, s_i at i and s_r at r. The DH private key y is labeled as Can_Read [S | i s_i], indicating that it is bound to a specific session at i. Similarly, y is labeled as Can_Read [S | r s_r]. The label of the shared secret is the join of the two labels, indicating that an attacker can only learn it by compromising one of the two sessions (s_i or s_r).

Using these labels, we can establish a trace invariant that states a precise forward secrecy guarantee for ISO-DH: a session key k that was stored in a session s_i (or s_r) at some trace index idx cannot be obtained by an adversary at some later trace index idx'>idx, unless the adversary has compromised one of the two sessions before idx’, or it has compromised the peer’s long-term signature key before idx. In particular, compromising long-term keys after idx, when the session is complete, does not affect session key secrecy.