Bob DyLean: A Framework for the Symbolic Analysis of Cryptographic Protocols in Lean

(v0.1 — July 2026)

Théophile Wallez
CISPA Helmholtz Center for Information Security

Cas Cremers
CISPA Helmholtz Center for Information Security

Abstract

Over the last decades, symbolic (Dolev-Yao) methods for the analysis of security protocols have proven to be effective to analyze and establish strong guarantees for widely deployed protocols and systems, such as TLS 1.3, E-voting protocols, EMV, and MLS. On the one hand, analysis methods like Tamarin and ProVerif provide automation and support for user-defined equational theories. On the other hand, methods like DY* offer more flexible and modular reasoning, but hardcode threat models and do not support custom equational theories.

We present DyLean, a framework for the symbolic analysis of cryptographic protocols in the Lean theorem prover. Our framework comprises both a flexible general-purpose symbolic semantics, as well as a concrete proof methodology.

DyLean allows defining protocols and expected security properties; its semantics and equational theories can be customized by the user. Furthermore, the semantics are agnostic of the specific proof methodology: our goal is to provide a generic framework that can be used by the community as a foundation to develop various proof methodologies.

Moreover, we provide a concrete proof methodology inspired by DY*, based on trace invariants. Thus, DyLean inherits from the qualities of DY*: it is able to analyze protocols involving unbounded loops or datastructures, and is able to compose security proofs in a variety of scenarios. Our proof methodology improves on DY* by allowing for user-defined equational theories and threat models. We exercise DyLean on several focused case studies, which include protocols using merkle trees, ratcheting protocols, post-quantum protocols, and protocols analyzed under different equational theories, which demonstrates that DyLean can effectively analyze protocols with each of these features.

1. Introduction

Over the last decades, the symbolic analysis of security protocols has proven very effective at scaling to larger protocol designs, which has both helped to establish strong security guarantees for protocols and standards [19, 38], but also to find vulnerabilities and propose fixes [20, 18, 10]. While the first works in this field used general-purpose model checkers and theorem proving methods, later advances have come from leveraging domain-specific insights and meta-results, leading to more automated modern tools such as Tamarin [30] and ProVerif [13]. These tools achieve automation by focusing on specific dedicated proof methods, and abstract algebraic structures.

Despite the ongoing improvement of such automated methods and their successes, they are reaching certain limits: their proof artifacts are typically not modular or robust under small protocol changes, their term definitions do not directly allow for defining structured datatypes (such as trees with nodes and leaves), and their dedicated proof methods may be insufficient to prove the desired properties for complex protocols. Because of this, there have been recent parallel developments that use symbolic reasoning with more generic proof methods. A prime example in this space is DY* [12, 39], a framework for symbolic analysis embedded in the proof assistant F* [33].

DY* [12, 39], by being embedded in a proof assistant, allows for structured datatypes (e.g. using inductive types), and allows for executing specifications (by compiling the specification and executing it with concrete bytestrings). Furthermore, DY* allows *modular verification*, in the sense that each function is analyzed independently, instead of relying on whole protocol analysis like Tamarin [30] and ProVerif [13]. In addition, the latest version of DY* [39] also allows for some form of *protocol composition*, which enables proofs of large protocols by composing security proofs of its sub-protocols. In the end, all these features enabled the analysis of large protocols such as TreeKEM [38], the continuous group key exchange sub-protocol of MLS [8].

Despite these qualities, DY* also has limitations: it comes with baked-in equational theories, meaning that users cannot analyze protocols that rely on cryptographic primitives not envisioned by DY* developers; and DY* comes with a fixed proof methodology, meaning that a protocol specified with DY* cannot be analyzed with something else than DY*'s proof methodology.

In this work we pursue two main objectives. Our first objective is to develop a new machine-checkable proof methodology, which is based on DY* [39], but extends DY* with user-defined equational theories and threat models. Additionally, we aim to leverage the active ecosystem and modern tooling of the Lean theorem prover [21].

Our second objective is more long-term: given recent developments in automated theorem proving, including AI-guided proof construction, we want to provide a generic framework

for symbolic reasoning for the research community. Our aim is that our underlying framework can be used to develop a range of novel proof techniques, integrate effective automated proof methods (such as those from Tamarin or ProVerif), and explore alternative proof methodologies.

Our objectives come together in the development of both an underlying library and theory for symbolic Dolev-Yao reasoning in Lean, which we subsequently leverage to develop our proof methodology that extends DY*.

Contributions. We present DyLean, a framework for symbolic security proofs in Lean [21]. Concretely:

- We design a framework to design equational theories, specify cryptographic protocols, and define reachability properties. We design this framework to be reusable by the community, so that multiple proof methods can be developed for, and applied to, the same specifications.
- We design a proof method for our framework that is inspired by DY*'s invariant-based proofs, but additionally allows for reasoning about equational theories and flexible threat models.
- We demonstrate the features of our approach in four case studies, covering key exchange, complex datastructures, equational theories, and advanced threat models. The latter two case studies go beyond what is possible to prove in DY*. For a comparable case study (key exchange), our proof verification is roughly twice as fast as DY*.

Artifact. DyLean is open source, we develop it publicly on GitHub here: `https://github.com/BobDyLean/dylean`. Furthermore, we provide a full artifact at [2] that includes our framework and case studies.

Overview. We proceed as follows. In §2 we describe how we model security protocols and their properties in Lean, which constitutes our core library. Then, in §3, we explain our proof technique. We then apply our methodology in §4. Finally, we discuss related work in §5 and conclude in §6.

2. Specifying Security Protocols in DyLean

In this section, we describe how we define cryptographic protocols and their intended security properties within our framework.

Overview. We will first provide a high-level overview of symbolic protocol modeling and related security properties, before turning to the details of our framework, for which we define

- terms (symbolic bytes) and equational theories in §2.2,
- protocol actions in §2.3,
- attacker knowledge in §2.4, and
- the complete transition system in §2.5.

We will then discuss some aspects of our framework in §2.6.

Notation. For two lists l, l', write $l + l'$ for the concatenation of l and l'. When clear from context, we write $l + e$ as a shorthand for $l + [e]$. We write $l \leq l'$ when l is a prefix of l'. We use $l[i]$ to denote the $(i+1)$th element of the list l, which is $\perp$ if i is out-of-bounds, and where $l[0]$ is the first element. We use $l[:i]$ to denote the first i elements of l, i.e. excluding $l[i]$. We write $l.\mathsf{len}$ for the length of the list.

2.1. Background

We first provide a high-level intuition for symbolic analysis of security protocols, which will provide context for the elements of our framework. The core idea behind symbolic protocol analysis is to achieve simpler (and hence more scalable) analysis by avoiding reasoning about concrete bitstrings, about the distributions of randomly generated values and about function outputs, but instead assume that they behave "as intended": encrypted values are meaningless to an attacker unless it has the corresponding decryption key, random values are uniformly distributed, and collisions between different message components occur at most with negligible probability and can hence be safely ignored. In practice, this is typically implemented by constructing messages as a term algebra, augmented with equational theories to more accurately model relevant cryptographic primitives (e.g. like commutativity for Diffie-Hellman exponentiation).

The next ingredient is then to model protocols. Depending on the framework, this can be approached in different ways. One tradition (followed by e.g. ProVerif) follows process-calculus based approaches, where the different roles in a protocol are modeled by (mostly linear) processes, with implicit interleaving and atomicity semantics. Another tradition (followed by e.g. Tamarin and DY*) follows state machine specifications, where the different roles are defined as a set of atomic state machine transitions with explicit interleaving.

A critical component is the definition of attacker knowledge. This was traditionally defined as taking the set of observable (sent) messages, and closing this set under the explicit attacker operations (like decrypting with a known key or applying public functions to known messages). Later frameworks also explicitly consider various corruption models, where the attacker can also learn messages not explicitly sent, e.g. by compromising long-term or short-term keys.

Once the protocol actions are modeled and the attacker knowledge is defined, these elements can be combined to define a transition system that models a (typically unbounded) number of participants performing any number of protocol role instances, while interacting with an untrusted network, which is effectively represented by the attacker.

Security properties are then defined over the set of reachable traces or executions of this transition system: for example, there is no trace in which the attacker can learn the session key computed by a client that is communicating with a peer whose long-term key was not compromised.

In the next sections, we will develop our DyLean model along these lines.

2.2. Symbolic Bytes and Equational Theories

We implement our message abstraction as terms that we refer to as *symbolic bytes*. We implement symbolic bytes using an

inductive type (with a twist so as to obtain a framework for symbolic bytes, rather than a fixed symbolic bytes), and we implement equational theories using term normalization, and justify the soundness of this approach.

Symbolic bytes. If we were to work with a fixed set of constructors for our symbolic bytes, we could create a type for symbolic bytes in Lean (or any proof assistant) using an inductive type: each constructor of the inductive type would correspond to a constructor of the symbolic bytes. However, since we aim to provide a *framework* of symbolic semantics, our set of constructors is not fixed, so we cannot simply write an inductive type. Instead, we use an approach inspired from Datatypes à la carte [34], which allows us to supply a *generic* type for symbolic bytes, which is parametrized by the set of constructors. We specify a constructor by writing a structure parametrized by the type of symbolic bytes (as illustrated in the next paragraph); then, given a set of constructors, our type for symbolic bytes is isomorphic to a fixpoint of all these constructors.

Defining a constructor in Lean. To define the constructor for a signature in DyLean, we write a structure parametrized by the type of symbolic bytes, which contains the two parameters of the signature constructor: the secret key and the message to be signed.

```
structure Sign (Bytes: Type) where
  sk: Bytes
  msg: Bytes
```

Equational theories. Computing whether two symbolic bytes are equal modulo equational theories is in general an undecidable problem (see e.g. [16]). Hence, every symbolic tool comes with a restriction on the equational theories they support: for example, ProVerif [13] and Tamarin [30] impose that equational theories have the Finite Variant Property [14, 22], which in particular requires equational theories to be convergent, i.e. that every bytestring must have a normal form. DyLean is not an exception, and imposes a similar requirement: we support equational theories for which one can exhibit a *normalization function* (which we will define in Definition 1). Such a normalization function will allow us to compute whether two symbolic bytes are equal modulo equational theories, by checking whether their normalization are *syntactically* equal (which we will prove in Theorem 1).

Normalization function. We first define what is a normalization function for a given equational theory, and then show how it relates to equality modulo equations.

Definition 1. *Given an equational theory E, a function $\mathcal{N} : \mathbb{B} \to \mathbb{B}$ is said to be a* normalization function for E *when the three conditions hold:*
(1) $\forall x, y, \{x, y\} \in E \implies \mathcal{N}(x) = \mathcal{N}(y)$
(2) $\forall x, x \simeq_E \mathcal{N}(x)$
(3) $\forall f \in \Sigma, \forall (x_i),$
$\mathcal{N}(f(\mathcal{N}(x_1), \ldots, \mathcal{N}(x_n))) = \mathcal{N}(f(x_1, \ldots, x_n))$
($\simeq_E$ is equality modulo equations, $=$ is syntactic equality)

Theorem 1. *Given an equational theory E and corresponding normalization function $\mathcal{N}$, then,*

$$\forall x, y, x \simeq_E y \iff \mathcal{N}(x) = \mathcal{N}(y).$$

The requirements for $\mathcal{N}$ have the following intuitive meaning: (1) means that equations yield syntactically equal bytestrings after normalization, (2) means that normalization does not normalize too much so as to make things equal when they shouldn't, (3) means that the normalization function takes care of normalizing the functions arguments.

Normalizing constructors. We construct such a normalization function using *normalizing constructors*: for every symbol $f \in \Sigma$, the user defines a normalizing constructor $\overline{f}$ which locally normalize the application of f. Then, we define the *global* normalization function $\mathcal{N}$ as follows:

$$\forall f \in \Sigma, \mathcal{N}(f(x_1, \ldots, x_n)) := \overline{f}(\mathcal{N}(x_1), \ldots, \mathcal{N}(x_n)))$$

In other words, global normalization function $\mathcal{N}$ applies the local normalization functions bottom-up. We will then see in Theorem 2 the properties that normalizing constructors must satisfy so that $\mathcal{N}$ is a normalizing function for an equational theory E (Definition 1)

Example of normalizing constructors. On the equational theory of signatures, we define the normalizing constructors:
- $\overline{\mathsf{Vk}}(sk) := \mathsf{Vk}(sk)$
- $\overline{\mathsf{Sign}}(sk, msg) := \mathsf{Sign}(sk, msg)$
- $\overline{\mathsf{Verify}}(vk, msg, sig) :=$
$$\begin{cases} \mathsf{true} & \text{if } \exists sk \text{ such that } vk \text{ matches } \mathsf{Vk}(sk) \\ & \text{and } sig \text{ matches } \mathsf{Sign}(sk, msg) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Defining a normalizing constructor in Lean. To implement $\overline{\mathsf{Verify}}$ in DyLean, we use pattern matching.

```
def verify (vk msg sig: Bytes): Bool :=
 match sig.view? Sign with
 | some { sk := sk, msg := msg' } =>
  msg = msg' &&
  vk = ({sk := sk} : Vk Bytes).pack
 | none => false
```

Soundness of normalizing constructors. We now give properties of normalizing constructors that imply that our function $\mathcal{N}$ is a normalization function for E (Definition 1).

Definition 2. *A symbolic bytestring x is said to be* normalized *when $x = \mathcal{N}(x)$.*

Theorem 2. *If*
(1) $\forall x, y, \{x, y\} \in E \implies \mathcal{N}(x) = \mathcal{N}(y)$
and for all symbols f, the corresponding normalizing constructor $\overline{f}$ satisfies the properties
(2) $\forall (x_i), f(x_1, \ldots, x_n) \simeq_E \overline{f}(x_1, \ldots, x_n)$
(3) $\forall (x_i), (\forall i, x_i \text{ is normalized}) \implies \overline{f}(x_1, \ldots, x_n)$ *is normalized*
then $\mathcal{N}$ is a normalization function for E.

Normalized bytestrings. In DyLean semantics, every symbolic bytestring is ground (i.e. has no variable), furthermore, we maintain the meta-property that every symbolic bytestring is normalized, because both protocol specifications and attacker knowledge only use normalizing constructors, and because applying a normalizing constructor on normalized bytestrings yields a normalized bytestring (case (3) of Theorem 2). In the end, this means that in protocol specifications, syntactic equality can be used to check for equality modulo equational theories. Furthermore, this means that the full normalization function $\mathcal{N}$ does not appear explicitly in DyLean: it stays implicit via the normalizing constructors.

Proof of Theorem 1 and Theorem 2. We prove both theorems in Lean in our artifact [2], in a separate, self-contained file, which serves to prove general facts about symbolic terms and equality modulo equations. We prove this in a separate file instead of integrating it in DyLean, because as explained in the previous paragraph, DyLean relies on normalizing constructors and syntactic equality, instead of equality modulo equations.

Checking hypotheses of Theorem 2. In practice, (2) and (3) are checked by inspection of the corresponding normalizing constructor; these checks can be performed independently on each set of normalizing constructors and equations. Finally, (1) is proven in Lean by proving a stronger statement: for example, on the equation of signatures, (1) is equivalent to proving that

$$\overline{\mathsf{Verify}}(\overline{\mathsf{Vk}}(sk), msg, \overline{\mathsf{Sign}}(sk, msg)) = \mathsf{true}$$

for normalized *sk* and *msg*; instead, we prove this statement in Lean even for non-normalized *sk* and *msg*.

2.3. Trace Entries and State Transitions

As we will see in §2.5, we represent all possible executions of a protocol in the presence of the attacker as a set of *execution traces*. Each execution trace is a list of *trace entries* from the user-defined set of trace entries, $\mathbb{E}_e$. We define the set of possible execution traces as $\mathbb{T}_e := \mathbb{E}_e^*$. We often use τ to denote a particular trace $\tau \in \mathbb{T}_e$.

State transition. In our symbolic semantics, protocol participants interact with each other by appending and looking up entries in the shared trace. We thus model state transition of a protocol participant by a function from traces to traces that extend the trace (i.e. the input trace is a prefix of the output trace). In addition to extending the trace, state transitions can also return a value (we will see such examples below), and sometimes return a failure (written $\bot$), for example when encountering a parsing error or a failed signature verification.

Conceptually, these actions are similar to the actions in DY* [39], rules in Tamarin's framework [30], or oracles in cryptographic game-based security definitions. They differ from the linear process descriptions in process-calculi such as applied-Pi, but can often be directly derived from such descriptions, see, e.g. Tamarin's SAPIC+ module [15].

At a technical level, protocol participant actions are functions of type $\mathsf{Traceful}\ \alpha := \mathbb{T}_e \to (\alpha \cup \{\bot\}) \times \mathbb{T}_e$. This is the type of exception and stateful monad, where the state is a trace, where furthermore the trace is only extended. This allows us to write participant actions as the (monadic) composition of several monadic sub-functions, which for example implement the network, state storage, or random bytestring generation.

Transition system. We defer to §2.5 the definition of which protocol actions are allowed to be used by the attacker.

Example: modeling a network. We first define the trace entry type corresponding to a message having been sent on the network, which is $\mathsf{MsgSent}(msg)$. Then, protocol participants can interact with the network using these two monadic functions:

- $\mathsf{sendMsg} : \mathbb{B} \to \mathsf{Traceful}\ \mathbb{N}$
- $\mathsf{recvMsg} : \mathbb{N} \to \mathsf{Traceful}\ \mathbb{B}$

The function sendMsg returns a *handle* (here, a natural number), which must then be given to recvMsg in order to retrieve the message that was sent. This notion of handle exists because possibly (and likely) in a given trace, several messages have been sent on the network, hence, which message must recvMsg return? The handle provides an answer to this question: recvMsg must return the message on which sendMsg returned this handle.

We implement the network functions as follows:

- $\mathsf{sendMsg}(msg, \tau) := (\tau.\mathsf{len}, \tau + \mathsf{MsgSent}(msg))$
- $\mathsf{recvMsg}(handle, \tau) :=$
$$\begin{cases} (msg, \tau) & \text{if } \tau[handle] = \mathsf{MsgSent}(msg) \\ (\bot, \tau) & \text{otherwise} \end{cases}$$

Note that this does not yet model the fact that the network is *untrusted* (i.e. the attacker can send messages). We will see how to do this in §2.5.

Example: modeling random generation. To model random bytestring generation, we rely on two ingredients: on the side of symbolic bytes, we define a constructor for fresh random bytestrings; on the side of execution traces, we define an entry logging that a randomness has been generated. To model *freshness* (i.e. two random bytestrings are distinct), the constructor in the symbolic bytes contain a natural number (the length the execution trace when this randomness has been generated) and the execution trace is extended with an entry so that its length increments.

In the end, we implement random generation as follows:

$$\mathsf{genRand}(\tau) := (\mathsf{RandBytes}(\tau.\mathsf{len}), \tau + \mathsf{RandGen})$$

2.4. Attacker Knowledge

Given a trace, we define the attacker knowledge in two steps: we first define the *base* attacker knowledge, which are the bytestrings that are known by the attacker because some entries are on the trace (e.g. bytestrings sent on the network, or compromised states); we then define the attacker knowledge by allowing the attacker to compute abstract functions

on inputs they know (which in particular includes all public constants, as zero-arity functions).

Base attacker knowledge. For each type of trace entry, we require the user to provide the base attacker knowledge for this entry, as a predicate

$$\mathit{baseAttKnows} : \mathbb{T}_e \to \mathbb{E}_e \to \mathbb{B} \to \mathit{Prop}$$

Then, a bytestring b is in the base attacker knowledge with respect to trace τ when

$$\exists i, \mathit{baseAttKnows}(\tau[:i], \tau[i], b)$$

For example, to express that the attacker knows any message sent on the network, we define

$$\mathit{baseAttKnows}(\tau, \mathsf{MsgSent}(\mathit{msg}), b) := \mathit{msg} = b$$

Attacker knowledge. We then define the attacker knowledge, with respect to a trace τ, as the smallest set of bytestrings such that

- it contains the base attacker knowledge (with respect to the trace τ)
- it is stable by computing abstract functions

We formally define this set as a Kleene least fixpoint. The definition of the attacker knowledge is controlled by the user, by choosing which abstract functions the attacker can compute.

Defining the attacker knowledge in Lean. To define the fact that the attacker can compute the abstract function $\overline{Vk}$ in Lean, we write a predicate which, given a predicate for the attacker knowledge `attKnows` and a bytestring `out`, tells whether `out` is the result from computing $\overline{Vk}$ from a bytestring known to the attacker.

```
def vk.attackerKnowledge: AttKnowledge where
 pred attKnows out :=
  ∃ sk, out = vk sk ∧ attKnows sk
```

2.5. Transition System and Reachable Traces

We can now define the set of reachable traces of a protocol in the presence of the attacker. At a high level, DyLean users gather the list of protocol actions (i.e. trace extending functions) that the attacker can use; then, we say that the empty trace is reachable, and we say that applying a protocol action on a reachable trace yields a reachable trace. Furthermore, applying a protocol action may come with a pre-condition: we will see for example that the attacker can send a bytestring on the network only when they know this bytestring (this prevents the attacker to "guess" secret keys). Finally, note that in our definition of reachable traces, the protocol actions are atomic: the attacker is not allowed to interleave the execution of several protocol actions, as discussed in §2.3.

Formal definition of a transition system. We define a transition of a protocol as a triplet $(\mathbb{I}, P, f)$:

- a type for inputs $\mathbb{I}$
- a precondition $P : \mathbb{I} \to \mathbb{T}_e \to \mathit{Prop}$
- a transition function $f : \mathbb{I} \to \mathsf{Traceful}\ \alpha$ for a type α

Then, we define a transition system to be a set of transition, by convention, indexed by ι: $(\mathbb{I}_\iota, P_\iota, f_\iota)$.

Example: transition for protocol step. Suppose we have a protocol step that receives a message from the network and retrieves a state. Therefore, this protocol step takes as argument two handles, a message handle, and a state handle. We want the attacker to be able to choose which message is received, and which state is retrieved, therefore we allow the attacker to choose the handles. We formally define this transition as follows:

- $\mathbb{I} := \mathbb{N} \times \mathbb{N}$ — the two handles
- $P((h_1, h_2), \tau) := \top$ — i.e. always true
- $f((h_1, h_2)) := \mathsf{step}(h_1, h_2)$

Example: sending message on the network. To model active attackers, we need to allow the attacker to send on the network any bytestring they know. Conversely we don't want the attacker to be able to send bytestrings they don't know: this is where we use the pre-condition of the transition. We model the fact that the attacker is active and can send messages with the following transition:

- $\mathbb{I} := \mathbb{B}$ — bytestrings
- $P(\mathit{msg}, \tau) := \mathcal{A}_\tau(\mathit{msg})$ — that the attacker knows
- $f(\mathit{msg}) := \mathsf{sendMsg}(\mathit{msg})$

Defining a transition in Lean. To define the transition described in the previous paragraph in Lean, it is simply a matter of filling the fields of a structure `Transition`.

```
def sendMessage.transition: Transition where
 Input := Bytes
 PreCond b tr := AttackerKnows b tr
 step b := ⟨ _, sendMessage b ⟩
```

Formal definition of reachability. Given a a transition system $(\mathbb{I}_\iota, P_\iota, f_\iota)$, we define the predicate of reachable traces (written $\mathcal{R}$) as the smallest predicate such that

- $\mathcal{R}(\varepsilon)$ (i.e. the empty trace is reachable), and
- $\forall \iota, \forall x \in \mathbb{I}_\iota, \forall \tau \in \mathbb{T}_e, \mathcal{R}(\tau) \wedge P_\iota(x, \tau) \Rightarrow \mathcal{R}(f_\iota(x, \tau))$

2.6. Discussion

In this section, we have seen all the components of DyLean that allow users to specify symbolic semantics and cryptographic protocols.

Library of components. If we were to only provide to the user a framework for symbolic semantics, this would leave a lot of work to users, and likely a lot of duplication across models. To this end, in addition to our symbolic semantics framework, we provide a *library* of equational theories and state transitions. For example, DyLean provides semantics to implement an attacker-controlled network, state storage, state compromise, and Public-Key Infrastructure (PKI). On the side of equational theories, DyLean provides literals, concatenation, Diffie-Hellman, hashes, signatures. Unlike in DY*, users are not restricted to only use what DyLean provides:

should a component be missing, users have the ability to add this component themselves. We explore this in our case studies: in the case study described in §4.4, we add a new equational theory for signatures and explore its impact on a protocol; in the case study described in §4.5, we add support for KEMs, which we did not yet integrate in DyLean's library of equational theories.

Proving executability theorems. With our framework, we can already prove statements of the form "there exists a reachable trace such that …". Indeed, it suffices to exhibit such a trace, prove that it is reachable (which is proved by definition of reachability), and prove that the trace satisfies the properties we want (which can be proved in Lean by *evaluating* the property over this concrete trace). Such executability theorems can be used as sanity checks, for example to ensure that there exists runs of the protocol that succeed, but can also be used to showcase attacks on protocols, as we will do in §4.4.

Proving security theorems. With our framework, we can *state* security theorems of the form "every reachable trace must satisfy …". However, we do not yet have the tools to *prove* these security theorems: this is the goal of the next section.

3. Proving Protocols Secure in DyLean

In §2, we saw how DyLean allows specifying cryptographic protocols and their expected security properties, in a way that is agnostic on the methodology used to prove the security properties. In this section, we develop a proof methodology for DyLean, which is inspired by DY*. The soundness of our proof methodology is guaranteed by Lean [21]: the security theorems are stated using the tools we provided in §2, and our proof methodology is just a means to convince Lean that the security theorem is true.

3.1. Background

We aim to prove *reachability properties*, that is, security properties of the form "all reachable traces must satisfy this property".

Trace invariant. DY* proves reachability properties by requiring the user to supply a *trace invariant*, such that (1) every protocol action preserves the trace invariant, which proves that every reachable trace must satisfy the trace invariant, and such that (2) the trace invariant implies the security properties. In the end, this proves that every reachable trace must satisfy the security property.

To avoid users having to provide a full-fledged trace invariant for each protocol analysis, DY* provides a blueprint for the trace invariant. For example, this blueprint lets the user provide the *state invariant*: on the one hand, when protocol participants store state, this action preserves the trace invariant only when the state invariant holds on the stored state; on the other hand, when protocol participants retrieve state, the trace invariant guarantees that the state invariant must hold on the retrieved state. The trace invariant blueprint in DY* also imposes that every bytestring on the network is *publishable* (we will this explain this notion in the next paragraph), that is, sending a bytestring on the network preserves the trace invariant only when the message is publishable; conversely, receiving a message on the network yields a publishable bytestring.

Publishability. One of the properties we aim to prove are *confidentiality properties*, which are of the form "if the attacker knows b, then the attacker must have done X", where typically X describes that the attacker compromised some secrets. In other words, confidentiality properties are about over-approximating the attacker knowledge on some particular bytestring b we care about. This over-approximation is captured by the notion of *publishability*: we maintain the invariant that for any bytestring b, if the attacker knows b, then b must be publishable. Conversely, if b is publishable, we consider b to be "safe to reveal to the attacker": indeed, doing so would preserve the fact that publishability is an over-approximation of the attacker knowledge. The notion of publishability is defined as the conjunction of two other components: *security labels*, which capture confidentiality, and the *bytes invariant*, which captures authentication.

Security labels for confidentiality. DY* associates each bytestring with a *security label* (a notion inspired from Information Flow Control), which is a predicate on traces that describes the compromise scenarios that may cause a bytestring to be known to the attacker. When the corresponding compromise scenario has happened, we say that the label is "corrupt". For each cryptographic function, DY* defines the security label of the output as a function of the labels of its inputs. For example, the label of a signature is defined to be equal to the label of the signed message, that is, from the viewpoint of labels, the message and its signature are considered to be "safe to reveal to the attacker" at the same time: this capture the fact that signatures may reveal information about the signed content.

Bytes invariant for authentication. We note that security labels do not account for authentication properties of signatures: indeed, from the viewpoint of labels, it would be safe to expose a signature oracle to the attacker, which would nullify the authentication guarantees provided by signatures.

To account for authentication guarantees provided by cryptographic functions such as signatures, DY* also defines the *bytes invariant* (which derives its name from the fact that it holds on any bytestring appearing in a protocol execution). For example, the bytes invariant for signatures depends on a user-provided predicate, called the *signature predicate*. The bytes invariant on a signature is defined to hold when either (1) the (private) signature key is publishable, or (2) the signature predicate holds on the signed message. This guarantees, upon signature verification, that either (1) the signature was computed by the attacker, in which case the corresponding signature key is publishable, or (2) the signature was com-

puted by an honest participant, in which case the signature predicate must hold on the signed message. The user-provided signature predicate is able to mention the label of bytestrings present in the message, and as such it can be used to transmit information about the confidentiality of bytestrings.

Generating random bytestring and ghost state. Suppose a participant generates a fresh bytestring b, and then sends b on the network. Should this action preserve the trace invariant? We give a partial answer to this question: generating fresh randomness always preserves the trace invariant, and we have seen that sending b on the network preserves the trace invariant exactly when b is publishable. Hence, the question boils down to: should b be publishable? Recall that b is publishable when it satisfies the bytes invariant and when its security label is corrupt: since b (as a freshly generated random bytestring) always satisfies the bytes invariant, the initial question then reduces to: suppose a participant generates a fresh bytestring b, what is the security label of b? If b is sent on the network (as in the first scenario), and then forgotten and never used again, then surely this is a safe action, and the label of b can be the public label (i.e. always corrupt), so that b is publishable. However, if the destiny of b is to be a long-term secret key, then surely sending b on the network is not a safe action, and its security label should reflect that b can be known to the attacker only if, e.g. the attacker compromised the participant. Therefore, we see that the label of a freshly generated bytestring b is a choice to be made for each randomly generated bytestring, and this choice impacts the rest of the trace invariant preservation proof (as we have seen in the example of sending b on the network).

In DY* [12, 39], this choice is made in the specification: the function that generates random bytestring mk_rand takes as an additional argument: the security label for the bytestring it will output. Internally, mk_rand controls the label of the generated bytestring by storing the label in the trace, as part of the RandGen trace entry. This label is what is called a *ghost state* in the literature (see e.g. [28]): it is data provided by the security proof, stored in the trace, but has no impact on the semantics of the specification. As noted in [39], this label is an artifact from the security proofs, although it is present in the protocol specifications. In §3.2, we will see how this is not the case in DyLean, which makes specifications written in DyLean completely free of any proof artifacts, which is a requirement for DyLean's goal to be agnostic of the proof methodology.

3.2. Proof trace and trace invariant

In §3.1, we discussed the proof methodology of DY*, based on trace invariant. In this section, we explain the DyLean framework to define trace invariants, and defer linking the trace invariant to reachable traces in §3.3.

Proof traces and ghost state. We discussed ghost state in §3.1, which allows the proof to make choices depending on the execution of the protocol. Ghost state is a crucial feature for random bytestring generation, and allows the proof to choose the security label of each freshly generated random bytestring. Since the label is a proof artifact, it is not part of the execution trace (described in §2.3). Instead, such ghost state belongs to what we call the *proof trace*, which is an execution trace augmented with ghost state. Given a proof trace, we can *erase* its ghost state, and obtain the corresponding execution trace.

Notation. Before explaining the proof trace and the trace invariant, we introduce further notation. We write $\mathbb{E}_p$ the set of proof entries, $\mathbb{T}_p := \mathbb{E}_p^*$ for the set of proof traces, $\mathcal{E}(\tau_p)$ for erasing of the ghost state from proof trace τ_p, $\mathcal{I}(\tau_p)$ when τ_p satisfies the trace invariant, and $\mathcal{P}_{\tau_p}(b)$ when b is publishable with respect to trace τ_p.

Proof trace entries. Similarly to how execution traces are defined as a list of (execution) trace entries (§2.3), a proof trace is a list of proof trace entries: for each type of execution entry, the user must define a corresponding proof entry, which may additionally contains some potential ghost state, and provide an erasing function from the proof entry to the execution entry. For example, we have seen (§2.3) that the execution entry to represent that a bytestring b has been sent on the network is $\mathsf{MsgSent}(b)$, and because we don't need any ghost state related to the network, the corresponding proof entry is also $\mathsf{MsgSent}(b)$. In the case of random generation, we do need to store a label in a ghost state (as explained in §3.1). Hence, to the execution entry RandGen, we associate the proof entry $\mathsf{ProofRandGen}(\ell)$, where ℓ is the label of the bytestring.

Trace invariant. For each proof trace entry, the user supplies an invariant corresponding to this trace entry, *entryInvariant* : $\mathbb{T}_p \to \mathbb{E}_p \to$ *Prop*. We then define the trace invariant as:

$$\mathcal{I}(\tau_p) := \forall i, \mathit{entryInvariant}(\tau_p[:i], \tau_p[i]).$$

For example, the entry invariant for $\mathsf{MsgSent}(b)$ is defined as $\mathit{entryInvariant}(\tau_p, \mathsf{MsgSent}(b)) := \mathcal{P}_{\tau_p}(b)$. This guarantees that $\mathsf{sendMsg}(b)$ (defined in §2.3) preserves the trace invariant only when b is publishable, and that $\mathsf{recvMsg}(\mathit{handle})$ returns a publishable message. The entry invariant for $\mathsf{ProofRandGen}(\ell)$ is always true, so that the function genRandalways preserve the trace invariant. The interesting bits of invariants related to random generation will be in the *bytes invariant* of the random bytestring. We will define the bytes invariant in §3.4.

Trace invariant preservation. We next expand on what we mean by "a function preserves the trace invariant". First, note that saying that this is an abuse of language, because functions append entries on the *execution* trace, whereas the trace invariant holds on a *proof* trace; we will make this precise in the next paragraph. Second, functions may not always preserve the trace invariant: they may preserve the trace invariant *under some condition* (for example, sending a message on the network preserves the trace invariant *only when* the message is publishable), this is also known as a *precondition*. Finally, knowing that the trace invariant holds after the execution of a function may not be precise enough, and we may like to guarantee some additional property (for example, that receiving a message preserves the trace invariant *and* yields a message that is publishable), this is also known as a

post-condition. We now have all the ingredients to formally define what it means to preserve the trace invariant, using Hoare Triples.

Hoare Triples. We formally state that a function preserves the trace invariant under some pre-condition and post-condition using a Hoare Triple [26]. In plain English, a Hoare Triple $\{pre\}f\{post\}$ means that for any proof trace τ_p that satisfies the trace invariant and the precondition, when running the function f on the corresponding execution trace $\mathcal{E}(\tau_p)$, which yields an output *res* and an (extended) execution trace τ'_e, then there exists a proof trace τ'_p compatible both with τ'_e (in that τ'_p erases to τ'_e) and with τ_p (in that τ'_p extends τ_p, that is, does not change ghost state already present in τ_p); furthermore, τ'_p satisfies the trace invariant, and the post-condition holds on *res* and τ'_p.

Example: Hoare Triple for network functions. For example, the function sending a message on the network has the Hoare Triple

$$\{\lambda\tau_p.\mathcal{P}_{\tau_p}(b)\}\mathsf{sendMsg}(b)\{\lambda_, _.\top\}$$

and the receiving message function has the Hoare Triple

$$\{\lambda_.\top\}\mathsf{recvMsg}(\mathit{handle})\{\lambda b, \tau_p.\mathcal{P}_{\tau_p}(b)\}.$$

Example: Hoare Triple for randomness. More interestingly, the function to generate fresh randomness has a *family* of Hoare Triples, one for each possible label ℓ of the generated bytestring:

$$\{\lambda_.\top\}\mathsf{genRand}\{\lambda b, \tau_p.\mathcal{L}_{\tau_p}(b) = \ell\}$$

Indeed, for each label ℓ, we can append $\mathsf{ProofRandGen}(\ell)$ to the trace τ_p and obtain a valid τ'_p that satisfies the trace invariant and post-condition. By choosing which Hoare Triple to use when conducting the proof, in effect, the proof chooses the label ℓ of the generated bytestring.

Proving a Hoare Triple. In a typical protocol specification, a protocol action is a monadic function (as explained in §2.3), which is written using other monadic functions as building block. Suppose we want to prove that this function preserves the trace invariant, that is, we want to prove a Hoare Triple on this function. To achieve this, we provide a Lean tactic called `step` that perform Verification Condition Generation: our tactic steps though the function, fetches the Hoare Triple for each function used inside our function, asks the user to provide ghost state and to prove pre-conditions. Then, to prove the pre-conditions, users can rely on the built-in tactic `grind`, which is inspired by SMT solvers and automatically construct proofs, and is in practice sufficient to prove most of the pre-conditions.

A tactic built into Lean, named `mvcgen`, aims to be a generic framework to prove Hoare Triples, whereas our tactic `step` only works for our specific Hoare Triples. When we started working on DyLean, the tactic `mvcgen` was still early development, and was missing several features we needed. Now that `mvcgen` is becoming more mature, and that its foundations are seeing a revamp based on Loom [24], we intend to deprecate `step` and migrate to `mvcgen`. To make this migration possible, we are currently in active discussion with the Lean developers, in order to integrate the last features we are missing into `mvcgen`.

Discussion: no higher-order ghost state. Note that the ghost state can contain predicates on execution traces (such as labels), but cannot contain predicates on proof traces: in other words, the ghost state is not higher-order, it is only second-order. This is because it is impossible to have higher-order ghost state for free: otherwise, this would allow a version of the liar paradox, by storing a predicate on the ghost state inside the ghost state, defined as the negation of itself. Program logics that allow for higher-order ghost state, such as Iris [28], use step indexing [4] to avoid paradoxes, which imposes to reason on the running time of programs. In practice, second-order ghost state is expressive enough for our proofs: although it imposes to choose predicates on *proof* traces upfront (such as the signature predicate), we have not yet seen a protocol where such a predicate needs to be chosen during the execution of the protocol (and stored into a ghost state).

3.3. The key reachability theorem

In §3.2 we defined the trace invariant and how functions preserve the trace invariant. We now explain how DyLean links the trace invariant to reachable traces.

The reachability theorem. The key property of this proof methodology is the following: for any execution trace that is reachable, there exists a corresponding proof trace that satisfies trace invariant, i.e.

$$\forall\tau_e \in \mathbb{T}_e, \mathcal{R}(\tau_e) \implies \exists\tau_p \in \mathbb{T}_p, \mathcal{I}(\tau_p) \wedge \mathcal{E}(\tau_p) = \tau_e.$$

Then, if a user proves that any proof trace satisfying the trace invariant implies that its erasing satisfies a security property $\mathcal{P}$, i.e.

$$\forall\tau_p \in \mathbb{T}_p, \mathcal{I}(\tau_p) \implies \mathcal{P}(\mathcal{E}(\tau_p)).$$

Combining the two theorems concludes that any reachable trace must satisfy the security property, i.e.

$$\forall\tau_e \in \mathbb{T}_e, \mathcal{R}(\tau_e) \implies \mathcal{P}(\tau_e),$$

which proves that the security property holds on the protocol. As such, the trace invariant is only a proof technique, and its precise definition does not need to be trusted nor audited.

Proving the reachability theorem. We have seen in §2.5 that the reachability predicate $\mathcal{R}$ can be configured by the user. Therefore, the reachability theorem relies on the crucial hypothesis (which the user must prove) that any transition must preserve the trace invariant. More precisely, for each transition $(\mathbb{I}, P, f)$ (see §2.5), for each input $x \in \mathbb{I}$ the user must supply a Hoare Triple $\{P(x)\}f(x)\{\lambda_.\top\}$ Then, by induction on reachable traces, we easily prove the reachability theorem.

Example: sending a message on the network. In §2.5, we have seen how to model an active attacker that is able to send message on the network. We did this by having in the transition system the function $\mathsf{sendMsg}(msg)$ with the pre-condition $\mathcal{A}_\tau(msg)$ (i.e. the attacker must know *msg* with respect to the current trace τ). Therefore, to prove the reachability theorem, we must prove the Hoare Triple

$$\{\lambda\tau_p.\mathcal{A}_{\mathcal{E}(\tau_p)}(msg)\}\mathsf{sendMsg}(msg)\{\lambda_,_.\top\}.$$

However, in §3.2, we have seen that the Hoare Triple for sending a message instead relies on *publishability*:

$$\{\lambda\tau_p.\mathcal{P}_{\tau_p}(msg)\}\mathsf{sendMsg}(msg)\{\lambda_,_.\top\}.$$

We prove the first Hoare Triple using the second one, by leveraging the key attacker knowledge theorem, which states that any bytestring known by the attacker must be publishable, and which we will fully describe in §3.4.

3.4. The key attacker knowledge theorem

In §3.1, we discussed the proof methodology of DY*, in which the attacker knowledge is over-approximated by the notion of *publishability*, also known as the *key attacker knowledge theorem*. We now formally state this theorem, and explain how it is proved.

The attacker knowledge theorem. On the side of bytestrings, the key property of our proof methodology is that on traces that satisfy the trace invariant, any bytestring known by the attacker must be publishable, i.e.

$$\forall\tau_p \in \mathbb{T}_p, \forall b \in \mathbb{B}, \mathcal{I}(\tau_p) \wedge \mathcal{A}_{\mathcal{E}(\tau_p)}(b) \implies \mathcal{P}_{\tau_p}(b)$$

Using the attacker knowledge. Suppose we have a secret key k (e.g. a key that is established by a key exchange protocol), and suppose we can prove that if k is publishable (i.e. $\mathcal{P}_{\tau_p}(k)$) then the attacker must have some specific compromises (written $C(\mathcal{E}(\tau_p))$). We can then combine the key attacker knowledge theorem with the key reachability theorem §3.3 and deduce that for any reachable trace, if the attacker knows k, then this compromise scenario must have happened, i.e.

$$\forall\tau_e \in \mathbb{T}_e, \mathcal{R}(\tau_e) \wedge \mathcal{A}_{\tau_e}(k) \implies C(\tau_e).$$

Proving the attacker knowledge theorem. Recall (§2.4) that the attacker knowledge, with respect to an execution trace τ_e, is defined as the smallest set of bytestrings that (1) contains the *base attacker knowledge* (with respect to τ_e), and (2) is stable by computing abstract functions. Because the attacker knowledge is fully controlled by the DyLean user, from the base attacker knowledge to the set of abstract functions the attacker can compute, in order to prove the attacker knowledge theorem, we require some hypothesis to be provided by the user. We require from the user (1) that for any *proof* trace τ_p that satisfies the trace invariant (i.e. $\mathcal{I}(\tau_p)$), a bytestring in the base attacker knowledge (with respect to $\mathcal{E}(\tau_p)$) must be publishable (with respect to τ_p) and (2) that computing abstract functions on publishable bytestring also yields a publishable bytestring. Then, because the attacker knowledge defined to be the *smallest* set satisfying requirements (1) and (2), we conclude that for any bytestring b, if the attacker knows b with respect to $\mathcal{E}(\tau_p)$ then b is publishable with respect to τ_p.

Example: messages sent on the network. We have seen in §2.4 that messages sent on the network are in the base attacker knowledge. Hence, one requirement of the attacker knowledge theorem is that for all (proof) trace that satisfies the trace invariant, any message sent on the network (with respect to this trace) must be publishable. Thankfully, this is exactly what is enforced by the trace invariant, as we have seen in §3.2.

Defining the publishability predicate. Because the precise definition of bytestrings is parametrized by the user, so must be the publishability predicate. For every bytestring constructor, the user must supply a few functions, of which the security label, and the bytes invariant. Because our definition of publishability closely follows the one in DY*, we refer the reader to [37, Chapter 2] for a detailed explanation.

4. Case Studies

We apply DyLean to several case studies, which we summarize in Table 4. Each case study aims to exercise DyLean in a particular axis, showcasing that DyLean can effectively be used to analyze protocols with a variety of features. Each case study comprises the protocol specification, sanity checks, security properties, and machine-checked proofs that these security properties indeed hold on the specified protocol. We provide all models in our artifact [2].

4.1. DyLean on simple key exchanges

Before testing that DyLean can effectively analyze protocols exhibiting complex features, we first test it in on a standard key exchange protocol. For this, we choose the unilateral signed Diffie-Hellman key exchange, from the Protocol Proof Ladder project [1].

Protocol flow. In this protocol, only the server authenticates itself to the client (hence "unilateral"), using a long-term signature key. The client generates an ephemeral Diffie-Hellman keypair (x_{sk}, x_{pk}), sends x_{pk} to the server. The server generates an ephemeral Diffie-Hellman keypair (y_{sk}, y_{pk}), signs x_{pk} and y_{pk}, sends y_{pk} and the signature to the client, and establishes and stores the key $\mathrm{Hash}((x_{pk})^{y_{sk}})$. Finally, the client checks the signature, and establishes and stores the key $\mathrm{Hash}((y_{pk})^{x_{sk}})$.

Threat model. We consider an attacker who can compromise the ephemeral state of both the client and the server, as well as the long-term signature key of the server.

Feature	DY*	Example	LoC	Build time	Person-time
Key exchange (§4.1)	●	Signed-DH	1.0k	7s	≈ 1 day
Complex datastructures (§4.2)	●	Merkle tree	1.1k	10s	≈ 2 days
Unbounded loops (§4.3)	●	Ratcheting	2.2k	30s	≈ 3 days
Equational theories (§4.4)	○	ACME draft 0	1.1k	12s	≈ 1 day
Threat models (§4.5)	○	Signed-DH+KEM	3.3k	22s	≈ 1 day

Table 1: **Overview of case studies.** For each case study, we mention whether DY* supports the feature required for this case study (●) or not (○). Lines of Code (LoC) covers specification, security properties, sanity checks, proofs, and potentially equational theories not part of DyLean's standard library. Build time is measured on an off-the-shelf laptop (ThinkPad X1 Carbon Gen 13), and building DyLean and the case studies requires a maximum of 2GB of memory. Person-time is an estimate of the number of working days we took to do this case study. Currently our files contain a high ratio of boilerplate code, and we expect to reduce this significantly in next versions.

Security properties. We prove that this protocol satisfies authentication and confidentiality properties. Our properties are stated from the side of the client: indeed, because the client is not authenticated, the protocol enjoys no such properties on the side of the server.

Our authentication theorem is stated as follows: if the client finishes a run of the protocol with the ephemeral keys x_{pk}, y_{pk}, and the established key k, then either (1) the server also finished a run of the protocol with x_{pk}, y_{pk} and k, or (2) the long-term signature key of the server was compromised before the client finished the run of the protocol.

Our confidentiality theorem is stated as follows: if the client finishes a run of the protocol with the ephemeral keys x_{pk}, y_{pk}, and the established key k, and the attacker knows the key k, then either (1) the client ephemeral state containing x_{sk} was compromised, or (2) the server final state containing k was compromised, or (3) the server long-term signature key was compromised *before* the client finished the run of the protocol.

Sanity checks. All our security theorems are of the form "for all reachable traces where the client finishes a run of the protocol, then…" Therefore, it is important to check that such traces exist: indeed, it could be that the protocol specification would contain mistakes, and for example, would lead the client to always reject the signature verification, leading to no such trace existing. We prove a few sanity check theorems. We first prove that an honest run of the protocol leads the client and server to succeed in establishing the same key, and yields a reachable trace. We then prove that an honest run of the protocol followed by a compromise of client ephemeral key leads the attacker to know the established key, and yields a reachable trace. We finally prove that if the attacker compromises the server long-term signature key and then performs a machine-in-the-middle attack, this leads the client to successfully finish a protocol run and establishing a key that the attacker know, furthermore this trace is reachable.

These sanity checks allow us to gain confidence that our protocol model and security property make sense, for example ensuring that our security theorems are not simply universal quantification on the empty set. Furthermore, the sanity checks involving compromises allow us to confirm that we did not forget to include compromise capabilities in our modeling of the attacker.

4.2. DyLean and complex datastructures

To show that DyLean can effectively analyze protocols involving complex datastructures, we analyze an authentication protocol based on Merkle Trees.

Protocol flow. In this protocol, the server receives a list of bytestrings from the network (i.e. the attacker), and computes the Merkle tree hash of this list (in a similar fashion to Certificate Transparency [29, §2.1.1]), then uses its long-term signature key to sign this Merkle tree root hash, and stores this list to later prove inclusion of an element in this list. Then, the server can be asked to prove inclusion of an element in the list, and outputs a Merkle inclusion proof. Finally, the client receives a bytestring, an inclusion proof, and a signed Merkle tree root hash, and accepts this bytestring if both the signature and the inclusion proof are valid.

Threat model. We consider an attacker who can compromise the long-term signature key of the server.

Security properties. We prove an authentication property: if the client accepts a bytestring, then either (1) this bytestring indeed belongs to the list processed by the server, or (2) the long-term signature key of the server was compromised *before* the client accepted this bytestring.

Sanity checks. We prove that an honest run of the protocol on a list of five elements, and inclusion proof on the fourth element leads the client to accept this fourth bytestring. We furthermore prove that if the attacker compromises the long-term signature key of the server, the attacker can perform a machine-in-the-middle attack and lead the client to accept a bytestring.

4.3. DyLean on looping protocols

To show that DyLean can effectively analyze protocols that involve unbounded loops we analyze a ratcheting protocol, and prove forward secrecy and post-compromise secrecy.

Protocol flow. The protocol involves two participants, which will derive a stream of *epoch keys*. At each step of the protocol, participants alternately generate and send a Diffie-Hellman public key over the network, compute the shared secret between their latest Diffie-Hellman keys, and using a Key Derivation Function (KDF), compute the new epoch key by mixing the previous epoch key with this Diffie-Hellman shared secret. At each step, the two participants maintain a transcript hash: the transcript hash of an epoch is computed by hashing the previous epoch transcript hash and the new Diffie-Hellman key transmitted on this epoch. The transcript hash is signed and sent alongside the Diffie-Hellman public key, and is also mixed into the epoch key with the KDF. To initialize the protocol, participants start with an all-zero key, and an empty transcript hash.

This ratcheting protocol is broadly inspired by Signal's Diffie-Hellman ratchet [32] for alternately sending Diffie-Hellman public keys and combining them with a KDF, and is also inspired by MLS [8] for the transcript hash, which is both signed and injected in the epoch key.

Threat model. We consider an attacker who can compromise the ephemeral state (containing the epoch key and Diffie-Hellman private key) and the long-term signature key of protocol participants.

Security properties. We prove that this ratcheting protocol satisfies authentication and confidentiality properties. We identify a specific epoch of a session between two participants by its *transcript:* the list of Diffie-Hellman public keys that led to this epoch.

Our authentication theorem is stated as follows: if a participant P is in an epoch with recipient R, transcript t, and epoch key k, and is in this epoch after receiving an update, then either (1) R has sent an update which led to an epoch with the same transcript t and established the same key k, or (2) the long-term signature key of R was compromised before P has processed the update to this epoch.

For confidentiality, we prove both forward secrecy and post-compromise security. We provide two equivalent versions of the confidentiality theorem: one which is defined recursively, the other which is unrolled. Although equivalent, each version might provide different intuitions. Following Python convention, we write $t[:-1]$ for the transcript t without its last element (i.e. the transcript of the previous step).

The recursive confidentiality theorem is stated as follows: if a participant P is in an epoch with recipient R, transcript t, and epoch key k, and the attacker knows k, then a predicate $\mathcal{C}(P, R, t)$ holds. The predicate $\mathcal{C}(P, R, t)$ is true when either (1) $t.\mathsf{len} \leq 1$, or (2) the attacker compromised the ephemeral state of P or R for the transcript t or $t[:-1]$, or (3) $\mathcal{C}(P, R, t[:-1])$ holds, and the attacker compromised the long-term signature key of R before P last accepted an update of R. Stating a confidentiality theorem in a recursive fashion is similar to prior analysis of the TreeKEM sub-protocol of MLS [38].

The unrolled confidentiality theorem is stated as follows: if a participant P is in an epoch with recipient R, transcript t, and epoch key k, and the attacker knows k, then either (1) $t.\mathsf{len} \leq 1$, or (2) the attacker compromised the ephemeral state of P or R for the transcript t or $t[:-1]$, or (3) there exists a prefix t_{prev} of the transcript t such that the attacker compromised the long-term signature key of R before P moved to the epoch corresponding to t_{prev} and either $t_{\mathsf{prev}}.\mathsf{len} \leq 2$ or the attacker compromised the ephemeral state of P for the transcript t_{prev} or $t_{\mathsf{prev}}[:-1]$, or of R for the transcript $t_{\mathsf{prev}}[:-1]$ or $t_{\mathsf{prev}}[:-2]$.

The confidentiality theorems imply forward secrecy because the attacker must have performed a compromise in the past. The confidentiality theorems also imply post-compromise security, because the attacker must have performed a compromise recently enough (or in the case of (3), recently before having compromised the long-term signature key).

Sanity checks. We prove that an honest run of the protocol for 20 steps leads two participants to establish the same key, and yields a reachable trace. We then prove that an honest run of the protocol followed by a compromise of an ephemeral state leads the attacker to know the established key, and yields a reachable trace. We finally prove that if the attacker compromises the a participant long-term signature key and then performs a machine-in-the-middle attack for 20 steps of the protocol, this leads participant in establishing a key that the attacker knows, furthermore this trace is reachable.

4.4. DyLean and custom equational theories

To show how DyLean supports custom equational theories, we analyze the draft 0 of the ACME protocol [9] under different equational theories. A particularity of this protocol is the following: it can be shown to be secure in the symbolic model, under the standard equational theory for signatures; however, there also exists an attack on this protocol when instantiated with signature schemes that fails to provide *Destructive Exclusive Ownership* (DEO) [27]. In such signature schemes, the fact that this signature scheme is unforgeable do not entail that signatures bind to the message being verified nor to to the verificication key used to verify it. It is possible to model signatures that do not satisfy DEO by having more precise equational theory for signatures [27]. We analyze ACME draft 0 [9] under two equational theories: with signatures that satisfy DEO, we prove the protocol secure, with signatures that do not satisfy DEO, we prove that the protocol is insecure, i.e. we showcase an attack.

Note that such an analysis would be impossible in DY*: indeed, DY* comes with a fixed equational theory which contains the standard equational theory for signatures (which are signatures that satisfy DEO). Therefore, DY* can only prove that the protocol is secure, but cannot showcase an attack when the signature scheme do not satisfy DEO.

Protocol flow. The protocol involves three parties: a Server Owner, Let's Encrypt, and a DNS server. The Server Owner aims to prove to Let's Encrypt that they are in control of the DNS record for a given domain name. Let's Encrypt generates

a token (a fresh random bytestring), sends it to the Server Owner. The Server Owner signs it, updates the DNS records for their domain name to contain the signature, then Let's Encrypt retrieves the signature from the DNS servers and verifies it.

Threat model. We consider that the communication with the DNS server is authenticated: only owners of the domain name can change their records, and the result of fetching the records cannot be tampered by the attacker. We consider an attacker who can compromise signature keys. On signatures, we analyze the protocol under two threat models: on signatures that satisfy DEO, and on signatures that do not satisfy DEO.

Methodology. Our protocol specification is parametric on the type of symbolic bytes, and requires (using a typeclass) that it must be possible to sign and concatenate bytestrings, as well as generating fresh randomness and containing literals. We can then instantiate this parameter and typeclass with symbolic bytes containing signatures that satisfy DEO, and do a security proof; and instantiate these with symbolic bytes containing signatures that do not satisfy DEO, and showcase an attack.

Security property. When the signature scheme satisfies DEO, we prove that if Let's Encrypt accepts a run of the protocol on a domain name and Server Owner signature public key, then the Server Owner indeed initiate a run of the protocol with the same parameters.

Sanity check. We implement an honest run of the protocol, show that this yields a reachable trace where the Server Owner and Let's Encrypt accept a run of the protocol with the same parameters.

Attack. When the signature scheme does not satisfy DEO, we implement the attack described in [27], show that this yields a reachable trace where Let's Encrypt accepts a run of the protocol on a domain name and Server Owner signature public key, but the Server Owner never initiated a run of the protocol with these parameters.

4.5. DyLean and custom threat models

To show how DyLean supports custom threat models, we analyze a post-quantum protocol against an attacker that can break Diffie-Hellman and signatures (in the event they get access to a quantum computer), and that can break KEM (in the event it is discovered to be insecure). For this, we choose the hybridization of the signed Diffie-Hellman protocol which we analyzed in §4.1, which we also took from the Protocol Proof Ladder project [1].

Note that such an analysis would be impossible in DY*: indeed, DY* comes with a fixed set of invariants for signatures, Diffie-Hellman, and KEMs. These invariants are designed considering it is impossible for an attacker break these primitives: to consider an attacker with such capabilities, one must change these invariants, which DY* does not allow because invariants are hardcoded.

Protocol flow. This protocol is similar to the unilateral signed Diffie-Hellman protocol we analyzed in §4.1, with a KEM added in the mix. The client generates an ephemeral Diffie-Hellman keypair (x_{sk}, x_{pk}) and a KEM keypair (z_{sk}, z_{pk}), then sends x_{pk} and z_{pk} to the server. The server generates an ephemeral Diffie-Hellman keypair (y_{sk}, y_{pk}), encapsulates a shared secret ss to z_{pk} yielding a ciphertext ct, then signs x_{pk}, y_{pk}, z_{pk} and ct, sends y_{pk}, ct and the signature to the client, and establishes and stores the key $\mathrm{Hash}((x_{pk})^{y_{sk}} \| ss)$. Finally, the client checks the signature, decapsulates ct to obtain ss and establishes and stores the key $\mathrm{Hash}((y_{pk})^{x_{sk}} \| ss)$.

Threat model. We consider an attacker who can compromise the ephemeral state of both the client and the server, as well as the long-term signature key of the server. In addition, we consider that the attacker can compute the secret key corresponding to a public key for Diffie-Hellman, signature, and KEM.

Security properties. We prove similar properties as the unilateral signed Diffie-Hellman protocol §4.1, but account for the additional KEM, as well as the possibility that the attacker may break cryptographic functions.

Our authentication theorem is stated as follows: if the client finishes a run of the protocol with the ephemeral keys x_{pk}, y_{pk}, z_{pk}, and the established key k, then either (1) the server also finished a run of the protocol with x_{pk}, y_{pk}, z_{pk} and k, or (2) the long-term signature key of the server was (2.1) compromised or (2.2) broken before the client finished the run of the protocol.

Our confidentiality theorem is stated as follows: if the client finishes a run of the protocol with the ephemeral keys x_{pk}, y_{pk}, z_{pk}, and the established key k, and the attacker knows the key k, then either (1) the client ephemeral state containing x_{sk} and z_{pk} was compromised, or (2) the server final state containing k was compromised, or (3) the attacker broke the KEM key z_{pk} and one of the Diffie-Hellman key x_{pk} or y_{pk} or (4) the server long-term signature key was (4.1) compromised or (4.2) broken *before* the client finished the run of the protocol.

Sanity checks. We prove a few sanity check theorems. We first prove that an honest run of the protocol leads the client and server to succeed in establishing the same key, and yields a reachable trace. We then prove that an honest run of the protocol followed by a breaking x_{pk} and z_{pk} leads the attacker to know the established key, and yields a reachable trace. We finally prove that if the attacker breaks the server long-term signature key and then performs a machine-in-the-middle attack, this leads the client to successfully finish a protocol run and establishing a key that the attacker know, furthermore this trace is reachable.

These sanity checks allow us to confirm that the specification of our protocol indeed includes the threat model we intended, that is, the attacker can compute the secret key corresponding to a public key for Diffie-Hellman, signature, and KEM.

5. Related Work

Historical work on symbolic analysis in interactive theorem provers. Before the advent of effective semi-automated tools for unbounded analysis, several works explored the use of interactive theorem provers, e.g. [31, 11, 40]. These early attempts could handle small abstract models, but did not allow for, e.g., compound keys (keys were considered atomic values, which mismatches modern designs that compute symmetric keys by applying a key derivation to secrets, labels, and binding information) or equational theories (cryptographic primitives were modeled extremely coarsely), and required extensive manual input. With the appearance of modern semi-automated tools with more expressivity, these methods became less prominent.

Semi-automated symbolic tools. Modern tools allow for semi-automated symbolic analysis of cryptographic protocols. Leading tools are Tamarin [30] and ProVerif [13].

On the side of symbolic semantics, like us, Tamarin and ProVerif allow for user-defined equational theories, which are restricted to equational theories with the Finite Variant Property modulo AC [14, 22], which, in particular, are AC-convergent. The requirement imposed by DyLean is that equational theories must allow a *normalization function* (see §2.2), which we claim is fulfilled by any AC-convergent equational theory (although a formal proof of this statement remains future work).

On the side of proofs, our flagship proof methodology presented in §3 is inspired by DY* [12, 39], and as such, inherits its advantages. In particular, our case studies (§4) have shown that our proof methodology work well on protocols involving complex datastructures. We also inherit the techniques to compose security proofs from [39].

Symbolic tools embedded in proof assistants. Other tools use similar invariant-based proofs, such as DY* [12] and DY* Unchained [39] which are embedded in F* [33], as well as Cryptis [6] which is embedded in Rocq [35], and a tool developed by Arquint et al. [5] embedded in Gobra [41].

DyLean lifts a common limitation of these tools: their symbolic semantics are hard-coded in the tool, hence equational theories or threat models cannot be extended by users.

Our proof methodology is closest to DY* Unchained [39], by which it is inspired. As such, we benefit from the advantages of DY* Unchained [39] over DY* [12], Cryptis [6] and Arquint et al. [5], in that we allow composing security proofs in a variety of scenarios by composing trace invariants.

Of these tools, Cryptis [6] is the only one whose specifications are in principle agnostic of the specific proof methodology (like DyLean). Similarly to us, they achieve this by leveraging a ghost state (as explained in §3.1) upon random bytestring generation. Whereas we use ghost state to choose and store a security label (a predicate on execution traces), Cryptis allows to choose and store the publishability predicate (a predicate on proof traces). They achieve this using *higher-order ghost state* (a feature inherited from Iris [28]), whereas we only support *second-order ghost state* (i.e. the ghost state stored in proof traces can contain execution trace predicate). Although one may think higher-order is strictly better than second-order, this is actually a trade-off: to avoid paradoxes (which we discussed in §3.2) higher-order order ghost state in Iris [28] introduces the "later modality" to reason on the execution time of programs. In practice, we did not find a need for higher-order ghost state, because the *bytes invariant* can be chosen ahead-of-execution, unlike *security labels* which are chosen depending on the protocol execution.

Computational model in proof assistants. Some tools embedded in proofs assistants provide computational guarantees, such as VCVio [36] in Lean [21], or SSProve [25] in Rocq [35]. These tools allow to specify and prove the security of cryptographic constructions in the computational model, whereas DyLean works in the symbolic model. On the one hand, because the computational model is more precise than the symbolic model, it means that security theorems proved with these tools are tighter than similar security theorems proved with DyLean; on the other hand, because proofs are easier in the symbolic model than in the computational model, it means that security theorems in DyLean can in principle tackle larger protocols, or more fine-grained compromise scenarios. We refer to [7] for a detailed survey on tools for cryptographic proofs that explains these trade-offs.

Dziembowski et al. [23] provide a framework for translating security proofs from the symbolic model into the computational model, by leveraging a *computational soundness* result (see [17] for a survey on computational soundness). This allows them to obtain the strong guarantees of the computational model, while only spending efforts required by the symbolic model. However, they only support a limited set of cryptographic primitives (and other abstract functions): symmetric encryption, tuples, and conditional swap. Instead, DyLean supports a much wider range of cryptographic primitives, which can additionally be extended by the user.

6. Conclusion

We showed how DyLean allows specifying cryptographic protocols (§2) and proving their security in the symbolic model (§3), and showcased the various capabilities of DyLean on several focused case studies (§4). DyLean allows for structured datatypes, equational theories, and threat models beyond existing methodologies.

Extensible semantics. Thanks to our framework for symbolic semantics, DyLean is versatile: as showcased by our case study on ACME draft 0 (§4.4), our symbolic semantics can be extended at will to account for the specific needs of the user. We believe this is an essential feature of DyLean to achieve our long-term objective, being that the research community can build on top of our symbolic semantics framework.

Investigating other proof methodologies. Our symbolic semantics (§2) allow DyLean users to specify cryptographic protocols and state their expected security properties. Then,

to *prove* that a protocol indeed satisfies its expected security properties, any means to achieve this proof is legitimate, as long as Lean accepts the proof. Our flagship proof methodology (§3) is centered around an input from the user (the trace invariant), which brings benefits inherited from DY* [39], such as allowing some form of protocol composition, a useful feature to analyze large protocols. However, the framework is open for exploring other and more automated proof methodologies, such as ones inspired from Tamarin's dependency graphs or ProVerif's Horn clauses.

Using generative AI for security proofs. The use of generative AI for proof construction has rapidly progressed in the last years, and one could envision applying it on security proofs.

On the side of *proofs*, as discussed in the last paragraph, any means to achieve this proof is legitimate, as long as Lean accepts the proof; and proofs constructed using generative AI are no exception. However, such a proof must be considered to be *adversarial*: in such a scenario, we advise using tools provided by the Lean ecosystem to check adversarial proofs, such as Comparator [3].

On the side of specifying *protocols* and *security properties*, however, we advise significantly more caution. Indeed, protocol specifications and security properties, be it handwritten by a human or generated using AI, must be critically reviewed to ensure they do not contain mistakes. To this end, it is useful to perform *sanity checks* (as we have described in §4), and it is useful to *execute* the specification against test vectors (if they exist). Although these checks increase the confidence that a specification makes sense, they do not, however, replace a thorough audit conducted by a human.

References

[1] Protocol proof ladder. `https://github.com/proof-ladders/protocol-ladder`, 2025.

[2] Dylean artifact including framework and case studies, 2026. `https://github.com/BobDyLean/bobdylean-paper-artifact`.

[3] Lean comparator. `https://github.com/leanprover/comparator`, 2026.

[4] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

[5] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. A generic methodology for the modular verification of security protocol implementations. In *Computer and Communications Security (CCS)*, CCS '23, pages 1377–1391, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Arthur Azevedo de Amorim, Amal Ahmed, and Marco Gaboardi. Cryptis: Cryptographic reasoning in separation logic. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy (S&P)*, pages 777–795, 2021.

[8] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[9] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. Automatic Certificate Management Environment (ACME). Internet-Draft draft-ietf-acme-acme-00, Internet Engineering Task Force, September 2015. Work in Progress.

[10] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV Standard: Break, Fix, Verify. In *SP*, pages 1766–1781. IEEE, 2021.

[11] Giampaolo Bella. *Inductive verification of cryptographic protocols*. PhD thesis, University of Cambridge, UK, 2000.

[12] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[13] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[14] Vincent Cheval and Caroline Fontaine. Automatic verification of finite variant property beyond convergent equational theories. In *2025 IEEE 38th Computer Security Foundations Symposium (CSF)*, pages 521–536, 2025.

[15] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. SAPIC+: protocol verifiers of the world, unite! In *USENIX Security Symposium*, pages 3935–3952. USENIX Association, 2022.

[16] Donald J Collins. A simple presentation of a group with unsolvable word problem. *Illinois Journal of Mathematics*, 30(2):230–234, 1986.

[17] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reason.*, 46(3–4):225–259, April 2011.

[18] Cas Cremers, Alexander Dax, and Aurora Naska. Breaking and Provably Restoring Authentication: A Formal Analysis of SPDM 1.2 including Cross-Protocol Attacks. In *CCS*, pages 1424–1438. ACM, 2025.

[19] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A Comprehensive Symbolic Analysis of TLS 1.3. In *CCS*, pages 1773–1788. ACM, 2017.

[20] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *IEEE Symposium on Security and Privacy*, pages 470–485. IEEE Computer Society, 2016.

[21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.

[22] Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 117–140, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[23] Stefan Dziembowski, Grzegorz Fabianski, Daniele Micciancio, and Rafal Stefanski. Computationally Sound Symbolic Cryptography in Lean . In *2026 2026 39th IEEE Computer Security Foundations Symposium (CSF) CSF*, pages 620–635, Los Alamitos, CA, USA, July 2026. IEEE Computer Society.

[24] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. Foundational multi-modal program verifiers. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[25] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. *ACM Trans. Program. Lang. Syst.*, 45(3), July 2023.

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[27] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2165–2180, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:20, 2018.

[29] Ben Laurie, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. RFC 9162, December 2021.

[30] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[31] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, 1999.

[32] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. `https://signal.org/docs/specifications/doubleratchet/`.

[33] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[34] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[35] The Coq Development Team. The coq proof assistant, September 2024.

[36] Devon Tuma, Quang Dao, James Waters, Alexander Hicks, and Nicholas Hopper. VCVio: Verified cryptography in lean via oracle effects and handlers. Cryptology ePrint Archive, Paper 2026/899, 2026.

[37] Théophile Wallez. *A Verification Framework for Secure Group Messaging*. Theses, École Normale Supérieure Paris, June 2025.

[38] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security. In *SP*, pages 4375–4390. IEEE, 2025.

[39] Théophile Wallez. DY* unchained: Now with composable security proofs and precise compromise scenarios. Cryptology ePrint Archive, Paper 2026/830, 2026.

[40] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *CADE*, Lecture Notes in Computer Science, pages 314–328. Springer, 1999.

[41] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021.