LeanDY: Type-Based and Trace-Based Symbolic Protocol Verification in Lean

Simon Jeanteur*, Lorenzo Veronese*, Magdalena Solitro†, Matteo Maffei*
*TU Wien, Vienna, Austria
†Fondazione Bruno Kessler, Trento, Italy

***Abstract*—Computer-aided formal verification is a widely used approach for the symbolic analysis of cryptographic protocols. However, many modern protocols rely on features that remain challenging for existing techniques. In particular, reasoning about state, time-dependent behavior, inductively defined data structures, unbounded executions, and conditional secrecy requires a level of expressiveness that is difficult to reconcile with effective automation. As a result, protocol verification has largely followed two disjoint paths: fully automated methods with limited expressiveness, or interactive proofs in general-purpose theorem provers that offer flexibility but only limited, non-specialized automation. Building on DY*, we bridge this gap by combining type-based reasoning with trace-based reasoning, enabling modular verification of stateful and unbounded protocols. Guided by the language and automation co-design (LAC) principle, our approach delivers protocol-specific automation while retaining high expressiveness. We implement this framework as the LeanDY library for the Lean proof assistant, extending the design of DY*, and combining protocol-specific automation with interactive proofs. Our framework supports, in a unified setting, a broad class of functional and security requirements, including secrecy and authentication for stateful protocols, as well as recursive conditional secrecy for protocols using XOR. We formalize SegWit-style blockchain primitives in LeanDY and demonstrate its expressiveness by developing an in-depth formalization of the Lightning Network on top of this blockchain model, verifying punishment mechanisms and properties that depend on chain liveness.**

I. Introduction

This is the technical report.

Over the past decades, computer-aided formal analysis of cryptographic protocols has matured into a widely used set of techniques and has become a key component of real-world protocol analysis. For example, ProVerif [1] and Tamarin [2] have supported the verification of the design of TLS 1.3 [3] and WireGuard [4], [5]; and F* [6], [7] has been used to verify the Signal protocol, the core of modern messaging applications [8]. However, many modern protocols rely on features that remain challenging for existing techniques. In particular, reasoning about global state, time-dependent behavior, unbounded executions, and *conditional secrecy*, i.e., secrets that must remain private only until a specific condition is met, requires a level of expressiveness that is difficult to reconcile with automation. These challenges are particularly prominent for blockchain-based protocols, which, despite their recent adoption in applications such as payment-channel networks, have seen fewer machine-aided verification efforts. The design of such protocols typically relies on a global, high-integrity ledger, requires reasoning across application-level and consensus-level specifications, and often combines safety and liveness arguments. Additionally, they may use constructions such as XOR, whose algebraic properties make symbolic reasoning notoriously difficult [9]. As a result, protocols such as the Lightning Network remain only partially verified, with the notable exceptions of [10], which analyzed the protocol up to fixed bounds (e.g., two in-flight payments), and [11], which formalized a simplified abstraction of the protocol (e.g., not supporting Hash Time Locked Contracts, HTLCs).

Symbolic methods for protocol analysis, as formalized by Dolev and Yao [12], can be categorized into two broad areas: trace-based and type-based methodologies. Trace-based symbolic proofs are based on a global append-only trace of all actions, e.g., sending messages or storing state, performed by protocol participants. Confidentiality and authentication properties are thus expressed in terms of *reachability* of specific trace configurations (e.g., there is no trace where a secret value is sent to the network). Type-based proofs, instead, employ types, notably *refinement types* [13], [14], to guarantee that well-typed protocol implementations respect some confidentiality, integrity, or authentication invariants (e.g., a program that sends over the network a value with type *high confidentiality* is rejected by the type system). Both approaches have limitations. Trace-based verification supports a high degree of automation, as symbolic provers such as ProVerif or Tamarin can explore all possible attacker behavior and automatically discover concrete attack traces or prove that such traces do not exist. However, these tools require examining complete protocol executions, which often requires user-provided helper lemmas for complex protocols composed of multiple modes or with an unbounded number of rounds. This degree of automation is well suited to certain classes of protocols, but becomes harder to retain when reasoning compositionally about cross-layer protocols or when mixing safety and liveness requirements. In contrast, type-based verification offers strong modularity: each function or sub-protocol is type-checked independently, and the type system guarantees that security invariants hold for any (unbounded) composition of well-typed modules. The main limitation of such advanced type systems is that proofs cannot be fully automated and require manual intervention.

In this paper, we build on the design of DY* [8], [15], which combines the above approaches into a unified methodology

where type-based proofs can be complemented with inductive trace-based reasoning to obtain security proofs expressed at the trace level. We implement this methodology as the LeanDY library for the Lean proof assistant, providing a mechanized framework for symbolic protocol verification with support for XOR and conditional secrecy (see Sec. VI for a detailed comparison). While XOR has been supported in Horn-clause–based approaches [9] and conditional secrecy has been studied in refinement-type systems [14], these features have never been combined in a single mechanized framework.

Leveraging Lean's powerful abstractions (e.g., quotients) and the *language and automation co-design* (LAC) [16] design principle, we improve on DY* by defining a structured embedded domain-specific language (eDSL) for specifying protocol implementations, which allows for automated type-based reasoning while providing controlled escape hatches (e.g., trace-based reasoning) when more expressivity is required. Specifically, LeanDY offers a smoother transition between automation and interactive proving compared to the previous F*-based approach, enabling users to fall back to the powerful Lean interactive mode when automation is insufficient for protocol-specific goals such as cross-layer reasoning, liveness, or punishment arguments in layer-2 blockchain protocols.

Contributions. More concretely, we contribute as follows.

(A) We present LeanDY, a library for protocol verification in the Lean proof assistant supporting dynamic compromise, mutable state, *conditional secrecy* properties, and XOR. The library supports a high degree of automation thanks to the type-based approach, while still allowing for interactive proofs in the trace-based model (Sec. III).

(B) We develop and integrate in LeanDY a type-based approach to *conditional secrecy* properties for protocols extending [15] to make use of XOR operations, including chained release patterns (Sec. II).

(C) We formalize SegWit-style blockchain primitives in LeanDY, providing a mechanized model of Bitcoin-like transactions, witness-based scripts, and timelocks (Sec. IV).

(D) We show the expressiveness of LeanDY through an in-depth formalization of the Lightning Network on top of this blockchain model, verifying punishment mechanisms and properties that depend on chain liveness. Our model is the first not to limit the state space and to support the core cryptographic and scripting building blocks of the Lightning Network (Sec. IV).

The LeanDY library, the formalization of blockchain primitives, the Lightning Network model, and additional example protocols are available as supplementary material.

II. Labels for Conditional Secrecy of XOR

In this section, we illustrate the security labels supported by LeanDY, exemplifying their use when type-checking protocols as well as the associated challenges. As a running example, we consider a blockchain protocol based on HTLCs that uses XOR, hashes, and signatures.

Conditional Release Protocol. Consider the protocol depicted in Fig. 1. We model a simplified abstraction of the opening and closing operations of payment-channel networks (PCNs) as described in [17]. This abstraction describes the conditional release chains present in protocols such as PCNs, atomic swaps using lock contracts, and the Tor network. Specifically, we assume that a setup phase has been successfully completed and that bob knows the secret rk_{bob} and the commitment $\mathcal{H}(rk_{alice} \oplus rk_{bob})$; that $carol$ knows the secret rk_{carol} and the commitment $\mathcal{H}(rk_{alice} \oplus rk_{bob} \oplus rk_{carol})$; and that $dave$ knows $rk_{alice} \oplus rk_{bob} \oplus rk_{carol}$; where $\mathcal{H}$ is a hash function. The protocol then proceeds in two phases: first, three channels are opened sequentially between $alice$ and bob, bob and $carol$, and $carol$ and $dave$. This is achieved by sending the signatures $\mathsf{sign}(\mathcal{H}(m))$ to the network, where m represents the revocation secret necessary to close the channel. For the purposes of this example, we abstract away the underlying blockchain mechanics and assume that broadcasting the revocation key to the network suffices to close the channel (see Sec. IV for our encoding of blockchain primitives). To allow for the sequential closing of all intermediate channels, this broadcasting of the revocation keys starts from $dave$, who sends $rk_{alice} \oplus rk_{bob} \oplus rk_{carol}$, allowing $carol$, who knows rk_{carol}, to close their channel and broadcast $rk_{alice} \oplus rk_{bob}$. Finally, since bob knows rk_{bob}, they can close their channel by broadcasting rk_{alice}. We can encode the requirement that all channels are closed (in order) when $alice$ receives rk_{alice} using the following formula.

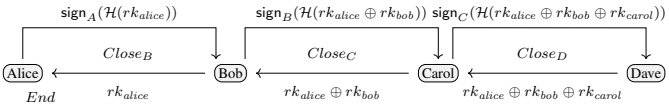


Fig. 1: Conditional release protocol with two intermediaries.

$$End \implies Close_B \wedge Close_C \wedge Close_D \tag{1}$$

where End, $Close_B$, $Close_C$, and $Close_D$ represent the protocol steps as depicted in Fig. 1. This corresponds to the *atomicity* property of PCNs, ensuring that either all payments are completed or none is.

Terms and Labels. We define symbolic terms M as follows:

$$\begin{array}{lll} N & \coloneqq x \mid x \oplus N & C \coloneqq alice \mid bob \mid \ldots \\ M & \coloneqq N \mid C \mid \mathcal{H}(M) \mid \mathsf{sign}_M(M) \mid \ldots \end{array}$$

where x represents an atomic nonce, and N is the XOR ($\oplus$) of multiple nonces; C denotes arbitrary constants (e.g., principal names); $\mathcal{H}$ represents hashes, sign signatures, etc.

We assign each term a secrecy label (or type) of the form:

$$L \coloneqq \mathcal{S}ec \mid \mathcal{P}ub \mid \mathcal{P}riv^n \mid [L_1/L_2]_E$$

where we denote public data as $\mathcal{P}ub$, secret data as $\mathcal{S}ec$; we use $\mathcal{P}riv^n$ to represent data that is only readable by the principal n and becomes public after compromise of n; and we use the notation $[L_1/L_2]_E$ for the conditional swap of labels L_1 and L_2 based on the condition E representing an event of the protocol. For instance, $[\mathcal{S}ec/\mathcal{P}ub]_{Leak}$ denotes a term that is initially secret and becomes public after the $Leak$ event. Events refer to specific steps of the execution of the protocol

and can be used to define its security specification in terms of *correspondence assertions* [18], i.e., ordering of protocol events. We assume that events are irreversible, in the sense that once an event is reached during execution, it cannot be undone; consequently, any conditional label swap triggered by such an event is permanent.

These confidentiality labels over-approximate the secrecy of symbolic bytes in the following sense: a term labeled $\mathcal{P}ub$ may be known to the adversary, whereas a $\mathcal{S}ec$ value is guaranteed to never be known by the attacker. In particular, we require every term that is sent to the network to have the $\mathcal{P}ub$ label, i.e., all data sent to the network is known to the attacker. This requirement allows us to infer the $\mathcal{P}ub$ label for all terms that appear on the network if the protocol and the adversary consistently follow the requirement (i.e., are well typed). Additionally, not every XOR-related term needs an explicit annotation: once labels are assigned to a small set of protocol terms, LeanDY can derive the labels of algebraically related XOR expressions automatically by taking the meet of the labels of their operands. Note that the labels defined above correspond to the specific instances required for the purposes of this section; however, LeanDY supports the definition of arbitrary dynamic labels. We refer to Sec. III-A3 for a more formal presentation of the typing rules, and Sec. III-B3 for the automated label derivation under XOR.

Proving Correspondence Assertions. More concretely, let us show how the above labels allow us to prove the correct ordering of protocol events in Fig. 1. We prove Eq. (1) using our conditional confidentiality labels. Assuming non-zero keys, let us define an initial labeling as follows.

$$\begin{array}{lll} T_1 & \coloneqq rk_{alice} \oplus rk_{bob} \oplus rk_{carol} & : [\mathcal{S}ec/\mathcal{P}ub]_{Close_D} \\ T_2 & \coloneqq rk_{alice} \oplus rk_{bob} & : [\mathcal{S}ec/\mathcal{P}ub]_{Close_C \wedge Close_D} \\ T_3 & \coloneqq rk_{bob} & : [\mathcal{S}ec/\mathcal{P}ub]_{Close_B} \end{array}$$

Here, T_1, T_2, and T_3 are identifiers that we can use to refer to the complex XOR expressions. The initial labels match the expected secrecy of the values that are sent on the network: since T_1 is the revocation key to close the channel, it is initially secret, but can be published to the network when the event $Close_D$ is reached. Similarly, T_2 is initially secret, but can be published only after both $Close_C$ and $Close_D$ have occurred ($Close_C \wedge Close_D$). Finally, in order to showcase our approach for automatically generating labels, we assign an initial label only to rk_{bob}, which is considered public after $Close_B$.

We can automatically derive labels for the remaining rk nonces by observing that the XOR of two conditionally confidential terms remains conditionally confidential, with a label given by the conjunction of the events in their respective conditions. Intuitively, the resulting label corresponds to the meet of the initial labels: it is public only if both operands are public, and private otherwise, since the XOR preserves the opacity of the individual nonces.

$$\begin{array}{lll} rk_{alice} & = T_2 \oplus T_3 & : [\mathcal{S}ec/\mathcal{P}ub]_{Close_B \wedge Close_C \wedge Close_D} \\ rk_{carol} & = T_1 \oplus T_2 & : [\mathcal{S}ec/\mathcal{P}ub]_{Close_C \wedge Close_D} \end{array}$$

Since rk_{alice} is $T_2 \oplus T_3$, the condition on the label is the conjunction of the three $Close$ events; similarly, the label of rk_{carol} is the conjunction of the labels of T_1 and T_2.

Once all values sent to the network have been assigned labels, we proceed as follows. For each message sent, we must establish that its label is public. For each value that is received, we compute the set of events that must have occurred for the value to be sent, under the assumption that its label is public. We analyze each message exchange (input/output pair) performed by each participant in isolation. We begin with $dave$. The value T_1 is sent over the network and must therefore be labeled $\mathcal{P}ub$, which is only possible after the occurrence of the $Close_D$ event. Continuing with $carol$, the value T_2 is sent and must also be public. While $carol$ can locally trigger the $Close_C$ event, establishing that $Close_D$ has occurred requires additional reasoning. In particular, $carol$ can compute T_2 only after receiving T_1 and combining it with their secret rk_{carol}. Since T_1 is received from the network, its label must be $\mathcal{P}ub$, which implies that the event $Close_D$ must already have taken place. A similar argument applies to bob. In order to send rk_{alice}, bob must execute the $Close_B$ event, and moreover, both $Close_C$ and $Close_D$ must have occurred in prior steps of the protocol. Finally, upon reaching the End event, $alice$ receives rk_{alice} from the network. Since this value must be public, it follows that all three $Close$ events have occurred, thereby establishing our correspondence assertion goal.

The above example highlights the compositional nature of our proof technique. In particular, we establish our security goal by analyzing each protocol participant in isolation, while maintaining the invariant that all data sent over the network is labeled as public. This enables us to derive, for each local action, the set of events that must have occurred to justify the release of a given value. By composing these local arguments, we can obtain a global ordering of events that satisfies the correspondence assertion, thus establishing our goal without requiring a global analysis of the protocol.

We refer to App. B for an extension of the above approach to handle dynamic compromise of protocol participants.

III. The LeanDY Library

The LeanDY library is a Lean framework for proving *trace properties*, such as authentication and confidentiality, of protocols modeled as state machines that rely on a specific set of APIs, covering cryptographic primitives (encryption, signatures, etc.), network communication, and mutable storage. Protocols use symbolic cryptography and are verified against a Dolev-Yao-style adversary. The basic proof technique is to establish a *trace invariant*: every state transition (i.e., any function that modifies the trace, including those modeling the adversary) is proved to preserve this invariant. Consequently, any reachable state, regardless of whether it is generated by an attacker or an honest protocol participant, satisfies the invariant.

Following the of language and automation co-design (LAC) design principle [16], LeanDY provides a general trace invariant, `valid_trace`, parameterized by protocol-specific behavior, and mechanisms for users to specify only these

protocol-dependent components. In addition, the library offers a Domain-Specific Language (DSL) for protocol specifications and includes a type system that supports the semi-automatic verification, for each function written in the DSL, of the protocol-specific parts of the trace invariant.

More specifically, users of the library write the protocol specifications as a collection of *message exchanges* using the DSL, instantiate the protocol-specific components of the trace invariant (states, events, and validity predicates), and then use the type system together with semi-automated proof support to show that each exchange satisfies `valid_exchange`. From these per-exchange proofs, LeanDY derives preservation of `valid_trace` for the trace updates generated by the protocol. As a final step, the user provides a separate extrinsic proof that the resulting trace invariant implies the desired security property (e.g., authentication or confidentiality).

We now present the main building blocks of the LeanDY library. We first describe the core library, which provides the foundational machinery for verification: the definition of traces, the trace invariant, the representation of symbolic bytes, the attacker model, and the type system used to reason about the confidentiality and integrity of messages. We then turn our attention to the user-facing layer of the library and present the DSL used to specify protocols and the proof automation included in the library.

A. Core Definitions

1) The Trace and the Trace Invariant: The trace represents the global trail of all observable actions performed by protocol participants and the adversary. We implement the trace as a list of `TraceEvents`, as reported in the following listing.

```
inductive TraceEvent (α η : Type) where
| MsgSent (msg : 𝔹 α η) : TraceEvent α η
| Corrupt (principal : ℙ) : TraceEvent α η
| SetState (content : 𝔹 α η) : TraceEvent α η
| Event (e : Event α η) : TraceEvent α η

abbrev 𝒯 (α η : Type) := List (TraceEvent α η) -- the trace
```

The trace records sent messages (`MsgSent`); the time point when a principal has been compromised (`Corrupt`, which can happen at any time); serialized protocol states (`SetState`); and protocol-specific events (`Event`) used for, e.g., authentication properties. Different traces are ordered by suffix, as new events are prepended to the trace. All trace properties in LeanDY, e.g., validity or secrecy of messages, are *monotone* with respect to this ordering. For instance, if a value is public (i.e., accessible to the adversary), it remains so after extending the trace.

We define a trace to be valid, i.e., `valid_trace`(tr) holds, if each event in the trace respects the following property:

```
abbrev valid_trace_event (Γ: LabelingEnv α η)
        (ev : TraceEvent α η) (tr: 𝒯 α η) : Prop :=
  match ev with
  | .MsgSent m => valid_bytes Γ m tr ∧ m.is_public Γ tr
  | .SetState s => match u_state_parse.parse s with
                   | some s => u_valid_state Γ s tr
                   | none => False
  | .Event e => u_valid_event Γ e tr
  | .Corrupt _ => True
```

Specifically, a message event is valid if the message is public with respect to its confidentiality label and its symbolic bytes are valid (Sec. III-A3). A state is valid if it can be parsed into a valid protocol-specific state, and an event must be valid according to the protocol-specific invariant (Sec. III-A4).

2) Symbolic Bytes and the Adversary: Every value sent over the network or stored in a principal's state is represented as symbolic bytes. Bytes are encoded as the following datatype:

```
inductive InnerBytes α η : Type
| Xor (content : Finset η)
| Const (c : Const)
| Key (p : α) (kind : Kind.Key) (k : InnerBytes α η)
| Sign (p : α) (m k : InnerBytes α η)
| Verify (p : α) (σ k : InnerBytes α η)
| Hash (p : α) (m : InnerBytes α η)
| SymEnc (p : α) (m k : InnerBytes α η)
| SymDec (p : α) (c k : InnerBytes α η)
| AEnc (p : α) (m k : InnerBytes α η)
| ADec (p : α) (c k : InnerBytes α η)
| Tuple (p : α) (a b : InnerBytes α η)
| Proj_1 (p : α) (x : InnerBytes α η)
| Proj_2 (p : α) (x : InnerBytes α η)

def Bytes (α η : Type) := Option (InnerBytes α η)
```

where we use the `Xor` constructor to represent the XOR-sum of all nonces in `content` (singletons effectively capture simple nonces); the `Const` constructor to represent constants, e.g., principal names; the `Key` constructor to represent keys containing their kind (e.g., signing, verification, or symmetric key) and their bitwise representation (usually a nonce); the rest of the `InnerBytes` constructors symbolically represent cryptographic operations (signatures, hashing, symmetric and asymmetric encryption), tuples, and their destructors. The α parameter distinguishes different versions of each constructor, allowing, e.g., the use of different hash functions in the same protocol. The available nonces, represented by η, are decided *ahead* of protocol execution. This departure from DY*—where nonces are dynamically generated as events on the trace—is inspired by the Bana-Comon logic [19] and gives us the fine-grained control over the labeling environment required for our XOR labeling (Sec. III-A3). Freshness can still be achieved by indexing nonces with, for instance, a fresh timestamp.

The `Bytes` type is the union of `InnerBytes` and a constant `fail`, represented by `(none : Bytes)`, to encode failure. For example, projecting a value other than a tuple or verifying an invalid signature returns `fail`. This reduction behavior is encoded in the normalization function γ. The function reduces a `Bytes` value to normal form, encoding the equational theory of symbolic cryptography: e.g., decrypting an encrypted message with the matching key reduces to the original plaintext.

Bytes Quotient and Equational Theories. To simplify the reasoning about equational theories, LeanDY exposes symbolic bytes using Lean's powerful quotient abstraction. Specifically, by considering *bytes modulo equality of their normal form*, we sidestep the problem of explicitly requiring the user to manually reduce constructor, destructor, and equivalence rules by calling γ. The main type for symbolic bytes exposed to users is the `BytesEq` (or $\mathbb{B}$) quotient type defined as follows.

$$[\![x]\!] : \mathbb{B} \coloneqq \{a : \texttt{Bytes} \mid \gamma(a) = \gamma(x)\}.$$

That is, we represent symbolic bytes as equivalence classes of bytes that reduce to the same normal form. This way, when encountering symbolic bytes in protocol code, users of LeanDY are not required to handle all possible ways specific bytes can be constructed. Consider, for instance, a protocol that decrypts a message, expects a pair, and compares the first projection with a previously stored nonce. Syntactically, `tuple(x,y).proj_1` is built from a projection constructor applied to a tuple constructor, but in $\mathbb{B}$ it is equal to `x` because both have the same γ-normal form, effectively making `proj_1` behave like a destructor. Protocol proofs can therefore use ordinary equality on $\mathbb{B}$ as the quotient hides the equational reasoning. Constructors lifted to $\mathbb{B}$ are written in lowercase; `xor_set` lifts `Xor`, the identifier `xor` being reserved for a smarter constructor.

Dolev-Yao Attacker. We model a Dolev-Yao-style adversary with the inductive predicate `AttackerKnows`:

```
inductive AttackerKnows Γ tr : 𝔹 α η → Prop
-- public knowledge
| Published {m} (hin : .MsgSent m ∈ tr) : AttackerKnows m
| Fail : AttackerKnows .fail
| Initial {n} (hin : n ∈ Γ.public_names tr) : AttackerKnows (.xor_set { n })
| Const {c} : AttackerKnows (.const c)
| Zero : AttackerKnows (.xor_set {})
-- constructors and destructors
| Xor : AttackerKnows (.xor_set s₁) → AttackerKnows (.xor_set s₂) →
      AttackerKnows (.xor_set (s₁ △ s₂))
| Key : AttackerKnows k → AttackerKnows (.key p kind k)
| Sign : AttackerKnows m → AttackerKnows k → AttackerKnows (.sign p m k)
| Verify : AttackerKnows σ → AttackerKnows k → AttackerKnows (.verify p σ k)
| Hash : AttackerKnows m → AttackerKnows (.hash p m)
| SymEnc : AttackerKnows m → AttackerKnows k → AttackerKnows (.symenc p m k)
| SymDec : AttackerKnows c → AttackerKnows k → AttackerKnows (.symdec p c k)
| AEnc : AttackerKnows m → AttackerKnows k → AttackerKnows (.asenc p m k)
| ADec : AttackerKnows c → AttackerKnows k → AttackerKnows (.adec p c k)
| Tuple : AttackerKnows a → AttackerKnows b → AttackerKnows (.tuple p a b)
| Proj_1 : AttackerKnows a → AttackerKnows (.proj_1 p a)
| Proj_2 : AttackerKnows a → AttackerKnows (.proj_2 p a)
```

The predicate models all information that can be derived by the attacker by applying public constructors starting from its initial knowledge. Specifically, the adversary initially knows all public information, including constants (`Const`), the `fail` value (`Fail`), and the nonce composed of all zeros (`Zero`; artifact of the finite set construction); all messages sent to the network (i.e., all messages in the trace, `Published`) and all nonces that are marked as initially public for the specific protocol (`Initial`). Then, the adversary can apply each constructor and destructor to the bytestrings it knows to derive new bytestrings. These are the traditional attacker rules (as used by, e.g., ProVerif [1]), which encode, for instance, that the adversary can decrypt encrypted messages if it knows the decryption key. The only exception is the rule for XOR: the symbol $\triangle$ denotes the symmetric difference of finite sets of nonce names ($s_1 \triangle s_2 \coloneqq (s_1 \cup s_2) \setminus (s_1 \cap s_2)$), effectively capturing the XOR of two "nonces" as per Prop. 1.

Property 1. $\left(\bigoplus_{n \in S_1} n\right) \oplus \left(\bigoplus_{n \in S_2} n\right) = \left(\bigoplus_{n \in S_1 \triangle S_2} n\right)$.

Remark 1. `AttackerKnows` *is defined over the quotient* $\mathbb{B}$*, capturing the equational theory for free.*

More generally, the encoding is designed to model a maximally powerful Dolev-Yao adversary. To this end, `AttackerKnows` itself is never constrained: any restrictions imposed on honest protocol executions must still permit all terms derivable by the adversary.

3) Type System: LeanDY includes a type system to support reasoning about the confidentiality and well-formedness of messages. Specifically, the trace invariant requires every message m added to the trace, i.e., sent on the network, to be public ($\Gamma \vdash_{tr} m : \top$) and valid ($\texttt{valid_bytes}^{\Gamma}_{tr}(m)$). We now present our construction of secrecy labels, the typing rules encoding confidentiality, and the well-formedness of bytes.

Secrecy Labels. Secrecy labels in LeanDY over-approximate the confidentiality level of bytestrings. Specifically, if a bytestring is labeled as public, it *may* be shared with the attacker, i.e., the public label is a precondition for sending it to a public channel (`MsgSent` in the trace). However, it can remain private. Conversely, private data is *guaranteed* to never be sent to a public channel or known to the attacker. We encode labels as general trace properties that express when (i.e., in which traces) the corresponding value may be compromised.

```
structure Label α η where -- later written ℓ
  corrupt₀ (tr: 𝒯 α η) : Prop
  monotone: Monotone corrupt₀
```

The `monotone` component of each label contains a proof that its trace property is growing with respect to the suffix ordering of traces. This notion corresponds to the intuition that corruption is permanent: since the attacker never forgets data once public, compromised data will remain compromised for all new traces.

Such an encoding, introduced in [20], [15], allows us to overcome the limited expressivity of the original DY* labels, which could only represent a small set of compromise conditions, and enables LeanDY to reason about conditional secrecy and temporal relationships between events. Moreover, new protocol-specific labels can be defined as part of each protocol's development without requiring modifications to the base library. For instance, LeanDY defines the following standard labels.

```
def 𝒫ub := {corrupt₀ _ := True, ...} -- public
def 𝒮ec := {corrupt₀ _ := False, ...} -- secret
def 𝒫 (p : ℙ) := {corrupt₀ tr := .Corrupt p ∈ tr, ...} -- priv. p
def meet l₁ l₂ := {corrupt₀ tr := l₁.corrupt₀ tr ∧ l₂.corrupt₀ tr, ...}
def join l₁ l₂ := {corrupt₀ tr := l₁.corrupt₀ tr ∨ l₂.corrupt₀ tr, ...}
```

where $\mathcal{P}$`ub` and $\mathcal{S}$`ec` denote, respectively, public and secret data; $\mathcal{P}$ `p` represents the data readable only by the principal `p`; meet and join combine labels using the $\wedge$ and $\vee$ of their `corrupt`$_0$.

Labels thus describe *who* can read a given set of messages and *when*. Following the above definitions of meet and join, we can define a *less public than* ordering of labels, parameterized by the current trace.

$$l_1 \leq_{tr} l_2 \coloneqq \forall tr', tr \leq tr' \to l_1.\texttt{corrupt}_0\ tr' \to l_2.\texttt{corrupt}_0\ tr'$$

Intuitively, l_1 is less public than l_2 starting from a specific point in time represented by the trace tr if, for all future traces, whenever l_1 is public, l_2 is public too. When tr is the empty trace, $l_1 \leq_{\emptyset} l_2$ represents the stronger property that l_1 is always less public than l_2. With this definition, we can encode the conditional swap label we used in Sec. II as follows, where $\texttt{h}_0$ ensures monotonicity:

```
def releasable (l₁ l₂: ℓ α η) (e: TraceEvent α η)
               (h₀: l₁ ≤∅ l₂) : ℓ α η where
      corrupt₀ tr := (if e ∈ tr then l₂ else l₁).corrupt₀ tr
      monotone := ...
```

This definition is extended in LeanDY beyond label pairs to accept any growing family of labels.

The relation $\leq_{tr}$ captures the information-flow ordering on labels, but for non-empty traces it is only a preorder, i.e., it lacks antisymmetry: two labels may differ syntactically while representing the same confidentiality level from trace tr onward. Therefore, we introduce the corresponding equivalence relation

$$l_1 \cong_{tr} l_2 \coloneqq l_1 \leq_{tr} l_2 \wedge l_2 \leq_{tr} l_1$$

which encodes which labels are indistinguishable starting from the trace tr. Quotienting labels by this equivalence yields the type $\mathfrak{l}_{tr}$ of labels at trace tr:

$$[\![l_1]\!]_{tr} : \mathfrak{l}_{tr} \coloneqq \{l_2 : \ell \mid l_1 \cong_{tr} l_2\}$$

Intuitively, $\mathfrak{l}_{tr}$ forgets historical distinctions that no longer affect confidentiality from trace tr onward. This quotient significantly simplifies formal reasoning with labels: properties can be stated directly on equivalence classes rather than introducing witnesses for equivalence (e.g., $\exists l, l \cong_{tr} \mathcal{P}ub$). More importantly, lifting $\leq_{tr}$ to the $\mathfrak{l}_{tr}$ equivalence classes yields a complete lattice.

Property 2 (Security Lattice). *Lifting $\leq_{tr}$ to $\mathfrak{l}_{tr}$ forms a* complete lattice. *We simply write this order $\leq$. The equivalence classes of* $\mathcal{P}$ub, $\mathcal{S}$ec, `meet` l_1 l_2 *and* `join` l_1 l_2 *are respectively the top ($\top$), bottom ($\bot$), meet ($[\![l_1]\!]_{tr} \sqcap [\![l_2]\!]_{tr}$) and join ($[\![l_1]\!]_{tr} \sqcup [\![l_2]\!]_{tr}$) of this lattice.*

Typing Rules for Confidentiality. Proofs of confidentiality that use the above secrecy labels are formulated in terms of the type system shown in Fig. 2. The judgment $\Gamma \vdash_{tr} b : l$ assigns a unique label $l \in \mathfrak{l}_{tr}$ to the term $b \in \mathbb{B}$ at trace tr depending on its shape. Specifically, constants, hashes, and the special value `fail` are public ($\top$). Signatures and secret keys preserve the label of their payload. The symmetric encryption of a value with a key is public only when the label of the key is *less public* than the label of the message. Similar considerations apply to asymmetric encryption. Finally, nonces, represented as the XOR of a finite set of nonces, are labeled according to a labeling environment Γ, which defines the expected confidentiality of the protocol. Γ maps sets of nonces to their initial labels. It is thus important for this initial assignment to be compatible with the expected behavior of XOR.

Example 1. *Let n_1, n_2, and n_3 be nonces. If $n_1 \oplus n_2$ and $n_2 \oplus n_3$ are published, then effectively $n_1 \oplus n_3$ is as well.*

We generalize Ex. 1 to finite sets of nonces using Prop. 1: if two sets of nonces are public, their XOR combination, i.e., their symmetric difference, must also be public. This yields (2):

$$\Gamma(s_1).\text{corrupt}_0\ tr \wedge \Gamma(s_2).\text{corrupt}_0\ tr \Rightarrow \Gamma(s_1 \triangle s_2).\text{corrupt}_0\ tr \quad (2)$$

Eq. (2) is then a proof obligation when defining a labeling environment. We present in Sec. III-B3 the automation that LeanDY provides for this definition and the automatic derivation of the initial labels we showed in Sec. II.

$$\textsc{L-Base}\ \frac{m \in \{\text{const}(c), \text{hash}(n), \text{fail}\}}{\Gamma \vdash_{tr} m : \top} \qquad \textsc{L-Xor}\ \frac{[\![\Gamma(t)]\!]_{tr} = l}{\Gamma \vdash_{tr} \bigoplus_{x \in t} x : l}$$

$$\textsc{L-Tuple}\ \frac{\Gamma \vdash_{tr} a : l_a \qquad \Gamma \vdash_{tr} b : l_b}{\Gamma \vdash_{tr} \text{tuple}(a, b) : l_a \sqcap l_b} \qquad \textsc{L-Sign}\ \frac{\Gamma \vdash_{tr} m : l}{\Gamma \vdash_{tr} \text{sign}(m, k) : l}$$

$$\textsc{L-PubKey}\ \frac{\text{kind} \in \{\text{Ver}, \text{AEnc}\} \quad \Gamma \vdash_{tr} k : l}{\Gamma \vdash_{tr} \text{key}(\text{kind}, k) : \top} \qquad \textsc{L-PrivKey}\ \frac{\text{kind} \in \{\text{Sign}, \text{ADec}, \text{SEnc}\} \quad \Gamma \vdash_{tr} k : l}{\Gamma \vdash_{tr} \text{key}(\text{kind}, k) : l}$$

$$\textsc{L-SEnc}\ \frac{\Gamma \vdash_{tr} m : l_m \quad \Gamma \vdash_{tr} k : l_k \quad l_k \leq l_m}{\Gamma \vdash_{tr} \text{symenc}(m, k) : \top} \qquad \textsc{L-AEnc}\ \frac{\Gamma \vdash_{tr} m : l_m \quad \Gamma \vdash_{tr} dk : l_{dk} \quad l_{dk} \leq l_m}{\Gamma \vdash_{tr} \text{aenc}(m, \text{key}(\text{aenc}, dk)) : \top}$$

Fig. 2: Typing rules for confidentiality labels.

Well-Formed Symbolic Bytes. Secrecy labels characterize the confidentiality of symbolic bytes; however, they do not, by themselves, restrict how honest protocol participants may use cryptographic primitives. Therefore, LeanDY provides a well-formedness predicate $\texttt{valid_bytes}^{\Gamma}_{tr}()$ for symbolic bytes and enforces that every message sent to the network satisfies this predicate. Users of the library can customize the predicate by instantiating the `CryptoPreds` type class.

```
class CryptoPreds α η where
  u_hash_pred     : LabelingEnv α η → α → 𝔹 α η → 𝒯 α η → Prop
  u_sign_pred     : LabelingEnv α η → α → 𝔹 α η → 𝒯 α η → Prop
  u_sym_enc_pred  : LabelingEnv α η → α → 𝔹 α η → 𝒯 α η → Prop
  u_asym_enc_pred : LabelingEnv α η → α → 𝔹 α η → 𝒯 α η → Prop
  ... -- proofs of monotonicity
```

Each field specifies protocol-specific restrictions on honest cryptographic uses; for example, a protocol may allow signing only messages of a particular shape or only after a specific event has occurred. The class defines default (unrestricted) implementations for all predicates, so for each protocol users can specify only the relevant fields, inheriting the default implementation for the others.

In order to allow for the unconstrained use of cryptography by the attacker, LeanDY requires each cryptographic operation to either satisfy the corresponding `CryptoPreds` predicate or be applied with a public key. For example, symmetric encryption is always permitted when the encryption key is public, otherwise it must satisfy the predicate `u_sym_enc_pred`. Analogous conditions apply to hashing, signing, and asymmetric encryption.

```
def hash_pred Γ p m tr := is_public Γ m tr ∨ u_hash_pred Γ p m tr
abbrev attacker_controlled Γ kind k m tr :=
  is_public Γ (key p kind k) tr ∧ is_public Γ m tr
def sign_pred Γ p m k tr :=
  attacker_controlled Γ .Sign k m tr ∨ u_sign_pred Γ p m k tr
def sym_enc_pred Γ p m k tr :=
  attacker_controlled Γ .SymEnc k m tr ∨ u_sym_enc_pred Γ p m k tr
def asym_enc_pred Γ p m k tr :=
  attacker_controlled Γ .AEncK k m tr ∨ u_asym_enc_pred Γ p m k tr
```

The $\texttt{valid_bytes}^{\Gamma}_{tr}()$ predicate is defined in terms of the above disjunctions and is reported in Fig. 3. Constants are always valid, tuples are well-formed when their components are well-formed, and cryptographic operations must satisfy the corresponding predicates: either the relevant keys are available to the attacker, or the protocol-specific usage predicates hold.

$$\text{V-BASE}\quad \frac{m \in \{\text{fail},\ \text{xor}(x),\ \text{const}(x)\}}{\texttt{valid_bytes}^{\Gamma}_{tr}(m)} \qquad \text{V-KEY}\quad \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(k)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{key}(\text{KIND}, k))}$$

$$\text{V-TUPLE}\quad \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(a) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(b)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{tuple}(a,b))} \qquad \text{V-HASH}\quad \frac{\texttt{hash_pred}^{\Gamma}_{tr}(m) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(m)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{hash}(m))}$$

$$\text{V-SIGN}\quad \frac{\texttt{sign_pred}^{\Gamma}_{tr}(m,k) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(m) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(k)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{sign}(m, \text{key}(\text{sign}, k)))}$$

$$\text{V-SENC}\quad \frac{\texttt{sym_enc_pred}^{\Gamma}_{tr}(m,k) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(m) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(k)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{symenc}(m, \text{key}(\text{symenc}, k)))}$$

$$\text{V-AENC}\quad \frac{\texttt{asym_enc_pred}^{\Gamma}_{tr}(m,k) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(m) \quad \texttt{valid_bytes}^{\Gamma}_{tr}(k)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{aenc}(m, \text{key}(\text{aenc}, k)))}$$

Fig. 3: Definition of `valid_bytes()`.

Soundness. We define the soundness of the LeanDY type system in terms of the confidentiality and well-formedness of the data that can be derived by the adversary.

Property 3 (Type-system soundness). *For every valid trace, anything known by the attacker is public and well-formed:*

$$\forall b.\, \texttt{AttackerKnows}^{\Gamma}_{tr}(b) \rightarrow \Gamma \vdash_{tr} b : \top \wedge \texttt{valid_bytes}^{\Gamma}_{tr}(b).$$

With this property (and its contraposition), we ensure that the adversary cannot recover private data starting only from publicly available information and that all adversary operations preserve the well-formedness of public data. Moreover, it shows that we can derive such confidentiality and well-formedness judgments without restricting the capabilities of the attacker.

4) Protocol-Specific States and Predicates: The trace invariant `valid_trace` refers to protocol-specific components that must be specified as part of each protocol definition. This is achieved using the `UserPreds` type class, which extends `CryptoPreds` with user events, user states, a validity predicate for states, defined as an instance of the `HasValid` type class, and a parser/serializer for states (App. C).

```
class UserPreds (α η : Type) extends CryptoPreds α η where
  u_valid_event : LabelingEnv α η → Event α η → 𝒯 α η → Prop
  u_valid_event_monotone (Γ e) : Monotone (u_valid_event Γ e)
  u_state : Type
  u_state_has_valid : HasValid α η u_state
  u_state_parse : Parsing.Parsable (𝔹 α η) u_state
```

As for `CryptoPreds`, all fields have default implementations (predicates return `True`, and monotonicity obligations are then proved automatically), so users only specify the fields relevant to their protocol. The `u_state_parse` field connects the symbolic trace, where states are stored as bytes, with the structured state type (`u_state`) used by protocol code. States are shared and global.

B. Protocols DSL and Proof Automation

1) Protocols as Exchanges: A cryptographic protocol is modeled as a set of atomic actions named *exchanges*. Each exchange corresponds to an (honest) protocol participant receiving a message from the network, performing a stateful computation on the received bytestring, and sending a new message, optionally updating its state or triggering protocol-specific events. More specifically, each exchange is a function of type `exchange`:

```
structure Atom α η [...] where
  (message: 𝔹 α η) -- message sent out
  (state: u_state) -- new state saved
  -- events triggered before and after output and state
  (events post_events): List (Event α η)

abbrev exchange α η [...] :=
  (i: ℕ) → (m : 𝔹 α η) → (s : up.u_state) → Atom α η
```

The input includes a timestamp, a message received from the network, represented as symbolic bytes, and a protocol-specific state; the output includes a message to be published, a possibly updated state, and two lists of protocol-specific events. The timestamp is unique to each execution, thereby enabling nonce freshness by picking a different inhabitant of η upon each replay of the same `exchange`.

Example 2. *Consider the simple authentication protocol [21]:*

$$A \rightarrow B : \langle A, B, Na \rangle \qquad B \rightarrow A : \{\langle Na, A \rangle\}^{s}_{Sk_{AB}}$$

We model the protocol as three exchanges: (i) A sends the tuple $\langle A, B, Na \rangle$ and stores the three values in its state; (ii) B receives the tuple, computes the symmetric encryption of $\langle Na, A \rangle$ using Sk_{AB} and sends it over the network; (iii) A receives the encrypted bytestring, decrypts it, and verifies that the received Na and A correspond to the values stored in the previous exchange. This last exchange is encoded as:

```
def initiator_check : exchange Unit Nonce :=
  λ _ input state => Id.run do
    let .Init sa sb sna := state | return default
    let dec := input.symdec () (sk sa sb)
    if dec ≠ .fail then
      let na' := dec.proj_1 (); let a' := dec.proj_2 ()
      if sa = a' ∧ sna = na' then
        return { default with events := [.done sa sb na'] }
    return default
```

The type `exchange Unit Nonce` represents an exchange that uses the concrete `Nonce` datatype for nonces (see Sec. III-A2). The exchange is a function that receives a timestamp (unused here), an `input`, and a `state`, and returns a structure with the four fields described above. We use a `default` structure with an empty list of events, an empty state, and the symbol `fail` as the default output in case of error (e.g., when decryption or equality checks fail). When the decryption is not `fail` and all checks are successful, the exchange returns the event `done` representing the principal A confirming the end of the protocol.

Each exchange must be proven correct with an extrinsic proof: the user must provide the associated correctness theorem proving the following proposition.

$$\begin{aligned}
&\texttt{valid_exchange}(op) \coloneqq \forall(\mathbb{t} : \mathbb{N})\,(in : \mathbb{B})\,(st : State),\\
&\quad \Big(\texttt{valid_bytes}^{\Gamma}_{tr}(in) \wedge\ \texttt{valid_state}^{\Gamma}_{tr}(st) \wedge\ (\Gamma \vdash_{tr} in : \top)\Big) \rightarrow\\
&\qquad (\forall e \in evs,\ \texttt{valid_event}^{\Gamma}_{tr}(e))\\
&\qquad \wedge\ \texttt{valid_bytes}^{\Gamma}_{tr++evs}(msg) \wedge\ (\Gamma \vdash_{tr++evs} msg : \top)\\
&\qquad \wedge\ \texttt{valid_state}^{\Gamma}_{tr++evs}(st')\\
&\qquad \wedge\ (\forall e \in postevs,\ \texttt{valid_event}^{\Gamma}_{tr++evs++[msg,st']}(e))\\
&\text{where } op(\mathbb{t}, in, st) = \langle msg, st', evs, postevs \rangle
\end{aligned} \tag{3}$$

Specifically, `valid_exchange` requires that protocol exchanges preserve the validity of states and messages: given a valid public input and a valid state for a trace tr in a labeling environment Γ, an exchange must ensure that (i) the new message msg is both valid and public for the trace $tr +\!\!+ evs$, (ii) the new state st' is valid for $tr +\!\!+ evs$, and (iii) each event in evs (resp. $postevs$) is valid with respect to the original trace tr (resp. the complete $tr +\!\!+ evs +\!\!+ [msg, st']$). LeanDY provides a set of exchange-specific tactics to automate such proofs, allowing users to focus on the protocol-specific parts of the proof while trivial goals are solved automatically. We present our proof automation in App. D and complete Lean snippets in App. G.

Once `valid_exchange` has been proved for all exchanges of a protocol, LeanDY lifts these per-exchange proofs to the whole protocol: every trace generated by interleaving its exchanges with arbitrary attacker actions satisfies `valid_trace` (Listing 1). The security goals of the protocol then follow from `valid_trace` by a separate extrinsic proof.

2) Execution Model and Soundness: The above result is achieved by composing the observable effects of exchanges on the global trace. We now describe the LeanDY operational semantics and prove that `valid_trace` is preserved by any valid execution of exchanges.

Executing Exchanges. The effect of an exchange on the trace is given by the `transition` function:

```
def transition (op : exchange α η) : Traceful Unit := do
  let input ← recv
  let state ← getState
  let time  ← getTime
  let ⟨msg, newState, evs, postevs⟩ := op time input state
  triggerEvents evs
  saveState newState
  send msg
  triggerEvents postevs
```

For each exchange `op`, the transition function fetches a state and a message from the trace and executes `op` on them. The four outputs are added to the trace by first triggering all events, storing the state, sending the new message, and finally triggering the `postevs`. The `Traceful` monad implements the functions for trace manipulation, e.g., `Traceful.recv` and `Traceful.send` for receiving and sending messages, and is implemented as:

```
structure ChoiceFn where
  msg (tr: 𝒯 α η) : Option { m : 𝔹 α η // .MsgSent m ∈ tr }
  state (tr: 𝒯 α η) : Option { s : 𝔹 α η // .SetState s ∈ tr }

abbrev Traceful := ReaderT ChoiceFn (StateT (𝒯 α η) Option)
```

Computations in `Traceful` may index into the trace using specific criteria. In particular, the `ChoiceFn` record provides functions to select a message and a state from the trace. We use this abstraction to reason about an over-approximation of all possible choices for messages or states: by keeping `ChoiceFn` universally quantified, we do not make any assumption on how states or messages passed to exchanges are selected. This is critical: any proof about `transition` holds regardless of which messages or states are chosen, which captures an adversary delivering out-of-order or malformed messages.

The `transition` function plays a central role in establishing the first soundness result of LeanDY, namely that each valid exchange preserves the validity of the trace:

```
theorem transition_valid {ch: ChoiceFn}
  (op: exchange α η) (tr: 𝒯 α η)
  (hop : valid_exchange Γ tr op) (h₀: valid_trace Γ tr)
  (ht : transition op ch tr = some ((),tr')) :
  valid_trace Γ tr'
```

Attacker and Scheduler. The attacker may, at any time, (i) send any message it can derive, i.e., any m such that $\texttt{AttackerKnows}^{\Gamma}_{tr}(m)$ (Sec. III-A2), or (ii) corrupt any principal. It cannot read or modify stored states directly; it learns the secrets of a corrupted principal through the labels of these secrets, which become public upon corruption and thus available to the attacker through `Initial`. A scheduler then decides the next action, either an attacker action or an honest exchange, based on the current trace, and `run_many` repeats this for a given number of steps:

```
abbrev Scheduler α η := 𝒯 α η → Action α η
def Action.execute : Action α η → Traceful α η Unit
  | Att (Send m) => send m | Att (Corrupt p) => corrupt p
  | Honest op => transition op
def Action.valid Γ tr : Action α η → Prop
  | Att (Send m) => AttackerKnows Γ tr m
  | Att (Corrupt _) => True
  | Honest op => valid_exchange Γ tr op

def run_many (next: Scheduler α η) (limit: ℕ) :
  Traceful α η Unit := repeatM limit $ do
    (next (← get)).execute
```

Both the scheduler `next` and the choice function `ch` are arbitrary functions of the whole trace: together, they model an adversary that controls the network and the interleaving of unboundedly many protocol sessions, at exchange granularity. Then, as long as there exists a state that is valid on the empty trace, a valid scheduler always produces valid traces, as stated in the main soundness theorem of LeanDY.

```
theorem soundness [..] [Inhabited up.u_state] Γ
  (ch: ChoiceFn) (next: Scheduler α η) (limit: ℕ) :
  -- the scheduler only performs valid actions
  (h: ∀ tr, valid_trace Γ tr → (next tr).valid Γ tr) →
  -- the default state is valid
  (hv: up.u_state_has_valid.valid Γ default []) →
  -- ch returns a message and a state whenever there are some
  (hresp: ch.responsive) →
  let tr₀ := [.MsgSent default, .SetState (serialize default)]
  ∃ tr, run_many next limit ch tr₀ = some ((), tr)
      ∧ valid_trace Γ tr
```

Listing 1: Soundness of LeanDY

The hypothesis `h` does not constrain the adversary: attacker actions are valid by definition, and `valid_exchange` is proved once for all traces, so `h` holds for any scheduler that picks among the exchanges of the protocol and the attacker actions. The initial trace $\texttt{tr}_0$ extends the empty trace (which is trivially valid) with a public constant and a default state, so that the first exchange has something to receive; the only obligation is that this default state is valid (`hv`). Finally, the responsiveness of `ch` (`hresp`) ensures that no transition fails, hence that the run produces a trace and the theorem is not vacuous. The general form of the theorem, accounting for optional restrictions on the attacker, is given in App. G-A.

3) Automatic Labeling Environments with XOR: Manually building a labeling environment that follows Eq. (2) can be difficult. A naive but intuitive method is to observe that $x_1 \oplus x_2$ can only be known by those who know both x_1 and x_2. This translates into Sec. III-A3's `meet` ($\sqcap$) as $\Gamma(x_1 \oplus x_2) = \Gamma(x_1) \sqcap \Gamma(x_2)$. Then, starting from an assignment Γ_0 of labels to atomic values (i.e., terms of the form `.Xor` $\{n\}$), we can naturally and easily build a consistent labeling environment:

```
def from_singles (Γ₀ : η → ℓ α η) : LabelingEnv α η where
  Λ s := ⨅_{s₀∈s} Γ₀ s₀
```

We discuss other solutions to Eq. (2) and extend our approach beyond atomic values in App. E.

IV. Verifying the Lightning Network

In this section, we present our formalization of the Lightning Network [22] using the LeanDY library. We first describe our encoding of blockchain primitives, which allows us to prove safety properties that depend on chain liveness. We then present the protocol itself, the encoding of its components in LeanDY (blockchain, liveness), and the corresponding safety theorems.

Remark 2. *All Lean* `structure`*s of type* `Type` *in this section derive* `ValidParsable` *(see App. C).* $\mathbb{B}^+$ *denotes the non-failing bytes type (i.e.,* $\{b : \mathbb{B} | b \neq \texttt{fail}\}$*), and* `Vec` τ n *denotes a list of elements of type* τ *of length* n*. We use ellipses ("…") to omit irrelevant content or inline proofs.*

A. Modeling Blockchain Primitives

We model a SegWit-style Unspent Transaction Output (UTXO) blockchain inside LeanDY. The model is intentionally close to Bitcoin on the core ledger mechanisms (transaction identifiers, spendable outputs, witness-based scripts, and timelocks) while abstracting away the full Bitcoin virtual machine and consensus rules. This is expressive enough to model real-world blockchain protocols, yet lightweight enough to support mechanized proofs. The key aspect of the model is that it is built on top of the primitives introduced above via the parsing framework (App. C). This brings, for free, all the soundness guarantees, composition properties, and proof automation of LeanDY to the realm of blockchain protocols.

1) Structures: We first introduce the core datatypes.

Pointers. Bitcoin spends outputs by referring to them through a transaction identifier and an output index. A `Pointer` packages a transaction identifier `TXID`—the hash of selected fields of the transaction—together with an output index `VOUT`.

```
structure TXID α η := (hash : 𝔹⁺ α η)
structure VOUT := (n : ℕ)
structure Pointer α η := (txid: TXID α η) (vout: VOUT)
```

Transactions. A transaction carries inputs, witnesses, outputs, and a locktime. We rule out malformed transactions with mismatched numbers of inputs and witnesses using vectors.

```
structure Transaction α η where
  (ninputs noutputs : ℕ)
  inputs    : Vec (Input α η) ninputs
  witnesses : Vec (𝔹⁺ α η) ninputs
  outputs   : Vec (Output α η) noutputs
  locktime  : ℕ
```

Each output contains an amount and a locking script `scriptPubKey`, while each input contains a pointer to a previous output, an unlocking script `scriptSig`, and a sequence number used for relative timelocks.

```
structure Output α η where
  amount : ₿
  scriptPubKey : 𝔹⁺ α η
```

```
structure Input α η where
  outptr : Pointer α η
  scriptSig : 𝔹⁺ α η
  sequence : ℕ
```

As in SegWit, transaction identifiers are computed excluding the `witnesses`. This allows for computing a transaction's `TXID` before its witnesses are attached, which is essential to break spending dependencies, e.g., in channel openings.

Blocks and Blockchain. A block is a list of transactions, and a blockchain is a list of blocks. Rather than treating the chain as a separate global state, we reconstruct it from the protocol trace: whenever a block is published, it is sent as a message event. The current chain is thus extracted by filtering the trace for block messages.

```
structure Block α η := (transactions : List (Transaction α η))
structure Blockchain α η := (blocks : List (Block α η))

def List.get_blockchain (tr : 𝒯 α η) : Blockchain α η :=
  tr.reverse.filterMap
      (λ | .MsgSent msg => do_parse msg as Block α η | _ => none)
```

This design is crucial for the rest of the development: blockchain facts are treated as ordinary trace facts. Consequently, the general trace machinery of LeanDY applies unchanged: blockchain-related messages live in the same trace as protocol states and events, thereby benefiting from the same monotonicity and validity infrastructure introduced in Sec. III. Moreover, this design enforces part of blockchain consensus: effectively, only one canonical blockchain matters. Restrictions on the attacker (App. G) additionally ensure that every blockchain snapshot embedded in a message is consistent with this chain, and that the chain remains valid.

2) Validity: Validity is defined hierarchically. Scripts justify individual spends, pointers determine whether a referenced output exists and is still unspent, transactions combine script checks with value preservation, and a blockchain is valid when each of its transactions is valid against its preceding prefix.

Scripts. A key feature of Bitcoin-like blockchains is the use of scripts to guard outputs. Fully implementing the Bitcoin virtual machine in LeanDY would significantly increase both proof size and proof engineering overhead. Instead, we abstract over script semantics with a `Script` type class.

```
structure ScriptInput α η where
  (arg witness stripped : 𝔹⁺ α η)
  (sequence locktime relativeHeight : ℕ)

class Script α η δ where
  eval (self : δ) (input : ScriptInput α η) : Prop
  decidableEval self input : Decidable (eval self input)
```

The `eval` predicate relates the spent output script (parsed as a value of type δ) to the data available during script execution: the `scriptSig`, the witness, the stripped transaction used as the signature preimage, and the fields required for timelock checks; `decidableEval` ensures executability.

Example 3. *The Pay-to-PubKey script (P2PK) is modeled as:*

```
structure P2PK α η := (pk : 𝔹⁺ α η)
structure P2PK.Redeem α η := (sign : 𝔹⁺ α η)

open P2PK in instance : Script α η P2PK where
  eval self input := Option.getD (dflt := False) $ do
    let {sign} ← do_parse input.witness as Redeem α η
    return verify () sign self.pk = input.stripped
  decidableEval := ...
```

Pointers. A pointer is valid if it refers to an existing output of an existing transaction in the current blockchain. It is *spendable* if, moreover, its output has not already been consumed by another on-chain transaction.

```
def spendable_ptrs : Finset (Pointer α η) :=
  (output_ptrs b).toFinset.filter (λ ptr => ptr.txid.unique b)
    \ (input_ptrs b).toFinset
```

The additional uniqueness check on `TXID`s prevents ambiguity when retrieving an output from its transaction identifier. This check is slightly stronger than Bitcoin's operational treatment of collisions; our model simply rejects ambiguous cases instead of selecting a specific transaction. For the setting considered here, this restriction does not impose significant limitations. Indeed, Bitcoin's `BIP 30` [23] rules out ambiguity at the level of unspent outputs: duplicate transaction identifiers are disallowed whenever the earlier transaction still has unspent outputs. Correspondingly, we prove that all spendable outputs of a valid chain have unique transaction identifiers.

Transactions. Transaction validity is parameterized by a blockchain (i.e., the chain against which the transaction is checked), a script language, and the height at which the transaction is to be inserted. A valid transaction must spend only spendable pointers, satisfy the locking scripts of all its inputs, and preserve value.

```
structure Transaction.valid (vheight := b.height) : Prop where
  unique : (tx.ninputs = 0) → tx.unique b
  spendable : tx.all_input_spendable b
  balance : (tx.ninputs ≠ 0) → tx.valid_balance b
  valid_scripts : tx.valid_eval_scripts b script_t
  locktime_ok : tx.locktime ≤ validationHeight
```

Transactions without inputs (i.e., coinbase transactions) spend no pointer from which `TXID` uniqueness could be derived; for those, the `unique` field requires a proof of uniqueness, reflecting `BIP 30`. For ordinary transactions, `spendable` prevents double spending, `balance` enforces that outputs do not create money, and `valid_scripts` ensures that every input actually unlocks the output it claims to spend. Finally, `locktime_ok` enforces that a certain height has been reached.

Blockchain. Finally, a blockchain is valid when each transaction it contains is valid with respect to its preceding prefix:

```
def Blockchain.valid := ∀ tx ∈ b, tx.valid_restr b script_t
```

where `tx.valid_restr b` checks `tx` against the sub-blockchain of `b` consisting of the transactions that precede `tx`, matching the causal reading of blockchain execution: only earlier transactions provide spendable outputs.

B. Lightning Network

Payment channels are the core building blocks of payment-channel networks such as Lightning [22]. Two parties first lock funds in a joint on-chain output (the *common fund*) and then perform an unbounded number of off-chain balance updates by exchanging signed replacement transactions. Only the opening and closing phases require interaction with the blockchain: all intermediate payments remain off chain. The main challenge is preventing a party from broadcasting an old commitment transaction that reflects a more favorable balance. This is addressed with revocation secrets and punishment: each update replaces the previous commitment state and simultaneously reveals the secret needed to punish the owner of that old state if it is ever posted on chain. The Lightning Network further extends this concept by enabling payments between any two participants of the network, even if they do not share a channel, by propagating the funds along a path of channels using HTLCs. These feature further revocation secrets, ensuring that no participant is left unable to propagate the payment (Fig. 4a), and timeout mechanisms to protect against disappearing nodes.

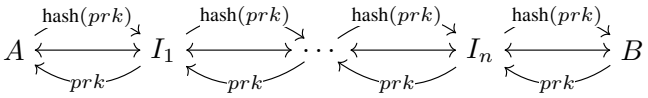
We formalize an unbounded Lightning Network, abstracting away its routing. We model unbounded channel openings, sequences of off-chain updates (balance updates, adding, fulfilling, and timing out in-flight payments), unilateral publications of a commitment transaction, and the corresponding punishment transactions. We further show the local propagation of knowledge required for locally atomic in-flight payments. Using LeanDY's composition, we effectively verify the safety of all intermediaries on a payment path. We abstract away from features orthogonal to our proof obligations, such as fees, balance negotiation, and cooperative closing. This section focuses on the protocol (Sec. IV-B1 and IV-B2), the formalization of liveness (Sec. IV-B4), and our verification results (Sec. IV-B5). Further details are available in App. F.

1) Protocol Overview: Abstracting away routing effectively simplifies the protocol into an extended payment channel, summarized in Fig. 4. We model the opening phase, the update phases, punishment, and force-closing. Conceptually, the protocol is built on top of smaller message exchange groups.

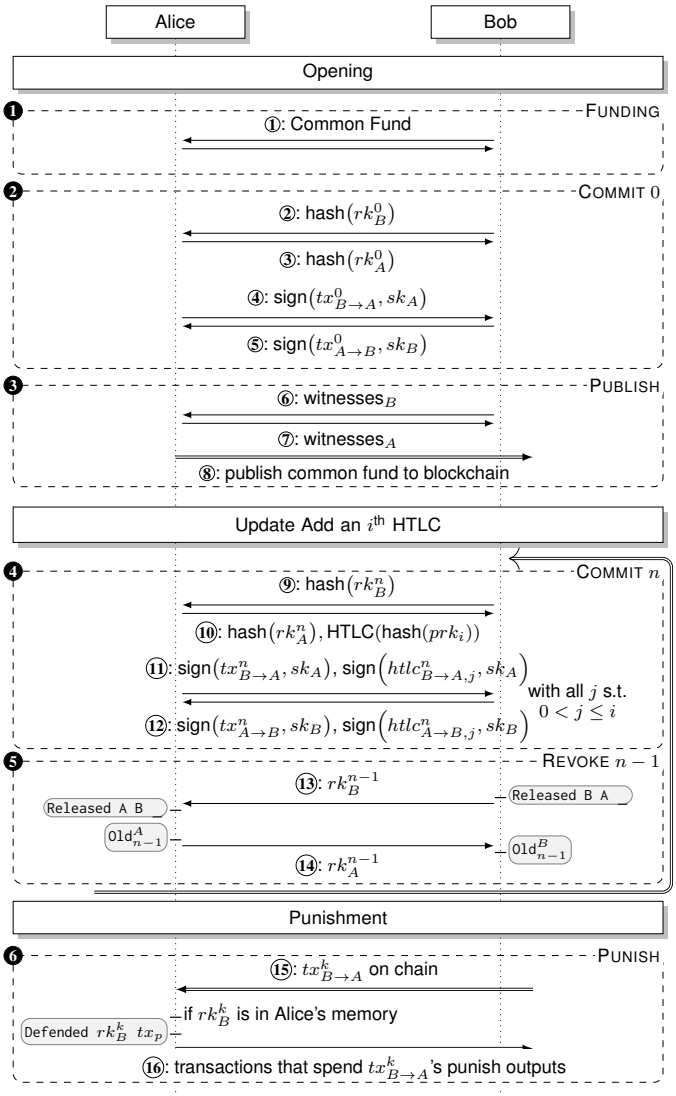
Funding and Initial Publication. The opening phase consists of a commitment and two additional groups (Groups ❶ and ❸) to set up the initial common fund used in all subsequent payment-channel transactions. Group ❶ captures the negotiation to agree on the inputs of the common fund (represented as an unconstrained parameter in our model). The parties pre-compute the `TXID` of the not-yet-published funding transaction and perform a *commitment*. They wait until Group ❸ to exchange the witnesses required to make the funding transaction valid and publish it, thereby avoiding locking money forever in case Group ❷ fails.

Commitments. When both parties agree on a new channel state (e.g., new balance, adding/fulfilling/timing out an HTLC), they exchange information letting each party build a unilateral closing transaction. In Groups ❷ and ❹, this consists of the hash of the new revocation key and the signatures of the (stripped) commitment transaction (Tx. 1 from Fig. 4c) and of all second-stage HTLC transactions (Tx. 2 and 3).

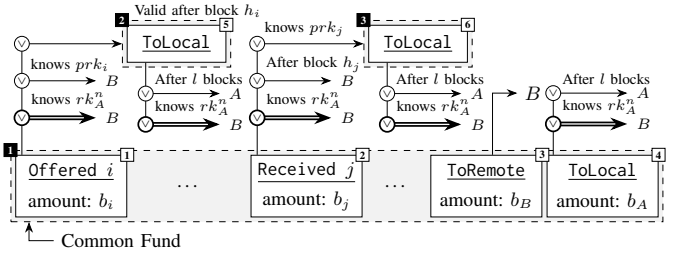
Revocation. Both parties can revoke an old channel state by exchanging their respective revocation keys (Group ❺). These revocation keys act as proof of good faith that neither party will



(a) Payment hash and preimage (prk) propagation along a path.



(b) Lightning payment channel.



where `to_self_delay` is set to l, and the timeout of HTLC k is set to height h_k for the `Offered` and `Received` outputs; similarly prk_k is the corresponding pre-image of the payment hash; ⟹ denotes the punishment path; and **1** refers to a transaction, while [1] an output.

(c) Diagram of the flow of coins of $tx_{A\to B}^n$.

Fig. 4: High-level model of the protocol.

attempt to publish the now-outdated state on chain. Failure to uphold this promise results in punishment. Event (Old$_n^A$) flags a state as old; it is used to trigger liveness. When revocation and commitment are executed sequentially, some of the messages can be merged (e.g., Messages ⑫ and ⑬).

Punishment. Group ❻ models punishment (for Alice in Fig. 4b). If a transaction spending the common fund appears on chain and Alice knows its revocation key, she sends transactions redeeming Bob's funds (the ⟹ paths in Fig. 4c).

Updates. We model adding, fulfilling, and timing out HTLCs, as well as updating the balance of the channel, one at a time. Fig. 4b shows HTLC addition. When adding an HTLC (i.e., left-to-right propagation in Fig. 4a), Message ⑩ contains the payment hash of the new HTLC, thus allocating coins to the in-flight payment. *Fulfilling* corresponds to the right-to-left sweep of Fig. 4a, unlocking the HTLC by sharing the preimage of the payment hash in Message ⑩. Finally, HTLCs can simply time out, in which case the agents may trigger an update to remove them from their commitment transactions.

2) Transactions and Scripts: The security of Lightning channels relies on their specific choice of transactions and scripts. The protocol involves three kinds of transactions: the *funding transaction*, which has unspecified inputs but is locked by a `CommonFund` script; the *closing transactions* (e.g., $tx_{A\to B}^n$ in Fig. 4), which spend the aforementioned funding transaction and are locked into the shape of Fig. 4c; and the *punishment transactions*, which spend cheating closing transactions (along Fig. 4c's ⟹ path). Scripts are wrapped in SegWit-style Pay-to-Witness-Script-Hash (P2WSH) [24] scripts (formalized in App. F-A1). This results in five kinds of scripts: `CommonFund`, `ToRemote`, `ToLocal`, `Offered`, and `Received`. This section introduces each of them; further details on their formalization are in App. F.

`CommonFund`. The common fund's script is a standard 2-of-2 multisig script [25]: it can only be redeemed with signatures from both parties, which enforces the shape of the commitment transactions. We capture these semantics using the following instance of `Script`:

```
structure CommonFund := (pka pkb : 𝔹⁺ Unit Nonce)
structure CommonFund.Redeem := (signa signb : 𝔹⁺ Unit Nonce)

open CommonFund in
instance : Script Unit Nonce CommonFund where
  eval self input := Option.getD (dflt:=False) $ do
    let {signa, signb} ← do_parse input.witness as Redeem
    return  verify () signa self.pka = input.stripped ∧
            verify () signb self.pkb = input.stripped
  decidableEval := ...
```

where the `Option` monad short-circuits to `False` on failure.

`ToRemote`. (Output [3]) The *remote* party of a closing transaction is the party that does not store this transaction in memory (e.g., Bob for $tx_{A\to B}^n$). They can retrieve their funds unconditionally and immediately from the transaction by providing a signature.

`ToLocal`. (Output [4]) Much of the power of a Lightning channel comes from the `ToLocal` output. This output is typically spent by the party initiating a unilateral closing to retrieve their share. It is parameterized by a hash (hash(rk_A^l) if the closing party is A). It can be redeemed by either party under some conditions:

```
1  -- Messages ⑫ and ⑬
2  def accept_add_htlc_and_revoke : exchange Unit Nonce :=
3  λ _ input state => Atom.fromOption $ do
4    -- (1) parse the input and retrieve the current state
5    let { blockchain := b, msg := {update, add}, .. } ←
6      do_parse input as WithBlockchain AddHTLCUpdate
7    let .Std s@{self, ps, common_fund, pc_old, pc_new, ..} := state  | none
8    let some { balance, htlcs, self_rk := old_rk,
9          other_rk := none, i := i_rk, /- rest to 'some _' -/ } := pc_old | none
10   let { self_rk := nselfRk, /- rest to 'none' -/ } := pc_new | none
11   -- (2) check the input
12   let nhrk := hash nselfRk -- pre-shared at Message ⑨
13   let htlcs := htlcs.withRevocationHash nhrk
14   let received := add.toReceivedOutput self ps nhrk
15   let newBalance := balance.set self.other (balance[self.other] - add.amount)
16   let newHtlcs := { htlcs with received := htlcs.received ++ [received] }
17   guard (add.amount ≤ balance[self.other])
18   guard (checkBaseUpdateSignature self ps common_fund newBalance nselfRk
         update.striped_tx_sign newHtlcs b)
19   guard (newHtlcs.RemoteSignaturesValid self ps common_fund newBalance nhrk
         update.htlc_signatures)
20   -- (3) conclude: sign, save the new state, and answer
21   let reply ← mkSignedBaseUpdate? self ps common_fund newBalance nselfRk
         update.hrk newHtlcs b
22   let nstate : PcState := { pc_new with
23     balance := newBalance, htlcs := newHtlcs, other_hrk := some update.hrk,
24     valid_sign_stripped := some update.striped_tx_sign,
25     valid_htlc_signatures := some update.htlc_signatures }
26   return {
27     events := [ (mkReleaseEvs self ps i_rk), -- releases old_rk
28                 ... ] ++ (peerCommitmentSigningEvents ...)
29     state := .Std {s with pc_old := pc_new, pc_new := nstate}
30     message := serialize (MkNewStateAndRevoke.mk reply {old_rk}) }
```

Listing 2: Exchange for one part of a channel update

(i) the closing party (A) once `to_self_delay` blocks have passed; (ii) the counterparty (B) immediately if it can provide the revocation key rk_A^l. Point (ii) is the punishment. B only knows rk_A^l if A gave it away. In other words, Message ⑭ and COMMIT $(l+1)$ (Group ❹) took place and a newer state was agreed upon. This ensures no gain can be made by putting an old state on chain.

`Offered` and `Received`. (Outputs 1 and 2) The HTLC outputs (one per HTLC) further include locking by the payment hash (hash(prk)) and timeout at an absolute height h. They can be spent in three different ways: (i) immediately when knowing prk; (ii) once the blockchain has reached height h; (iii) to the counterparty (e.g., B) with the revocation key rk_A^l. We use an `Offered` script when the HTLC transfers coins from the closing party (A) to B upon fulfillment. Its timeout path is locked to a `ToLocal` transaction by a 2-of-2 multisig, ensuring punishment remains possible, with the height constraint enforced by the transaction's `locktime`. Conversely, a `Received` HTLC transfers coins to the closing party upon fulfillment through a `ToLocal` transaction, while its timeout path pays B.

3) Formalizing the Protocol: An Example Exchange: We illustrate our encoding with the exchange of Listing 2 (Messages ⑫ and ⑬), which features most patterns of the model. Agents can play either role; here, the agent is Bob.

Generalities. Generic roles rely on `AOrB`, the sum of the constants `A` and `B`, and on `×× T`, effectively the product `T × T`. The role `self` indexes a `×× T` (e.g., `balance[self.other]` for `balance : ×× ₿`), which encodes the normalization of events and keys (e.g., in scripts and multiparty signatures). Control flow uses Lean's `Option` monad and `do`-notation; `guard` fails when its condition does not hold.

States. The agent state (`StdState`) records the channel context—the role (`self : AOrB`), the participants (`ps : ×× ℙ`), and the common fund output—and at most two *channel states* (`PcState`, App. F-B1), each carrying the balance, the in-flight HTLCs, the revocation material (keys, hashes, counter `i`), and the peer's signatures. Here, `pc_old`, the active state and source of truth for most channel parameters, must be *complete* (all fields but Alice's revocation key filled), while `pc_new`, under negotiation, must only hold Bob's revocation key from Message ⑨ (Lines 8 to 10). Note the asymmetry: Bob provides Alice with two valid states while holding only one himself.

Messages. Every exchange takes a `WithBlockchain T` input, pairing a payload `T` with a blockchain snapshot `b` on which exchanges can safely reason, as the attacker can only embed snapshots consistent with the valid chain (App. G). Here, `T` combines an `AddHTLC add` (amount, payment hash, and expiry of the new HTLC) and Alice's half of Group ❹, `update`: her fresh revocation hash `hrk` (Message ⑩) and her signatures of Bob's new commitment (`striped_tx_sign`) and second-stage HTLC transactions (`htlc_signatures`, Message ⑪).

The Exchange. (1) *Retrieve the state* (Lines 5 to 10): parse the input, match the agent state, and check the shape of the channel states. (2) *Check the input* (Lines 12 to 19): as Alice only sends signatures, Bob *recomputes* what she should have signed, deriving the new balance and HTLC set (re-keyed under his fresh revocation hash) from `add` and his state. He then checks her signatures of the resulting commitment (Line 18) and of every second-stage HTLC transaction (Line 19); as a commitment may carry arbitrarily many HTLCs, these checks are factored into utility functions over whole HTLC sets. (3) *Conclude* (Lines 21 to 30): Bob signs Alice's new commitment and second-stage transactions, saves the new channel state (Line 22), shifts the channel states, and emits the signing events and the release event of Sec. II's conditional swap (`mkReleaseEvs`), allowing him to publish `old_rk`. He answers with `MkNewStateAndRevoke`, merging Messages ⑫ and ⑬.

4) Formalizing Liveness: The security of Lightning channels relies on liveness. Indeed, the punishment mechanism only functions if the participants are sufficiently reactive to execute the punishment. Unfortunately, such properties are typically out of reach of standard Dolev-Yao models. These assume an adversary with full control over the network, including the ability to permanently block a party from interacting with the protocol, preventing them from defending themselves.

We therefore turn towards restricting the attacker. Although solutions exist for traditional Dolev-Yao-based verifiers [26], these push tools like PROVERIF and TAMARIN to their limits. In contrast, LeanDY can leverage the full power of Lean and Mathlib [27] to reason about arbitrary list (and therefore trace) properties, letting us restrict the proofs to the relevant traces, notably those that uphold liveness. Lightning's channels are safe under the liveness of two entities: the defending party, which sends the punishment transactions, and the miners, which

eventually include them on chain.

Participant Liveness. We first define a mechanism to *select* an `exchange` for execution. The core component is a predicate `P` that, given a trace, selects a specific `exchange` along with its associated message and state. We express that `op` is selected by `P` with message `msg` and state `s` in the trace `tr` at indices t_{msg} and t_s using the following predicate:

```
def Liveness.Select [up : UserPreds α η]
  (P : 𝒯 α η → exchange α η → 𝕥 → 𝕥 → Prop) (op: exchange α η)
  (tr) (t_msg t_s: 𝕥) (msg:𝔹 α η) (s: up.u_state) : Prop :=
    P tr op t_msg t_s -- P selected op with indices t_msg, t_s
  ∧ tr.IsMessageAt t_msg msg -- msg is in the trace at t_msg
  ∧ tr.IsStateAt t_s s -- s is in the trace at t_s
```

Liveness then guarantees that whatever exchange is selected by `Liveness.Select` (if any) is eventually executed, provided the trace grows by a sufficient number of blocks (`n`).

```
def Liveness [up : UserPreds α η]
  (P : 𝒯 α η → exchange α η → 𝕥 → 𝕥 → Prop)
  (n : ℕ) (tr tr'): Prop :=
  ∀ (msg:𝔹 α η) (s: up.u_state) op,
    (Liveness.Select P op tr _ _ msg s) → -- op is selected
    (tr ≤ tr') → (tr.get_blockchain.height + n
                  ≤ tr'.get_blockchain.height) → -- the traces are
      related and sufficiently many blocks have been mined
    ∃ l₁ l₂, tr' = l₂ ++ (op msg s).as_list ++ l₁ ++ tr -- then op
      is executed somewhere between tr and tr'
```

Miner Liveness. Blockchain consensus algorithms rely on the presence of a sufficient number of honest miners, which eventually add all valid transactions to the blockchain [28]. Assuming miner liveness means that if a transaction is broadcast to the miners (captured by the `published_tx_at` predicate) and can remain valid for at least `n` blocks, it will be included in the blockchain within those `n` blocks.

```
def miner_liveness (n : ℕ) tr (t:𝕥) tx : Prop :=
  (h₁: published_tx_at tr tx t) →
  (hv: tx.ValidLongEnough script_t n tr t) →
    tx ∈ tr.get_blockchain
```

where `ValidLongEnough script_t n tr t tx` captures that `tx` can remain valid for `n` blocks after being sent to the miners.

5) Results: We provide three main results (snippets in App. F-C).

Theorem 1 (Closing). *Barring closing, each party can construct a valid closing transaction after Groups ❷ and ❹.*

Theorem 2 (Local Atomicity). *We can publicly retrieve the preimage of the payment hash of any HTLC output spent before its timeout or revocation.*

Theorem 3 (Safety of Punishment). *Informally, assuming the liveness of both the participants (Group ❻) and the miners, if a revoked commitment transaction is published on the blockchain, then all its coins will eventually belong to the defender.*

Scope and Limitations. Our model is a *generalized payment channel*: routing, fees, and chain reorganizations are abstracted away, and updates apply one at a time (e.g., two HTLCs cannot be added at once). Hence, Th. 2 proves knowledge of the payment preimage—what an intermediary needs to claim an in-flight payment after a remote force-close—rather than actual fund recovery. Our results further assume a single valid chain on which all parties agree, participant and miner liveness, and an uncorrupted defender, either explicitly as premises or through trace restrictions (App. G). Scripts are hand-written `Script` instances rather than a Bitcoin Script interpreter.

Other Verifications of Lightning. Fabiański et al. [11] prove in WHY3 [29] that an honest party recovers its expected balance on chain within bounded time against a Byzantine counterparty, but for a single channel without HTLCs. Grundmann and Hartenstein [10] verify the full BOLT specification, multi-hop payments included, against an ideal payment system in TLA+ [30], but by model checking finite instances under a restricted adversary, with pen-and-paper refinement proofs. Our results are more local and not composed end-to-end, but cover an unbounded model with HTLCs against a Dolev-Yao attacker with liveness hypotheses, and Th. 3 recovers the cheater's *full* commitment value, not just the defender's balance.

Proof Effort. The Lightning development totals 18k lines of code (excluding comments, blanks, and demo harness): 4.6k of specification and 13.5k of proofs, atop the 12.3k-line LeanDY library (blockchain layer included). Counting the unique lines of each theorem's dependency cone, Ths. 1 and 2 are cheap—0.3k and 0.6k lines respectively—on top of 4.1k of validity proofs. On the other hand Th. 3 takes 7.8k lines, two thirds of which implement the liveness reachability machinery (selection, scheduling, response-horizon bookkeeping) undermined by the complex commitment transaction (Fig. 4c) requiring a reachability proof for each punishment path. Unaccounted lines are spent on miscellaneous properties and sanity-checks.

V. Discussion

A. The LeanDY Verification Pipeline

A LeanDY development is organized in four layers, each discharging the proof obligations of the next. First, the library fixes the symbolic bytes $\mathbb{B}$, the attacker `AttackerKnows`, and the label lattice $\mathfrak{l}_{tr}$, and proves once and for all that the attacker only knows public and well-formed bytes (Prop. 3). Second, each protocol instantiates the generic invariant `valid_trace` with a labeling environment Γ and the `CryptoPreds` and `UserPreds` describing its protocol-specific states, events, and cryptographic usages. Third, every exchange is verified in isolation against `valid_exchange` using the type system with the help of our semi-automated tactics. The soundness theorem (Sec. III-B2) lifts these local proofs to every trace produced by any scheduler interleaving unboundedly many sessions with attacker actions. Finally, security goals are proved extrinsically from `valid_trace`. The conditional release protocol of Sec. II follows this pipeline: Γ is derived automatically from the labels of T_1, T_2, and T_3, each exchange is checked locally, and Eq. (1)'s atomicity follows from `valid_trace`. The Lightning Network exercises every layer at scale. Its Γ assigns `releasable` labels to revocation keys and payment preimages, which remain private to their owner until the corresponding release event; the blockchain model is a reusable library, built on the parsing primitives, that stores the chain in the trace and encodes agreement and validity of the chain as restrictions; channel, HTLC, and punishment operations are written as exchanges and verified individually; and the closing, local atomicity, and punishment theorems are trace-based proofs

from `valid_trace`, the last one taking participant and miner liveness as explicit hypotheses.

B. Assumptions and Limitations

Trust and Model. Beyond the Lean kernel, LeanDY trusts its term algebra and normalization γ, the attacker `AttackerKnows` (one rule per constructor and destructor), and its operational semantics; per protocol, the model, the security statements, restrictions, and liveness hypotheses are trusted too. Labels and the trace invariant are proof devices, except that Γ's labels on individual nonces fix the attacker's initial knowledge and what each corruption reveals. As in DY*, results hold in the symbolic model, without computational soundness, and only for trace properties (no equivalence-based goals). The term algebra is fixed: LeanDY currently lacks Diffie-Hellman and key derivation, and XOR is restricted to nonces; adding a primitive requires re-proving Prop. 3. The parameter α and the parsing infrastructure (App. C) mitigate this restriction.

States, Corruption, and Nonces. Unlike in DY*, the library does not force stored secrets to be readable by their owner: corruption leaks exactly the nonces whose labels become public, and labeling them accordingly is up to the user (App. B). States are recorded rather than overwritten, and an exchange may resume from any stored state. This over-approximation is sound, but it rules out properties that rely on an honest agent never reusing an old state, which is why Th. 3 is stated from the defender's side. Similarly, nonce freshness is a modeling discipline (indexing by timestamps) rather than a library guarantee.

Executable Code. Executable implementations are not a goal of this work. Exchanges are ordinary Lean functions, so models already run symbolically: our Lightning demo executes channel opening and HTLC updates, and we prove that recorded runs are runs of the semantics, and that the opening produces a valid trace. Concrete reference implementations appear feasible, but would require an abstract interface for bytes (as in DY* Unchained [15]), concrete wire formats [31], and, for Lightning, real Bitcoin Script semantics.

VI. Related Work

Mechanized Symbolic Protocol Analysis. Over the years, several works have focused on the mechanized security of cryptographic protocols in the Dolev-Yao model [12]. In his seminal work, Lawrence C. Paulson [32] developed a rigorous technique for verifying cryptographic protocols within the Isabelle/HOL proof assistant [33], representing protocols as execution traces, and applied it to the NSL protocol and, in a later work, Kerberos [34]. Despite its expressiveness, automation remains limited: proofs are largely manual, as general-purpose tools (e.g., Sledgehammer [35]) struggle with protocol-specific inductive reasoning. Busi et al. [21] follow a similar methodology in their `StrandsRocq` library, which encodes the inductive proof technique of strand spaces [36] in RocQ and provides strand-specific tactics that automatically solve trivial goals. However, its representation stays close to the original strand space model, which limits the degree of achievable automation compared to approaches following the LAC principle [16], where the language can be deliberately restricted to enable more effective automation.

The DY* library [8] instead models protocols as effectful F* procedures that manipulate a global trace and verifies them with SMT-based automation, at the cost of `assert` annotations that mix specification and proof. Wallez's overhaul of DY* [15], [20] moves to extrinsic proofs, a workflow close to interactive theorem proving, which further highlights the limited support of F* for tactic-based proofs and motivates our choice of Lean. Arquint et al. [37] instead target protocol *implementations*: inspired by the global trace and labels of DY*, they verify existing Go and C code in concurrent separation logic. In concurrent work, Wallez is developing a Lean library based on this extension [38], published on ePrint 14 days after our arXiv version [39]; it focuses on extensible equational theories, whereas ours focuses on automation (e.g., the automatic labeling of XOR terms required by HTLC-based protocols, Sec. II).

Comparison with DY* and Arquint et al.. LeanDY builds on the trace-invariant methodology of DY* [8], from which it inherits the combination of type-based and trace-based reasoning, mutable state, and dynamic compromise. We compare LeanDY with DY*, its overhaul by Wallez [15] (DY*U), and the work of Arquint et al. [37] along three axes: (i) the expressiveness of labels, in particular for XOR; (ii) the execution model and composition; and (iii) the trusted base.

First, regarding labels, while DY*U already encodes labels as trace predicates, which we adopt in LeanDY (and in which our `releasable` label can be expressed), none of these frameworks supports XOR. Adding the XOR *equations* is relatively simple, as XOR-sums of nonces are represented as finite sets and the quotient $\mathbb{B}$ hides their normalization (Sec. III-A2). However, *typing* XOR terms requires labeling environments that are closed under symmetric difference (Eq. (2)), since the attacker can combine any two XOR-sums it knows. LeanDY derives such environments automatically from a basis (Sec. III-B3); this relies on nonces being predetermined, following the Bana-Comon approach [19], so that sets of nonces drawn by different principals at different times can be labeled ahead of execution. In DY*, by contrast, the label of a nonce is fixed when the nonce is generated. Additionally, the LeanDY approach of quotienting labels by their equivalence at a given trace ($\mathfrak{l}_{tr}$) turns the label preorder into a complete lattice, which removes existential witnesses from proofs (Sec. III-A3).

Second, regarding the execution model, while DY* defines reachable traces through arbitrary F* attacker programs, LeanDY fixes an explicit semantics in which soundness quantifies over arbitrary schedulers and choice functions (Sec. III-B2). Moreover, the semantics of LeanDY is parametric in environment assumptions: restrictions (App. G) uniformly weaken the attacker for trace-based results, while type-based results hold against the unrestricted attacker. Protocol steps are atomic in both frameworks, but DY* relies on a manually enforced coding discipline (one action per function), whose violation silently weakens the attacker [37]. In LeanDY, instead, this

discipline is enforced by the type of exchanges: since an exchange is a pure function that receives one message and one state and returns one message, its atomic execution does not hide intermediate knowledge from the attacker. Steps that interleave several receptions and emissions must therefore be split into several exchanges, similarly to TAMARIN rules [2]. Arquint et al. avoid the coding discipline altogether, as their proofs hold for all interleavings, and use linear resources to prove injective agreement, for which LeanDY has no dedicated support; however, they keep the original DY* labels and do not support XOR. Regarding composition, verification in LeanDY is modular with respect to exchanges, and soundness holds for any set of exchanges that are valid for the same invariant; however, the protocol-specific invariant is monolithic, as in the original DY*. In particular, LeanDY does not yet offer either the layered predicates of [40] or the composable sub-invariants of DY*U. In the Lightning case study, reuse instead comes from the blockchain library, which is generic in the protocol invariant and contributes its consensus assumptions as restrictions. Since the sub-invariants of DY*U only rely on mechanisms also available in Lean [15], porting them to LeanDY is a natural direction for future work.

Third, regarding the trusted base, all proofs in LeanDY, including those found by our automation, are checked by the Lean kernel and depend only on Lean's standard axioms (in particular, not on `native_decide`, which keeps even the compiler out of the trusted base). In contrast, DY* and DY*U discharge proof obligations with the Z3 SMT solver through F*, and Arquint et al. rely on SMT-backed program verifiers.

Automated Symbolic Protocol Analysis. In parallel with the above efforts on the mechanized analysis of protocols, fully automated solvers have been developed over the years. Notably, Blanchet introduced PROVERIF [1], one of the earliest and most widely used tools supporting so-called push-button verification of protocols expressed in the applied π-calculus [41]. A comparable approach is taken by Meier et al. with TAMARIN [2]. Both tools support the analysis of protocols with an unbounded number of sessions and messages. However, they require reasoning about the full protocol execution, lacking mechanisms to analyze sub-protocols in isolation and to compose verification results. In subsequent work, PROVERIF has been extended to support observational equivalence [42], a limited form of induction [43], and XOR [44]. Additional techniques have been proposed to integrate XOR into these verifiers, such as reducing Horn theories with XOR to the XOR-free case [9], or extending the TAMARIN prover to support stateful protocols with XOR [45]. While these approaches move automated verifiers toward greater flexibility, they fall short when automation alone is insufficient.

Blockchain Protocol Verification. The emergence of blockchain technologies introduced new classes of protocols and security properties to verify. Significant effort has been devoted to formalizing the foundations of blockchain systems, covering both consensus [46] and smart contract execution [47]. At the contract level, Andrychowicz et al. [48] model contracts using UPPAAL [49], and Jacquot et al. develop CHAUSSETTE [50], which uses symbolic execution to discover vulnerabilities in Bitcoin scripts. Our work, instead, focuses on modeling layer-2 (off-chain) protocols [22], specifically the Lightning Network and its channels. Lightning has attracted substantial research interest across several aspects. This includes game-theoretic analyses such as CHECKMATE [51] and the work of Rain et al. [52]. Meanwhile, in the symbolic model, Boyd et al. [53] use TAMARIN to analyze HTLCs, and Barthe et al. [54] extend TAMARIN with support for timed cryptographic protocols. Additionally, Hüttel et al. [55], [56] verified secrecy and authenticity of specific Lightning sub-protocols using PROVERIF. Closest to our scope are the recent verifications of the complete Lightning Network. These include a bounded TLA+ model by Grundmann and Hartenstein [10] and a simplified WHY3 formalization without HTLCs [11] (see Sec. IV-B5 for a comparison). While these are impressive achievements, they focus on the monolithic verification of a single protocol. In contrast, we focus on the usability and generality of LeanDY. Our formalization of an unbounded Lightning Network demonstrates that LeanDY is expressive enough for this class of protocols, providing a general framework to verify the safety properties of complex protocols.

VII. CONCLUSION

In this paper, we presented LeanDY, a library for symbolic protocol verification in Lean that combines type-based reasoning with inductive trace-based proofs. The framework builds on the design of DY* and supports mutable state, dynamic compromise, and conditional secrecy properties for protocols using XOR. Following the LAC design principle, LeanDY provides protocol-specific automation while still allowing users to complement it with interactive proofs when more expressivity is required. We developed a mechanized model of SegWit-style blockchain primitives in LeanDY, capturing Bitcoin-like transactions, scripts, and timelocks. We showed the expressiveness of the framework by developing an in-depth formalization of the Lightning Network, verifying punishment mechanisms and properties that depend on chain liveness.

ACKNOWLEDGMENTS

Use of Artificial Intelligence Tools. Large language models were used during the preparation of this work for two distinct purposes. First, LLMs have been used for editorial purposes, including proofreading and refining the presentation of the paper. Second, we experimented with various LLMs for proof automation and to assist in specification and proof writing for the Lightning Network development. We present the results of our use of LLMs in App. A. The final design decisions, implementations, proofs, and specifications presented in this work were produced and verified by the authors. Specifically, the LeanDY library was implemented by hand, and LLM assistance has only been used as a form of autocomplete for specific proofs.

References

[1] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *IEEE CSFW*, 2001, pp. 82–96.

[2] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *CAV*, 2013, pp. 696–701.

[3] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin, "Proving the TLS Handshake Secure (As It Is)," in *CRYPTO*, 2014, pp. 235–255.

[4] B. Lipp, B. Blanchet, and K. Bhargavan, "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol," in *IEEE EuroS&P*, 2019, pp. 231–246.

[5] N. Kobeissi, G. Nicolas, and K. Bhargavan, "Noise explorer: Fully automated modeling and verification for arbitrary noise protocols," in *IEEE EuroS&P*, 2019, pp. 356–370.

[6] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *ACM ICFP*, 2011, pp. 266–278.

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

[8] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY*: A modular symbolic verification framework for executable cryptographic protocol code," in *IEEE EuroS&P*, 2021, pp. 523–542.

[9] R. Küsters and T. Truderung, "Reducing protocol analysis with XOR to the XOR-free case in the horn theory based approach," in *ACM CCS*, 2008, pp. 129–138.

[10] M. Grundmann and H. Hartenstein, "Security of the lightning network: Model checking a stepwise refinement with TLA+," in *IFM*, 2026, pp. 313–335.

[11] G. Fabiański, R. Stefański, and O. S. T. Litos, "A Formally Verified Lightning Network," in *FC*, 2026, pp. 3–20.

[12] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[13] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM TOPLAS*, vol. 33, no. 2, 2011.

[14] M. Backes, C. Hriţcu, and M. Maffei, "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations," *J. Comput. Secur.*, vol. 22, no. 2, pp. 301–353, 2014.

[15] T. Wallez, "DY* Unchained: Now with Composable Security Proofs and Precise Compromise Scenarios," in *IEEE S&P*, 2026.

[16] D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner, "Automating formal proofs for reactive systems," in *ACM PLDI*, 2014, pp. 452–462.

[17] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *ACM CCS*, 2017, pp. 455–471.

[18] T. Woo and S. Lam, "A semantic model for authentication protocols," in *IEEE S&P*, 1993, pp. 178–194.

[19] G. Bana and H. Comon-Lundh, "Towards Unconditional Soundness: Computationally Complete Symbolic Attacker," in *POST*, 2012, pp. 189–208.

[20] T. Wallez, "A verification framework for secure group messaging," Ph.D. dissertation, ENS Paris, 2025.

[21] M. Busi, R. Focardi, and F. L. Luccio, "Strands rocq: Why is a security protocol correct, mechanically?" in *IEEE CSF*, 2025, pp. 33–48.

[22] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," White paper, 2016.

[23] P. Wuille, "BIP 30: Duplicate transactions," https://github.com/bitcoin/bips, 2012.

[24] E. Lombrozo, J. Lau, and P. Wuille, "BIP 141: Segregated witness (consensus layer)," https://github.com/bitcoin/bips, 2015.

[25] G. Andresen, "BIP 11: M-of-N Standard Transactions," https://github.com/bitcoin/bips, 2011.

[26] M. Backes, J. Dreier, S. Kremer, and R. Kunnemann, "A Novel Approach for Reasoning about Liveness in Cryptographic Protocols and Its Application to Fair Exchange," in *IEEE EuroS&P*, 2017, pp. 76–91.

[27] The mathlib Community, "The lean mathematical library," in *ACM CPP*, 2020, pp. 367–381.

[28] J. A. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol: Analysis and Applications," in *EUROCRYPT*, 2015, pp. 281–310.

[29] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd Your Herd of Provers," in *Boogie Workshop*, 2011, pp. 53–64.

[30] L. Lamport, "The temporal logic of actions," *ACM TOPLAS*, vol. 16, no. 3, pp. 872–923, 1994.

[31] T. Wallez, J. Protzenko, and K. Bhargavan, "Comparse: Provably secure formats for cryptographic protocols," in *ACM CCS*, 2023, pp. 564–578.

[32] L. C. Paulson, "The inductive approach to verifying cryptographic protocols," *J. Comput. Secur.*, vol. 6, no. 1–2, pp. 85–128, 1998.

[33] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[34] G. Bella and L. C. Paulson, "Kerberos version IV: Inductive analysis of the secrecy goals," in *ESORICS*, 1998, pp. 361–375.

[35] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT Solvers," *J. Autom. Reason.*, vol. 51, no. 1, pp. 109–128, 2013.

[36] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman, "Strand spaces: Proving security protocols correct," *J. Comput. Secur.*, vol. 7, no. 2–3, pp. 191–230, 1999.

[37] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A Generic Methodology for the Modular Verification of Security Protocol Implementations," in *ACM CCS*, 2023, pp. 1377–1391.

[38] T. Wallez and C. Cremers, "Bob DyLean: A Framework for the Symbolic Analysis of Cryptographic Protocols in Lean," IACR ePrint 2026/1493, 2026.

[39] S. Jeanteur, L. Veronese, M. Solitro, and M. Maffei, "Leandy: Type-based and trace-based symbolic protocol verification in lean," 2026. https://arxiv.org/abs/2607.03406

[40] K. Bhargavan, A. Bichhawat, P. Hosseyni, R. Küsters, K. Pruiksma, G. Schmitz, C. Waldmann, and T. Würtele, "Layered symbolic security analysis in DY*," in *ESORICS*, 2023, pp. 3–21.

[41] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *ACM POPL*, 2001, pp. 104–115.

[42] V. Cheval and B. Blanchet, "Proving More Observational Equivalences with ProVerif," in *POST*, 2013, pp. 226–246.

[43] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More," in *IEEE S&P*, 2022, pp. 69–86.

[44] V. Cheval and C. Fontaine, "Automatic Verification of Finite Variant Property Beyond Convergent Equational Theories," in *IEEE CSF*, 2025, pp. 521–536.

[45] J. Dreier, L. Hirschi, S. Radomirovic, and R. Sasse, "Automated unbounded verification of stateful cryptographic protocols with exclusive or," in *IEEE CSF*, 2018, pp. 359–373.

[46] K. Rupić, L. Rožić, and A. Derek, "Mechanized Formal Model of Bitcoin's Blockchain Validation Procedures," in *FMBC*, 2020, pp. 7:1–7:14.

[47] D. Annenkov, J. B. Nielsen, and B. Spitters, "ConCert: A smart contract certification framework in Coq," in *ACM CPP*, 2020, pp. 215–228.

[48] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek, "Modeling Bitcoin Contracts by Timed Automata," in *FORMATS*, 2014, pp. 7–22.

[49] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, 1996, pp. 232–243.

[50] V. Jacquot and B. Donnet, "CHAUSSETTE: A symbolic verification of Bitcoin scripts," in *ESORICS Workshops (CBT)*, 2023, pp. 359–375.

[51] L. S. Brugger, L. Kovács, A. Petkovic Komel, S. Rain, and M. Rawson, "CheckMate: Automated Game-Theoretic Security Reasoning," in *ACM CCS*, 2023, pp. 1407–1421.

[52] S. Rain, G. Avarikioti, L. Kovács, and M. Maffei, "Towards a Game-Theoretic Security Analysis of Off-Chain Protocols," in *IEEE CSF*, 2023, pp. 107–122.

[53] C. Boyd, K. Gjøsteen, and S. Wu, "A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract," in *FMBC*, 2020, pp. 5:1–5:13.

[54] G. Barthe, U. Dal Lago, G. Malavolta, and I. Rakotonirina, "Tidy: Symbolic Verification of Timed Cryptographic Protocols," in *ACM CCS*, 2022, pp. 263–276.

[55] H. Hüttel and V. Staroveški, "Secrecy and Authenticity Properties of the Lightning Network Protocol," in *ICISSP*, 2020, pp. 119–130.

[56] ——, "Key Agreement in the Lightning Network Protocol," in *ICISSP*, 2022, pp. 139–155.

[57] GLM-5 Team, "GLM-5: From Vibe Coding to Agentic Engineering," arXiv:2602.15763, 2026.
[58] OpenAI, "Introducing GPT-5.5," https://openai.com/index/introducing-gpt-5-5/, 2026.
[59] ——, "GPT-5.6: Frontier intelligence that scales with your ambition," https://openai.com/index/gpt-5-6/, 2026.
[60] Mistral AI, "Leanstral 1.5: Proof Abundance for All," https://mistral.ai/news/leanstral-1-5/, 2026.
[61] Qwen Team, "Qwen3.5: Towards native multimodal agents," https://qwen.ai/blog?id=qwen3.5, 2026.
[62] ——, "Qwen3.6-35B-A3B: Agentic coding power, now open to all," https://qwen.ai/blog?id=qwen3.6-35b-a3b, 2026.
[63] W. A. Howard, "The formulae-as-types notion of construction," 1995.
[64] Mistral AI, "Leanstral: Open-Source foundation for trustworthy vibe-coding," https://mistral.ai/news/leanstral/, 2026.
[65] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A cross-protocol attack on the TLS protocol," in *ACM CCS*, 2012, pp. 62–72.
[66] K. G. Paterson, M. Scarlata, and K. T. Truong, "Three Lessons From Threema: Analysis of a Secure Messenger," in *USENIX Security*, 2023, pp. 1289–1306.
[67] T. Wallez, J. Protzenko, B. Beurdouche, and K. Bhargavan, "TreeSync: Authenticated Group Management for Messaging Layer Security," in *USENIX Security*, 2023, pp. 1217–1233.
[68] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella, "TulaFale: A Security Tool for Web Services," in *FMCO*, 2004, pp. 197–222.
[69] H. Kleisli, "Every Standard Construction is Induced by a Pair of Adjoint Functors," *Proc. Amer. Math. Soc.*, vol. 16, no. 3, pp. 544–546, 1965.
[70] L. de Moura, "The `grind` tactic," The Lean Language Reference, https://lean-lang.org/doc/reference/latest/The--grind--tactic/, 2025.
[71] J. Limperg and A. H. From, "Aesop: White-Box Best-First Proof Search for Lean," in *ACM CPP*, 2023, pp. 253–266.
[72] Lightning Network developers, "BOLT: Basis of Lightning Technology (Lightning Network specifications)," https://github.com/lightning/bolts, 2026.

APPENDIX A
LLM USAGE EXPERIMENT

Proof assistants are particularly enticing tools to use with LLMs. Designed to prevent humans from making mistakes in their proofs, they can offer the perfect protection against LLM hallucination: an LLM-generated proof *must* be correct for the proof assistant to accept it. With this observation in mind, we tested how LLMs behave with LeanDY as an alternative approach to the classical automation explained in App. D. We used (in order of relevance) GLM 5.2 [57], GPT 5.5 [58] and 5.6 Sol [59], Leanstral 1.5 [60] and more marginally Qwen 3.5 397B [61] and 3.6 35B [62]. Opus 5.5 was used for editorial purposes. We attempted to use LLMs to extend a manually written simple payment channel model into our current model, notably bringing in all the complexity of the new scripts and second-stage transactions.

A. AI-Generated Specifications

LLM generation of protocol specifications (i.e., defining exchanges) was very unreliable. The AI often modeled similar looking, but deeply flawed protocols and added unrealistic assumptions, even in the presence of sanity checks in the form of an execution trace of the protocol. The model of the Lightning Network presented in this paper is therefore the result of many iterations and manual interventions. It is unclear to us whether any time was gained by using AI in laying out the specification.

Model	Messages	Input	Output	Time† (hr)	Tool (%)
GLM 5.2 [57]	27K	3.57 Gtk	25.3 Mtk	255.6	16.7
GPT [58], [59]	20K	3.41 Gtk	7.7 Mtk	113.4	32.3
Leanstral 1.5 [60]	12K	1.15 Gtk	4.8 Mtk	30.3	32.3
Qwen [61], [62]	5K	259.8 Mtk	2.72 Mtk	18.4	13.5
Others	149	5.72 Mtk	0.1 Mtk	1.0	47.3
Total‡	63K	8.40 Gtk	40.7 Mtk	418.7	22.7

†: best effort under-approximation;
‡: excluding the other rows' rounding.

TABLE I: LLM usage for modeling the Lightning Network.

B. AI-Generated Proofs

On the other hand, LLMs were impressively good at making proofs. Most of the protocol-specific proofs for the Lightning Network example were eventually written by AI with some guidance by us (but surprisingly little of it). The proofs were mainly made by GLM 5.2 and a bit of Leanstral 1.5, under the supervision of GPT 5.5 or 5.6 Sol to attempt to split proofs into smaller units. Unfortunately, the generated proofs were massive and unidiomatic: the proofs typically used very little automation and looked quite close to the program representing the proof (remember, in Lean, proofs are programs as per the Curry-Howard isomorphism [63]).

Among the models available to us, only state-of-the-art ones (as of 2026) were meaningfully capable of proofs. Notably, the Qwen models or the original Leanstral [64] were incapable of proving anything related to LeanDY. Leanstral 1.5 stands out for its relative usefulness in focused sub-proofs despite its small size; it proved to be a powerful workhorse as a subagent.

Moreover, AI-generated proofs were quite slow to make, potentially taking hours: a single LLM agent (of any model) is typically as fast as, or significantly slower than, a human making the same proof. However, we cannot overstate the massive time gains achieved by parallelization.

C. Metrics

We tracked, to the best of our ability, the per-model usage while generating the Lightning Network model; this includes subagents and failed attempts. Our data is summarized in Table I. One should note that gathering these statistics was quite unreliable due to flaky harnesses and inconsistent provider metrics. Similarly, time tracking is very circumstance-dependent, varying with the load on the provider and the computing power available for Lean type checking. Since we cannot capture metrics on a per-protocol basis, Table I is only a snapshot of our initial proof of the Lightning Network; LLM were used since then for various purposes (e.g., propagating little changes to some definitions), the scale allocated to the Lightning Network does not change.

We immediately notice that, under normal circumstances, LLM automation is not cost-effective. Moreover, without parallelization, using LLMs for proofs is very slow regardless of the provider, backing our subjective claim in App. A-B.

Appendix B
Labels for Conditional Secrecy of XOR: Handling Dynamic Compromise

We can extend our type-based approach of conditional confidentiality to handle dynamic compromise of protocol participants. Specifically, we use the $\mathcal{P}riv^n$ label to represent data that is private to principal n but becomes public upon compromise of n. We then replace the $\mathcal{S}ec$ label in the initial labeling of Sec. II with the corresponding $\mathcal{P}riv$ label for each principal. However, $\mathcal{P}riv$ alone is insufficient to model the secrecy of T_2 and T_3. For instance, when $carol$ is compromised, T_2 becomes public only if either $dave$ is also compromised or the $Close_D$ event has occurred. We capture this dependency using the meet operator ($\sqcap$), which yields a label that is public only when both operands are public. The initial labeling of the values used in the protocol is thus:

$$\begin{aligned} T_1 &: [\mathcal{P}riv^{dave}/\mathcal{P}ub]_{Close_D} \\ T_2 &: [\mathcal{P}riv^{carol} \sqcap \mathcal{L}(T_1)/\mathcal{P}ub]_{Close_C \wedge Close_D} \\ T_3 &: [\mathcal{P}riv^{bob} \sqcap \mathcal{L}(T_2)/\mathcal{P}ub]_{Close_B} \end{aligned}$$

where we use $\mathcal{L}(T_n)$ to refer to the initial label of T_n. Note that the structure of the label matches the chaining of compromises required for the values to be public.

The computed labels follow the same principle as above: the label of an XOR is given by the meet of the labels of its operands.

$$rk_{alice} = T_2 \oplus T_3 : \left[(\mathcal{P}riv^{carol} \sqcap \mathcal{L}(T_1)) \sqcap (\mathcal{P}riv^{bob} \sqcap \mathcal{L}(T_2)) \Big/ \mathcal{P}ub\right]_{Close_B \wedge Close_C \wedge Close_D}$$

Note that the dynamic compromise of principals must be accounted for in the security goal: we either assume that principals remain uncompromised for the duration of the protocol, yielding the proof above, or we explicitly incorporate the possibility of compromise into the security goal:

$$☢_{alice} \vee End \implies \begin{pmatrix} (Close_B \vee ☢_{bob}) \wedge (Close_C \vee ☢_{carol}) \\ \wedge\, (Close_D \vee ☢_{dave}) \end{pmatrix}$$

Here, compromised participants, denoted $☢_p$, share their private knowledge with the adversary. Such an adversary can thus broadcast the release secrets without triggering the honest principals' events. This is reflected in the property, where, for each event, either the event is triggered or the respective principal is compromised. The proof follows the steps of the previous example, with an additional case split explicitly handling the possibility of compromise for each participant.

Appendix C
Parsing and Serializing Bytes

Protocol specifications rely heavily on custom formats to abstract low-level details away. This abstraction does not always reflect reality, yielding the entire class of *format confusion attacks* [65], [66], [67]. There is an ongoing effort to make existing tools aware of such attacks via frameworks like TulaFale [68] or, even more relevant to our context, Comparse [31].

The core of LeanDY is intentionally built around the symbolic bytes datatype introduced in Sec. III-A2. This restriction keeps the implementation simple and limits the soundness arguments of Sec. III-A3 to the constructors of the term algebra. To make this representation practical for protocol specifications, LeanDY provides a generic parsing and serialization infrastructure between user-defined data types and symbolic bytes. Protocol implementations can therefore be written in terms of domain-specific types, such as protocol states or structured messages, while the library core and its proofs only need to reason about the underlying symbolic byte algebra.

Parsing. Parsing and serialization between any two types is performed through the `Parsable` type class.

```
class Parsable (α β: Type) where
  parse: α → Option β
  serialize: β → α
  sound (x:β): parse (serialize x) = some x
notation "do_parse " x " as " γ => Parsable.parse (β:=γ) x
```

where, typically, α is instantiated as $\mathbb{B}$. The `do_parse` notation is used as a user-friendly typing annotation for parsing.

Parsing is used throughout LeanDY, and all relevant types implement it. It is also used to abstract away $\mathbb{B}$ for state management. This genericity enables composing parsing definitions. Notably, `Parsable` is a transitive relation between types:

```
def trans_parsable (γ₁ γ₂ γ₃: Type)
  (h₁: Parsable γ₁ γ₂) (h₂: Parsable γ₂ γ₃): Parsable γ₁ γ₃
where
  parse := h₁.parse >=> h₂.parse -- Kleisli composition [69]
  serialize := h₁.serialize ∘ h₂.serialize
  sound := ...
```

This allows for easily instantiating `Parsable` on complex types and is what some of the automation relies on: we derive a trivial parsing to a nested tuple, and then, by transitivity, derive the `Parsable` instance all the way to $\mathbb{B}$.

Remark 3. *We do not enforce bijective parsing. This enables support for formats where serialization is loosely specified. This can notably matter for blockchain protocols (e.g., Sec. IV), where scripts* can *be written in multiple semantically equivalent ways (e.g., using* `OP_NOP`*s).*

Extending the Type System. We allow the user to extend the notion of *valid public* to more than just $\mathbb{B}$ using the `HasValidPublic` type class. Combined with parsing (using the `ValidParsable` type class), this effectively corresponds to extending the type system with a new constructor.

```
class HasValidPublic (α η γ: Type) where
  is_valid_public (Γ: LabelingEnv α η) (x: γ) (tr: 𝒯 α η): Prop
  is_valid_public_monotone Γ (x: γ): Monotone (is_valid_public Γ x)

class ValidParsable (α η γ δ: Type)
  [HasValidPublic α η γ] [HasValidPublic α η δ] [Parsable γ δ]
where
  valid_parse Γ tr (x:γ) (y:δ) (h: is_valid_public Γ x tr)
    (h': parse x = some y) : is_valid_public Γ y tr
  valid_serialize Γ tr (y:δ)
    (h: is_valid_public Γ y tr) : is_valid_public Γ (serialize y) tr
```

$$\textsc{L-TupleMono}\ \frac{\Gamma \vdash_{tr} a : l \qquad \Gamma \vdash_{tr} b : l}{\Gamma \vdash_{tr} \text{tuple}(a, b) : l} \qquad \textsc{L-Proj1}\ \frac{\Gamma \vdash_{tr} a : \top}{\Gamma \vdash_{tr} \text{proj}_1(a) : \top}$$

$$\textsc{L-Proj2}\ \frac{\Gamma \vdash_{tr} a : \top}{\Gamma \vdash_{tr} \text{proj}_2(a) : \top} \qquad \textsc{L-Verify}\ \frac{\Gamma \vdash_{tr} \sigma : \top}{\Gamma \vdash_{tr} \text{verify}(\sigma, k) : \top}$$

(a) Derived confidentiality labeling rules.

$$\textsc{V-Proj1}\ \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(a)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{proj}_1(a))} \qquad \textsc{V-Proj2}\ \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(a)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{proj}_2(a))}$$

$$\textsc{V-ADec}\ \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(c)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{adec}(c, k))} \qquad \textsc{V-SDec}\ \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(c)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{symdec}(c, k))}$$

$$\textsc{V-Verify}\ \frac{\texttt{valid_bytes}^{\Gamma}_{tr}(\sigma)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\text{verify}(\sigma, k))}$$

(b) Derived validity rules.

Fig. 5: Selected derived rules for destructors and tuples.

Thus, implementing `ValidParsable` effectively adds the following VALIDSERIALIZE rule to the type system.

$$\textsc{ValidSerialize}\ \frac{\texttt{is_valid_public}^{\Gamma}_{tr}(x)}{\texttt{valid_bytes}^{\Gamma}_{tr}(\texttt{serialize}\ x) \qquad \Gamma \vdash_{tr} \texttt{serialize}\ x : \top}$$

Automation. The VALIDSERIALIZE rule is part of the type system's automation, and LeanDY also features macros to automatically derive `Parsable` and `ValidParsable` instances. In particular, all types in Sec. IV-B (states, messages, custom scripts,...) have parsing automatically derived.

APPENDIX D
LEANDY PROOF AUTOMATION: EXCHANGE-SPECIFIC TACTICS

By proving trace-level preservation once and for all via `transition_valid`, LeanDY shifts the entire verification burden to the static properties of individual message exchanges. Proving these exchange-specific invariants—formulated as `valid_exchange`-shaped goals—offers a great starting point for automation. Recall from Eq. (3) the shape of `valid_exchange`. It loosely consists of two parts: type-system proofs and user predicate proofs.

A. Type-System Automation

Type checking is generally a mundane task for computers. LeanDY's type system, however, features arbitrary user-provided predicates (i.e., `CryptoPreds`) and a generally undecidable subtyping relation (due to the trace-dependent label ordering in L-SENC and L-AENC). Aside from these two aspects, the type system does not feature a general subsumption rule, making type checking mostly deterministic.

Leveraging this determinism, LeanDY provides the `prove_*` tactics to prove $\Gamma \vdash_{tr} m : l$ and/or $\texttt{valid_bytes}^{\Gamma}_{tr}(m)$. To make such automation usable in practice, we derive the typing rules for destructors, some of which are transcribed in Fig. 5. We can omit some of the expected premises by leveraging the fact that some terms effectively get simplified away; this is notably the case for the `valid_bytes` rules. We specialized the rule L-TUPLE into L-TUPLEMONO to focus the automation on the most common case of public labels. The decryption rules for labels are not present in Fig. 5 as they involve more complex quantified formulas.

B. Full Automation of Exchanges

Beyond simple type-system automation, we build on the extensible automation machinery of Lean to automate the remaining proof obligations.

To simplify reasoning about control flow, we split execution paths in a manner similar to `StrandsRocq` [21]. We then leverage our `prove_*` tactics alongside Lean's built-in simplifier and automated proof search machinery (e.g., the `grind` tactic [70]) to dispatch straightforward subgoals. Ultimately, this leaves only the non-trivial goals that capture the core cryptographic properties of the protocol.

This culminates in the `intro_valid_exchange` tactic. We showcase its effectiveness by automatically proving the entire `valid_exchange` property for `initiator_check` from Ex. 2. The automation gracefully hands control back to the user for the cryptographically relevant case of `valid_events`, thereby automating away all security-irrelevant details.

```
theorem valid_initiator_check (tr: 𝒯 Unit Nonce) :
      valid_exchange.alt Γ tr (initiator_check) := by
  intro_valid_exchange for initiator_check
  case valid_events sa sb sna _ _ =>
    simp_all
    exists sa, sb, sna.1
    grind
```

C. AESOP-Based Automation

LeanDY also provides a fine-tuned configuration for the AESOP rule-based tactic [71]. This enables a backtracking search through the type system rules and helper lemmas as well as basic first-order reasoning. This approach is significantly more lightweight than the previous one, enabling its use in contexts where the simplifier might otherwise loop (e.g., on extremely large terms or complex user predicates). Due to its higher degree of automation, our `aesop_type_system` tactic can notably make use of the more complex decryption and tuple rules. In the end, it can single-handedly dispatch the proof for the responder role of Ex. 2:

```
theorem valid_responder (a b : ℙ) (tr: 𝒯 Unit Nonce) :
    valid_exchange Γ tr (responder a b) := by
  aesop_type_system
```

Unlike `intro_valid_exchange`, however, it has a less graceful failure mode, rarely offering the user meaningful progress on failure.

APPENDIX E
AUTOMATIC LABELING ENVIRONMENTS WITH XOR

A. Basics

Manually building a labeling environment that follows Eq. (2) may look like a daunting task. A naive but intuitive method is to consider Eq. (4): $x_1 \oplus x_2$ can only be known by those who know both x_1 and x_2.

$$\Gamma(x_1 \oplus x_2) = \Gamma(x_1) \sqcap \Gamma(x_2) \tag{4}$$

where $\sqcap$ is Sec. III-A3's `meet`.

Then, starting from an assignment Γ_0 of labels to atomic values (i.e., terms of the form `.Xor` $\{n\}$ in our setting), we can naturally and easily build a consistent labeling environment from Eq. (4).

```
def from_singles (Γ₀: η → ℓ α η) : LabelingEnv α η where
  Λ s := ⨅_{s₀∈s} Γ₀ s₀
  ...
```

B. Derived Labeling Environment from Any Basis

Looking deeper, it might seem that `from_singles` and Eq. (4) are inadequate. Already a typical use of the one-time pad breaks them: in that setting $m \oplus k$ is expected to be $\mathcal{P}$`ub` while `from_singles` and Eq. (4) would assign it the label $\Gamma(m) \sqcap \Gamma(k)$, meaning $m \oplus k$ could only be published after *both* m and k are published.

Thankfully, we are not constrained to atomic values. Finite sets form a vector space (over the Booleans, with $\triangle$ as the addition), and `from_singles` can be generalized to any basis of that space. This means the user only needs to focus on defining the labels for the most important terms (the basis), letting Lean compute the rest.

```
-- b is an ι-indexed family which forms a basis
def from_basis (b : Basis ι Bool (Finset η))
               (Γ₀ : ι → ℓ α η)
    : LabelingEnv α η where
  Λ s := ⨅_{i∈(b.repr s)} Γ₀ i
  ...
```

where `(b.repr s i : Bool)` is the projection of `s` on the i^{th} vector of `b`; as the scalars of this vector space are Booleans, we use set notation to denote the result of this projection.

Example 4 (Sec. II's labels). *$\{rk_{bob}\}$, $\{rk_{alice}, rk_{bob}\}$, and $\{rk_{alice}, rk_{bob}, rk_{carol}\}$ form a basis of the set of linear combinations of revocation keys. In Sec. II, we assigned them the labels T_3, T_2, and T_1, respectively, and used* `from_basis` *to derive the labels of $\{rk_{alice}\}$ and $\{rk_{carol}\}$.*

C. Esoteric Labeling Environments

We should note that relying on the `meet` is not the only solution to Eq. (2). Consider the following labeling environment.

```
def release_exact (n: Finset η) : ℓ α η where
  corrupt₀ tr := n = ∅ -- the first entry is the message 'n'
                 ∨ tr.getLast? = some (.MsgSent $ ⟦ .Xor n ⟧)
def Γ₀ : LabelingEnv α η where
  Λ := release_exact
```

Then Eq. (4) does not hold in general as, when n and n' are distinct and non-empty, `release_exact` $n \sqcap$ `release_exact` n' is $\bot$ but Γ_0 $(n \triangle n')$ is not.

Such labels and labeling environments are esoteric and unergonomic; therefore, LeanDY only provides automation for the well-behaved cases of App. E-A and E-B, pushing users to rely on labels with a more useful lattice-like structure.

APPENDIX F
DETAILS ON THE LIGHTNING NETWORK MODEL

This section provides further details on our Lightning Network model and the properties proved over it. The formalization closely models blockchain interactions, including scripts and liveness assumptions, while simplifying some aspects of Lightning. In particular, we abstract away routing, effectively modeling a generalized Lightning channel in which in-flight payments are added, fulfilled, or timed out one at a time. Given the size and complexity of the Lightning Network specification [72], a more complete formalization is left as future work. Nevertheless, our model suffices to prove safety properties relevant to intermediaries, including punishment and local atomicity.

A. Concrete Scripts

In this section, we detail the scripts introduced in Sec. IV-B2. The scripts are wrapped in a Pay-to-Witness-Script-Hash (P2WSH) [24] construct formalized in App. F-A1. Thus, the scripts considered here have type `P2WSH LN.Script`, where `LN.Script` is a static-dispatch `inductive` for the `Blockchain.Script` type class from Sec. IV-A2.

```
inductive LN.Script where
-- payment channel scripts (App. F-A2)
| CommonFundS  (s : CommonFund) -- funding output
| ToRemoteS    (s : ToRemote) -- remote commitment output
| ToLocalS     (s : ToLocal) -- delayed / punishable output
-- lightning network scripts (App. F-A3)
| OfferedS  (s : OfferedHTLC) -- htlc to remote
| ReceivedS (s : ReceivedHTLC) -- htlc to local
| ...
```

Fig. 4c provides a high-level flow chart for these scripts.

1) Pay-to-Witness-Script-Hash: Our `Script` type class is expressive enough to encode both native scripts and wrappers such as Pay-to-Witness-Script-Hash (P2WSH) [24]. In the latter case, the witness reveals both redeeming arguments and the underlying script, whose hash must match the hash stored in the spent output.

```
structure P2WSH (S : Type) := (hash : 𝔹⁺ α η)
structure P2WSH.Witness := (args script : 𝔹⁺ α η)

instance [Script α η S] : Script α η (P2WSH S) where
  eval self input := Option.getD (dflt := False) $ do
    let {args, script} ← do_parse input.witness as P2WSH.Witness |
      return False
    guard (hash _ script = self.hash)
    return Script.eval (← do_parse script as S)
      {input with witness := args}
  decidableEval := ...
```

Listing 3: Pay-to-Witness-Script-Hash

where we exploit the `Option` monad. `Option.getD` unwraps the `Option` object, returning `dflt` on failure. Thus, instantiating `Script` for `P2WSH S` amounts to parsing the witness, checking the script hash, parsing the script as an `S`, and evaluating it.

2) Base Payment-Channel Scripts: The first three scripts are inherited from the underlying payment-channel construction.

`CommonFund`. The common fund transaction is guarded by a standard 2-of-2 multisig script [25]. It can be modeled as follows:

```
structure CommonFund := (pka pkb : 𝔹⁺ Unit Nonce)
structure CommonFund.Redeem := (signa signb : 𝔹⁺ Unit Nonce)

open CommonFund in
instance : Script Unit Nonce CommonFund where
  eval self input := Option.getD (dflt:=False) $ do
    let {signa, signb} ← do_parse input.witness as Redeem
    return  verify () signa self.pka = input.stripped ∧
            verify () signb self.pkb = input.stripped
  decidableEval := ...
```

Listing 4: `CommonFund`

`ToRemote`. The *remote* party of a closing transaction is the party that does not store this transaction in their memory (e.g., Bob for $tx^n_{A\to B}$). This party can retrieve their funds unconditionally and immediately from the closing transaction.

```
structure ToRemote := (pk : 𝔹⁺ Unit Nonce)
structure ToRemote.Redeem := (sign : 𝔹⁺ Unit Nonce)

open ToRemote in
instance : Script Unit Nonce ToRemote where
  eval self input := Option.getD (dflt:=False) $ do
    let m ← do_parse input.witness as Redeem
    return verify () m.sign self.pk = input.stripped ∧ ...
  decidableEval := ...
```

Listing 5: `ToRemote`

`ToLocal`. Much of the power of the underlying payment channel comes from the `ToLocal` output. This is the output a party is expected to spend to retrieve their share of the balance when closing unilaterally.

Since unilateral closing can occur without consulting the counterparty, the punishment mechanism must deter closing the channel in an outdated state. Therefore, the output can be redeemed by either party: the closing party must wait for `to_self_delay` blocks, which gives the counterparty time to execute the punishment.

```
structure ToLocal where
  (pk_regular pk_punish hrk_punish: 𝔹⁺ Unit Nonce)
  to_self_delay : ℕ
structure ToLocal.Regular := (sign    : 𝔹⁺ Unit Nonce)
structure ToLocal.Punishment := (sign rk : 𝔹⁺ Unit Nonce)

open ToLocal in
inductive ToLocal.Eval: ToLocal → ScriptInput _ _ → Prop where
| regular {sign input self}
    (hp: (do_parse input.witness as Regular) = some {sign})
    (hsign: verify () sign self.pk_regular = input.stripped)
    (hcsv: checkSequenceVerify input self.to_self_delay)  -- needs
      to wait
      : Eval self input
| punish {sign rk input self}
    (hp: (do_parse input.witness as Punishment) = some {sign, rk})
    (hsign: verify () sign self.pk_punish = input.stripped)
    (hhash: BytesEq.hash () rk = self.hrk_punish)
      : Eval self input

instance : Script Unit Nonce ToLocal where
  eval := ToLocal.Eval
  decidableEval := ...
```

Listing 6: `ToLocal`

In other words, `ToLocal` offers two redemption paths. The regular owner can recover the output with a valid signature only after waiting for `to_self_delay` (the `regular` case), while the counterparty can spend it immediately by providing a valid punishment signature together with a revocation key whose hash matches `hrk_punish` (the `punish` case). This is the core revocation mechanism of Lightning-style channels.

`ToLocal` also showcases that even `inductive` propositions can be used as script definitions.

Remark 4. *The* `Regular` *and* `Punishment` *types are serialized with a discriminant, making the redemption path unambiguous.*

Remark 5. *The inductive's order matches Fig. 4c's branches.*

Rmks. 4 and 5 also apply to the following scripts.

3) Lightning Network Scripts: The HTLC-specific scripts are more involved. The `Offered` script locks coins flowing *from* the closing party, while the `Received` script locks coins flowing *to* the closing party. Punishment is ensured by forcing the closing party to *always* retrieve their coins from `ToLocal` outputs. Lightning therefore uses a two-stage transaction system: 2-of-2 multisig outputs force the closing party to spend its coins through an agreed secondary transaction containing the `ToLocal` output. The following definitions make this pattern explicit.

`Offered`. The output presents three paths: punishment, fulfillment, and timeout:

```
structure OfferedHTLC where
  (revocationPubKeyHash localPubKey
   remotePubKey paymentHash         : 𝔹⁺ Unit Nonce)

namespace OfferedHTLC
structure LocalTimeout := (localSign remoteSign : 𝔹⁺ Unit Nonce)
structure RemoteFulfill := (sign preimage : 𝔹⁺ Unit Nonce)
structure Punishment := (sign rk : 𝔹⁺ Unit Nonce)

inductive Eval : OfferedHTLC → ScriptInput Unit Nonce → Prop where
| localTimeout {input self localSign remoteSign}
    (hp : (do_parse input.witness as LocalTimeout)
      = some {localSign, remoteSign})
    (hs_local : verify () localSign self.localPubKey = input.stripped)
    (hs_remote : verify () remoteSign self.remotePubKey =
       input.stripped)
      : Eval self input
| remoteFulfill {input self sign preimage}
    (hp : (do_parse input.witness as RemoteFulfill)
      = some {sign, preimage})
    (hs : verify () sign self.remotePubKey = input.stripped)
    (hhash : hash () preimage = self.paymentHash)
      : Eval self input
| punish {input self sign rk}
    (hp : (do_parse input.witness as Punishment)
      = some {sign, rk})
    (hs : verify () sign self.remotePubKey = input.stripped)
    (hhash : hash () rk = self.revocationPubKeyHash)
      : Eval self input

instance : Script Unit Nonce OfferedHTLC where
  eval := Eval
  decidableEval := ...
end OfferedHTLC
```

Listing 7: `Offered`

The `punish` branch (Line 23) mirrors `ToLocal.Eval.punish` (Line 14 of Listing 6), while `localTimeout` (Line 11) is

effectively a CommonFund branch. The specification and both agents can therefore constrain the redeeming transaction to the ToLocal shape of Tx. 2. The script itself contains no time constraint; the locktime field of the redeeming transaction enforces the timeout instead.

Received. Finally, Received mirrors Offered:

```
structure ReceivedHTLC where
  (revocationPubKeyHash localPubKey
   remotePubKey paymentHash          : 𝔹⁺ Unit Nonce)
  cltvExpiry : ℕ

namespace ReceivedHTLC
structure LocalFulfill :=
  (localSign remoteSign preimage : 𝔹⁺ Unit Nonce)
structure RemoteTimeout := (sign : 𝔹⁺ Unit Nonce)
structure Punishment  := (sign rk : 𝔹⁺ Unit Nonce)

inductive Eval : ReceivedHTLC → ScriptInput _ _ → Prop where
| localFulfill {input self localSign remoteSign preimage}
    (hp : (do_parse input.witness as LocalFulfill)
      = some {localSign, remoteSign, preimage})
    (hs_local : verify () localSign self.localPubKey = input.stripped)
    (hs_remote : verify () remoteSign self.remotePubKey
      = input.stripped)
    (hhash : hash () preimage = self.paymentHash)
      : Eval self input
| remoteTimeout {input self sign}
    (hp : (do_parse input.witness as RemoteTimeout) = some {sign})
    (hs : verify () sign self.remotePubKey = input.stripped)
    (hcltv : ScriptInput.checkLockTimeVerify input self.cltvExpiry)
      : Eval self input
| punish {input self sign rk}
    (hp : (do_parse input.witness as Punishment)
      = some {sign, rk})
    (hs : verify () sign self.remotePubKey = input.stripped)
    (hhash : hash () rk = self.revocationPubKeyHash)
      : Eval self input

instance : Script Unit Nonce ReceivedHTLC where
  eval := Eval
  decidableEval := ...
end ReceivedHTLC
```

Listing 8: Received

In contrast to the Offered script, the Received script directly enforces the timeout through Line 24.

B. Instantiating LeanDY: States, Labels, Events, and Their Invariants

1) States: Each party keeps track of up to two channel states, each stored in a structure called PcState:

```
structure PcState where
  (self_rk: 𝔹⁺ Unit Nonce)
  (other_rk other_hrk valid_sign_stripped : Option (𝔹⁺ Unit
      Nonce))
  (valid_htlc_signatures : Option (List (𝔹⁺ Unit Nonce)))
  ...
```

where self_rk, other_rk, other_hrk, and valid_sign_stripped are respectively the party's revocation key, the other party's revocation key and revocation hash, and the local transaction's signature by the other party.

The state invariants ensure consistency among all fields: valid_sign_stripped signs the correct transaction, other_rk is a preimage of other_hrk, and signatures are populated only after other_hrk,... A state is *ready* once other_hrk and the signatures are populated.

2) Labels: This protocol features a comparatively simple set of labels, as the variables are either long-term secrets (e.g., keys with private labels) or public. The only exceptions are the revocation keys (for channels or payement hashes), for which we directly use the conditional-swap operator defined in Sec. III-A3:

$$\Gamma(\texttt{Rk a b i}) := \texttt{releasable } (\mathcal{P}\ \texttt{a})\ \mathcal{P}\texttt{ub } \boxed{\texttt{Released a b i}}\ \ldots$$

Thus, a revocation key is initially private and its label becomes public when the corresponding $\boxed{\texttt{Released _ _ _}}$ event is emitted at the revocation step of Fig. 4.

3) Events: Besides simple events marking successful exchanges, which are generally omitted from the paper, we summarize the semantically relevant events of our model.

Old. We mark old channel states with $\boxed{\texttt{Old}_I^k}$ events. Such an event signifies that the agent considers a state old and will actively defend against it. Liveness of the defense is notably defined in terms of these events: the punishment exchange is triggered whenever such an event is in the trace and a transaction appears on chain; the punishment exchange is further parameterized by the $\boxed{\texttt{Old}_I^k}$ event, which provides the trace index of the punishment exchange's state.

Defended. We use a Defended rk tx event to mark successful execution of a punishment with revocation key rk and transaction tx. Although this event is unconstrained, invariants and assumptions use it to relate other aspects of the protocol.

The PostDefended and PostDefendedSecondState post-events instead capture effective punishment. They are added to the trace *after* writing the state and message, simplifying invariants that relate these values. In particular, they enforce that tx is sent to the miners, valid, and of the expected shape.

Signature Events. We trigger an event whenever a transaction is signed, allowing later theorems to recover information about that transaction.

The event $\boxed{\texttt{HtlcTimeoutSigned}^{\texttt{owner,signer}}_{\texttt{commonFund}}\ \texttt{timeout stx}}$ marks the signature of the (stripped) transaction redeeming an Offered timeout output (e.g., Tx. 2). It records the timeout, which the event predicate binds to the transaction:

```
instance up : UserPreds Unit Nonce where
  u_valid_event | Γ, ⟨_, ev⟩, tr => match do_parse ev as Evs with
    | some (HtlcTimeoutSigned^{owner,signer}_{commonFund} timeout stx) =>
      stx.locktime = timeout
    ... -- the rest of the event predicates
  ... -- the rest of the definition of UserPreds
```

Listing 9: HtlcTimeoutSigned invariant

We use this invariant at Line 17 in Listing 12.

4) Signing Invariants: The most important invariants in our model are imposed on signatures via the CryptoPreds type class introduced in Sec. III-A3, which constrains honest agents to sign only valid transactions.[1] These transactions are further restricted to the protocol-relevant forms described in Sec. IV-B2.

```
instance instCryptoPreds : CryptoPreds Unit Nonce where
  u_sign_pred Γ _ m k tr :=
    sign_tx_pred_commitment tr m k ∨ -- commitment transaction
```

[1]At least with the keys reserved for this protocol.

```
4     punishment_tx_pred tr m k ∨ -- ToLocal punishment transaction
5     *_claim_signature ∨ -- honest claiming of outputs ('*'
        stands for each output)
6     ...
7
8  inductive sign_tx_pred_commitment tr m k : Prop  where
9    | intro
10    (ps : ×× ℙ) -- the parties of the channel
11    (self : AOrB) -- the 'role' of the signer
12    ... -- some omitted hypotheses used within inlined proofs
13    (hkey : BytesEq.key () .Verify k = (ps.vk[signer]).val)
14    -- a commitment spends a common fund
15    (cf_ptr_is_cf : cf_ptr.points_to_common_fund ps.vk b ...)
16    (cf_funds_eq : -- the money is consistent
17      (b.get_output cf_ptr cf_ptr_valid).amount
18        = balance.sum
19          + (·.amount <$> offered).sum + (·.amount <$> received).sum)
20    (hevs: SigningEvents ...) -- the events are triggered
21    (def_m : -- the transaction is a commitment transaction
22      m = serialize $
23            (mk_commitment_transaction ps.vk tx_owner cf_ptr balance
24                commitment_hrk offered received).strip b 0 ...)
25
26 inductive tolocal_claim_signature : Prop where
27   | intro
28      ... -- similar to sign_tx_pred_commitment
29      (def_m : m = serialize (tx.strip b j ...))
30      -- we enforce paying to a P2PK to simplify the
        formalization of Th. 3
31      (houtputs : tx.outputs.val = [PayToPubKey.mkOutput sourceAmount
         ps[self]])
32
33 ... -- similar invariants
```

Listing 10: Signing Invariants

Here, `(×× Y)` is a type isomorphic to a pair of `Y`, used to track symmetric channel information such as participants (`ps`), verification keys (`ps.vk`), and balances. The `AOrB` type, with constructors `A` and `B`, selects an agent within such a pair. This makes the model generic over each agent's role: for instance, either can initiate an update or open a channel.

We define the amount owned by an agent as the total value of its unspent P2PK outputs. To simplify Th. 3's statement, we require honest transactions spending their outputs to pay into outputs of that form.

C. Results

Overall, we prove three main properties of our model.

1) Closing: We ensure participants can leave their channel.

Theorem (1—Closing). *Barring closing, each party can construct a valid closing transaction after Groups ❷ and ❹.*

```
1  def isReady : PcState → Prop
2    | {other_hrk:= some _, valid_sign_stripped := some _,
3       valid_htlc_signatures := some _, ..} => True
4    | _ => False
5
6  theorem stored_tx_valid
7    {t : ℕ} s {pc : PcState}
8    -- bind the state to a valid trace
9    (hv : Crypto.valid_trace Γ tr)
10   (hstate : tr.IsStateAt t (.Std s))
11   (hmem : pc ∈ s) (hready : pc.IsReady) -- the state is ready
12   -- the common fund must remain unspent
13   (hnot_spent : s.common_fund ∉ tr.get_blockchain.input_ptrs) :
14   (s.mk_transaction (List.get_blockchain tr) pc).valid
       tr.get_blockchain P2WSH := ...
```

Listing 11: Closing

That is, once a participant acquires the other party's signatures —for example, after Groups ❷ and ❹—they can build a valid closing transaction.

2) Local Atomicity: Despite not modeling routes, we can still verify a key property of Lightning: upon force-closing, an agent learns enough to propagate the HTLCs. That is, force-closing a channel and fulfilling an HTLC reveals the preimage of its payment hash.

Theorem (2—Local Atomicity). *We can publicly retrieve the preimage of the payment hash of any HTLC output spent before its timeout or revocation.*

The proof has four parts: (i) `extract_preimage?` (Line 1) attempts to extract a usable preimage from a transaction; (ii) `extract_preimage?_valid_public` (Line 3) shows that the extracted value is public; and (iii) `extract_preimage?_offeredHTLC` and (iv) `extract_preimage?_receivedHTLC` (Lines 9 and 34) establish correctness for the two HTLC forms. Extraction succeeds and returns the correct preimage when an HTLC output is spent before timeout or revocation.

As usual, these results additionally require trace and blockchain validity and standard honesty assumptions; the Lean statements make these assumptions explicit.

```
1  def extract_preimage?: Transaction Unit Nonce → ℕ → Option (𝔹⁺
       Unit Nonce)
2
3  lemma extract_preimage?_valid_public
4    tx idx
5    (h : HasValidPublic.is_valid_public Γ tx tr)
6    : HasValidPublic.is_valid_public Γ (extract_preimage? tx idx) tr :=
         ...
7
8  /-- preimage extraction is correct on offeredHTLC outputs -/
9  theorem extract_preimage?_offeredHTLC tx idx rk
10   (self : ℙ) (hi : idx < tx.ninputs) (hvb : (List.get_blockchain
       tr).valid _)
11   (hvt : Crypto.valid_trace Γ tr) (hin : tx ∈ List.get_blockchain tr)
12
13   (ohtlc : OfferedHTLCOutput)
14   (oheq : tr.get_blockchain.get_output tx.inputs[idx].outptr =
       ohtlc.toOutput)
15   -- no timeout
16   (ohnotimeout : tx.inclusionHeight tr.get_blockchain <
       ohtlc.cltvExpiry)
17   (ohnotimeout' :
18     ∀ commonFund (owner signer : AOrB) timeout,
19       let stx := (tx.strip (tx.get_sub_blockchain _) idx tr)
20       (HtlcTimeoutSigned^{owner,signer}_{commonFund} timeout stx ∈' tr) →
21         timeout = ohtlc.cltvExpiry)
22   (ohpk : ohtlc.script.localPubKey = vk[self])
23   (hnotcorrupt : TraceEvent.Corrupt self ∉ tr)
24   -- no revocation
25   (ohnotrevoked :
26     ∀ x, ohtlc.script.revocationPubKeyHash = hash () x
27       → ¬is_public Γ x tr)
28   (ohkeys_ne : ohtlc.script.localPubKey ≠ ohtlc.script.remotePubKey)
29   -- preimage extraction succeeds and is correct
30   (oheq₁ : ohtlc.script.paymentHash = hash () rk):
31     extract_preimage? tx idx = some rk := ...
32
33 /-- preimage extraction is correct on receivedHTLC outputs -/
34 theorem extract_preimage?_receivedHTLC tx idx rk
35   (hi : idx < tx.ninputs) (hvb : tr.get_blockchain.valid _)
36   (hvt : Crypto.valid_trace Γ tr) (hin : tx ∈ List.get_blockchain tr)
37
38   (rhtlc : ReceivedHTLCOutput)
39   (rheq : tr.get_blockchain.get_output tx.inputs[idx].outptr =
       rhtlc.toOutput)
40   -- no timeout
41   (rhnotimeout : tx.inclusionHeight tr.get_blockchain <
       rhtlc.script.cltvExpiry)
42   -- no revocation
43   (ohnotrevoked :
44     ∀ x, rhtlc.script.revocationPubKeyHash = hash () x
45       → ¬is_public Γ x tr)
46   -- preimage extraction succeeds and is correct
47   (rheq₁ : rhtlc.script.paymentHash = hash () rk) :
```

```
48  extract_preimage? tx idx = some rk := ...
```

Listing 12: Local Atomicity

The `extract_preimage?_offeredHTLC` theorem is more involved because the `Offered` script does not enforce its timeout directly, instead relying on the 2-of-2 multisig. We instead rely on `HtlcTimeoutSigned` and its invariant introduced in App. F-B3 via Line 17 to bind the `locktime` field of the transaction. This also means that at least one agent needs to be honest.

3) Punishment:

Theorem (3—Safety of Punishment). *Informally, assuming the liveness of both the participants (Group ❻) and the miners, if a revoked commitment transaction is published on the blockchain, then all its coins will eventually belong to the defender.*

Liveness and Honesty. We first formalize our liveness and honesty assumptions. We focus here on the defense of the base `ToLocal` output (i.e., Output 4 in Fig. 4c).

As mentioned in App. F-B3, we trigger punishment on the Old event. This event also records the trace index of the corresponding state. Hence the following selection function:

```
1 inductive DefenseSelector
2   tr : exchange _ _ → ℕ → ℕ → Prop where
3 | intro {t rk}
4     (htr: Blockchain.mined tr)
5     (hnoutputs: ∀ s, noutputs s ≠ 0)
6     (hold: Old {common_fund,i:=t, rk, ...} ∈' tr):
7       DefenseSelector tr punish (Blockchain.get_blockchain_idx tr) t
```

Listing 13: Selection function for Liveness of Punishment

We similarly define selectors for defending the second-stage outputs (i.e., Outputs 5 and 6) and for the claiming exchanges.

Then we gather all our liveness and honesty assumptions into a single predicate.

```
1  structure CompletePunishmentLiveness
2     (trFinal : 𝒯 Unit Nonce)
3     (punishTime claimTime minerTime : ℕ)
4     (ps commonFund)
5     -- a transaction spends the common fund
6     (state : RevokedCommitmentState tr ps commonFund)
7   where
8   -- liveness
9   punishCommitment :
10    ∀ tr₀ trPunish : 𝒯 Unit Nonce,
11      tr₀ ≤ trFinal → trPunish ≤ trFinal →
12        Blockchain.Liveness (DefenseSelector (P2PK.outputs
          state.defender)) punishTime tr₀ trPunish
13  claimToRemote : ... -- similar statements
14  claimOffered : ...
15  claimReceived : ...
16  punishSecondStage : ...
17  miner :
18    ∀ t tx, miner_liveness _ published_to_miners minerTime trFinal t
        tx
19
20  sourceCollision :
21    ∀ (source : Pointer Unit Nonce) rk
22       (tx tx' : Transaction Unit Nonce),
23      (Evs.Defended rk tx ∈' trFinal) →
24      (Evs.Defended rk tx' ∈' trFinal) →
25      tx.spends source → tx'.spends source → tx = tx'
26
27  -- this is not an honest closing
28  defenseActivated :
29    ∀ tr', tr' ≤ trFinal → punish.HOld tr' ps commonFund
30  defenseActivatedSecondStage :
31    ∀ tr', tr' ≤ trFinal → SecondStage.HOld tr' ps
32
```

```
33  ...
```

Listing 14: Liveness Assumptions

The `state` object of Line 5 captures that a transaction spent the `commonFund` and gathers information about it.

The fields beginning at Line 8 collect the participant and miner liveness assumptions.

Line 20 addresses a consequence of LeanDY's composability: a protocol may be composed with another instance of itself. Multiple defending processes could otherwise race to spend the same output. Line 20's assumption requires every defense of a given source (e.g., the common fund in the first-stage defense) to use the same transaction, preventing such races.

Finally, we exclude an honest unilateral close. We ensure this by assuming that the revocation key of any transaction spending the `commonFund` has been tagged with Old. This is captured by the assumptions around Line 27.

Main Statement. Finally, we formalize Th. 3.

```
1  theorem punishment
2    -- the cheating trace and the defending trace
3    (tr trFinal : 𝒯 Unit Nonce)
4    (ps : ×× 𝒫)
5    (commonFund : Pointer Unit Nonce)
6    (punishTime claimTime minerTime : ℕ)
7
8    -- a transaction spends the common fund
9    (state : RevokedCommitmentState tr ps commonFund)
10   -- liveness and honesty of the defense
11   (live : CompletePunishmentLiveness trFinal punishTime claimTime
         minerTime ps commonFund state)
12   -- time budgeting
13   (hgrow : tr ≤ trFinal)
14   (hresponseBudget : claimTime + punishTime + minerTime <
         to_self_delay)
15   (hresponseHorizon : tr.get_blockchain.height + timeout + claimTime +
          punishTime + 2 * minerTime < trFinal.get_blockchain.height)
16   -- validity, corruption, etc.
17   (htraceValid : Crypto.valid_trace Γ trFinal) (hchainValid :
         trFinal.get_blockchain.valid P2WSH)
18   (huncorrupted : .Corrupt ps[state.cheater.other] ∉ trFinal)
19   (hcommonFund : commonFund.points_to_common_fund ps.vk
         tr.get_blockchain commonFund ...)
20    -- the defender does not spend their funds during the
        accounting period
21   (hpayoutStable : did_not_spend state.defender trFinal)
22   : amount_owned_by state.defender trFinal.get_blockchain ≥
         state.tx.output_value := ...
```

Listing 15: Safety of Punishment

The theorem gathers the state and liveness assumptions introduced above in Lines 9 and 11. The bounds at Line 12 require sufficient time for claiming, punishment, and miner response; in particular, `to_self_delay` must leave enough time to react. Line 16 adds the standard LeanDY assumptions of validity and non-corruption. Finally, Line 21 ensures simple accounting by requiring the defender not to spend their funds. Consequently, at `trFinal`, the defender owns at least the full output value of the revoked commitment transaction and has successfully punished the cheater.

Our coin accounting is intentionally coarse: it does not compare holdings before and after publication, and `hpayoutStable` is a strong assumption. The proof uses more general lemmas showing that all `ToLocal` outputs (Output 4 to Output 6) can be spent to any output satisfying the validity constraints. We present this simpler corollary because it is substantially more readable.

APPENDIX G
COMPLETE DEFINITIONS OF VALIDITY AND RESTRICTIONS ON THE DOLEV-YAO ATTACKER

The Dolev-Yao attacker is, by design, maximally powerful within the symbolic model: it controls the network and can compute any term derivable from its knowledge. Some protocols, however, are only correct under environment assumptions that this attacker violates by construction. The Lightning Network of Sec. IV is a case in point: since exchanges act on the blockchain snapshot embedded in the received message (e.g., `common_fund.valid b` in Listing 2), a standard Dolev-Yao attacker could feed an agent a stale or fabricated snapshot, making it act on an outdated view of the chain, trivially breaking the protocol.

LeanDY therefore provides an escape hatch to *weaken* the attacker by adding constraints that even it has to follow. This is achieved through the field `u_restrictions` of `UserPreds`, which specifies a predicate over traces.

```
class UserPreds (α η : Type) extends CryptoPreds α η where
  ... -- previously introduced fields
  -- Defaults to 'True' (standard Dolev-Yao semantics)
  u_restrictions : 𝒯 α η → Prop := λ _ ↦ True
  u_restrictions_antitone : Antitone u_restrictions
```

where `Antitone` is defined in Mathlib [27] and captures that a function is monotonically *decreasing*.

The predicate is integrated into the trace invariant `valid_trace`: the empty trace is valid whenever it satisfies `u_restrictions`, and a trace can be extended by an event only if the extended trace still satisfies it. This yields the formal definition of `valid_trace`:

```
inductive valid_trace : 𝒯 α η → Prop where
| nil (hr : up.u_restrictions []) : valid_trace []
| cons (hr : up.u_restrictions (hd :: tl))
    (hev : valid_trace_event hd (hd :: tl))
    (hrec : valid_trace tl) : valid_trace (hd :: tl)
```

Since new events are *prepended* to the trace, requiring the restriction at every `cons` amounts to requiring that it holds at every point in time, i.e., for every suffix of the final trace. This is what the antitonicity side-condition prepares for: if the restriction holds for a trace, it holds for all of its (older) suffixes, so that `valid_trace` can be destructured along arbitrary trace extensions.

Crucially, the restriction applies uniformly to *all* producers of trace events. Honest exchanges must prove it as part of `valid_exchange`(op): the obligation covers the full trace update of the exchange (its events, message, and saved state). This yields the formalization of Eq. (3):

```
structure ValidExchange
    (msg : 𝔹 α η) state (evs postevs: TraceEvent α η)
where
  valid_bytes  : valid_bytes Γ msg
    (.SetState (serialize state) :: evs ++ tr)
  has_label    : msg.is_public Γ
      (.SetState (serialize state) :: evs ++ tr)
  valid_state  : Crypto.valid_state Γ state (evs ++ tr)
  valid_events : (∀ ev ∈ evs, valid_trace_event Γ ev tr)
  valid_post_events : (∀ ev ∈ postevs,
    valid_trace_event Γ ev
      (.MsgSent msg :: .SetState (serialize state) :: evs
        ++ tr))
  valid_restrictions : up.u_restrictions
    (postevs ++ .MsgSent msg :: .SetState (serialize state)
      :: evs ++ tr)

abbrev valid_exchange (op : exchange α η) :=
  ∀ {msg} {newState} {evs} {postevs},
    (hr: up.u_restrictions tr) →
    (input : 𝔹⁺ α η) →
      (hi₁: valid_bytes Γ input tr) →
      (hi₂: input.is_public Γ tr) →
      (hi_in: .MsgSent input ∈ tr) →
    (state : up.u_state) →
      (hs : valid_state Γ state tr) →
    (hstep : {message:=msg, state:=newState, events:=evs,
      post_events:=postevs} = op tr.length input state) →
      ValidExchange Γ tr msg newState evs postevs
```

The attacker, in turn, is only granted the operations that preserve it, which leads us to amend the definitions of Sec. III-B2 as follows:

```
def Action.valid Γ tr : Action α η → Prop
  | Att (Corrupt p) => up.u_restrictions (.Corrupt p :: tr)
  | Att (Send m) =>
      AttackerKnows Γ tr m ∧ up.u_restrictions (.MsgSent m :: tr)
  | Honest op => valid_exchange Γ tr op
```

Listing 16: Attacker operations under restrictions

That is, the attacker remains maximally powerful *modulo* the restriction: it can still send every message it can derive from its knowledge and corrupt any principal, but only along traces satisfying `u_restrictions`.

A. Soundness under Restrictions

What becomes of the soundness results of the library? Their *statements* are unchanged: each valid exchange still preserves `valid_trace` (Sec. III-B2), and the top-level soundness theorem still guarantees that every trace generated by the scheduler satisfies `valid_trace`. What changes is, first, the meaning of `valid_trace`, which now includes the restrictions, and, second, the attacker model, which is now the restricted Dolev-Yao attacker.

Recall the scheduler of Sec. III-B2, interleaving attacker actions and honest exchanges. Under restrictions, its soundness theorem becomes:

```
theorem soundness [..] Γ (ch: ChoiceFn) (next: Scheduler α η)
                  (limit: ℕ) :
  -- the scheduler only performs valid actions
  (h: ∀ tr, valid_trace Γ tr → (next tr).valid Γ tr) →
  -- the default state is valid on the empty trace
  (hv: up.u_state_has_valid.valid Γ default []) →
  (hresp: ch.responsive) →
  let tr₀ := [.MsgSent default, .SetState (serialize default)]
  -- the seed satisfies the restrictions
  (hr₀: up.u_restrictions tr₀) →
  ∃ tr, run_many next limit ch tr₀ = some ((), tr)
      ∧ valid_trace Γ tr
```

Listing 17: Top-level soundness under restrictions

The theorem reads as in the standard Dolev-Yao setting, with two differences in its assumptions. First, validity of attacker actions (`Action.valid`) now pairs the attacker's knowledge with the preservation of the restriction; for honest exchanges, `valid_exchange.full` already contains the restriction obligation, so validity remains discharged once and for all per exchange. Second, the base case now requires the seed

trace $\mathtt{tr}_0$ to satisfy the restrictions ($\mathtt{hr}_0$); by antitonicity, this also covers its suffixes, including the empty trace. Provided these assumptions hold, every trace reachable by arbitrarily scheduling valid exchanges and attacker operations is valid, and hence satisfies the restrictions at every point in time.

Two remarks are in order. (i) The restrictions affect *neither* `AttackerKnows` nor the type system: type-based guarantees (e.g., Prop. 3) are proved for all traces, hence a fortiori for the restricted ones. The restriction only shrinks the set of *reachable* traces, so only trace-based results (those proved from `valid_trace`, such as chain validity below) depend on it. (ii) Conversely, every security result of a restricted protocol holds against a *weaker* attacker: the restriction is an explicit modeling assumption, part of the trusted base of the case study on par with the liveness assumptions of Sec. IV-B4, and each use of `u_restrictions` must be justified accordingly.

B. Restricting the Lightning Network Attacker

The Lightning Network instantiation of LeanDY uses restrictions to enforce that every blockchain snapshot carried by a message is consistent with the actual chain, and that the chain is valid, at every point in time.

```
structure LocalRestrictions (tr : 𝒯 Unit Nonce) : Prop where
  with_blockchain {T} [NotFailParsable T]
      (b : Blockchain Unit Nonce) (msg : T) (m : 𝔹 Unit Nonce)
      (hin : .MsgSent m ∈ tr)
      (hp : do_parse m as WithBlockchain T = some {blockchain := b,
       msg}) :
    b ≤ tr.get_blockchain
  valid_blockchain : tr.get_blockchain.valid Script

structure Restrictions (tr : 𝒯 Unit Nonce) : Prop where
  holds : ∀ tr' ≤ tr, LocalRestrictions tr'
```

The field `with_blockchain` requires that every blockchain snapshot `b` embedded in a message of the trace is a *prefix* of the actual chain extracted from the trace ($\mathtt{b} \leq \mathtt{tr.get_blockchain}$). Since the chain is reconstructed from the trace and only grows, honest snapshots remain valid prefixes forever, whereas the attacker is prevented from replaying stale or forged snapshots: any `WithBlockchain` message it delivers must still describe a past state of the one canonical chain. The field `valid_blockchain` requires the chain itself to be valid (in the sense of Sec. IV-A2); lifted to every suffix by `holds`, this means that the chain is valid *at every point in time*. In particular, blocks are ordinary messages, so chain extensions (including attacker-published ones) are subject to the restriction: the attacker may only extend the chain with valid blocks, i.e., blocks whose transactions spend spendable pointers with fulfilled scripts and preserved value. Together, the two fields encode the *agreement* and *validity* properties of blockchain consensus as trace restrictions, while the missing *liveness* property is supplied by the explicit assumptions of Sec. IV-B4.

In effect, the attacker models an asynchronous network over an agreed, valid, common chain: it may still reorder, drop, or delay messages (within the liveness assumptions), but it cannot lie about the chain. By the remarks above, all type-based guarantees of the Lightning model hold against the unrestricted attacker, while its trace-based results (closing, local atomicity, punishment; Sec. IV-B5) additionally rely on this restricted environment. We decided against including liveness constraints in `u_restrictions` so that they can be enabled or disabled on a per-property basis.