A Formally Verified Lightning Network

Grzegorz Fabiański[1], Rafał Stefański[1], and Orfeas Stefanos Thyfronitis Litos[2]

[1] University of Warsaw
[2] Imperial College London

Abstract. In this work we use formal verification to prove that the Lightning Network (LN), the most prominent scaling technique for Bitcoin, always safeguards the funds of honest users. We provide a custom implementation of (a simplification of) LN, express the desired security goals and, for the first time, we provide a machine checkable proof that they are upheld under every scenario, all in an integrated fashion. We build our system using the Why3 platform.

1 Introduction

Bitcoin [1] is the oldest and consistently the highest valued blockchain. However, despite its value and prominence, it faces severe scalability issues [2] both in terms of throughput and of latency: it natively supports only up to 7 transactions per second and requires 1 hour from transaction submission to its verification.

To alleviate this limitation, the *Lightning Network* (LN) [3] has been developed. It lifts payments *off-chain*, i.e., it enables parties to securely pay each other without interacting with the blockchain. LN users enjoy latency and throughput limited only by the communication network between the transacting parties, avoid the hefty per-payment on-chain fees, and relieve other blockchain users from having to reach consensus on every single payment.

As of this writing, over $120M are contained in LN. This makes LN a security-critical application. Nevertheless, to the best of the authors' knowledge, its complex design[3] has never been formally verified for crucial properties such as safeguarding honest parties' funds. The current work aims to fill this gap. We employ the Why3 [4] formal verification platform to implement a subset of LN, formally specify the desired security properties and mechanically prove that our implementation adheres to them under all execution paths. Our system consists of 6060 lines of program code and 7420 lines of verification code.

1.1 Overview of LN

The central construct of LN is the *payment channel*. Each channel has exactly two parties, each of which has some coins in it. One party can pay the other via an update of the channel state, which is done collaboratively with its counterparty. Each party is guaranteed that they can *unilaterally* close the channel on-chain

[3] https://github.com/lightning/bolts/blob/master/00-introduction.md

and get their fair share of coins at any time. Channel updates are very cheap: the counterparties just need to exchange a few messages — no interaction with third parties or the blockchain is needed for paying within one's channel.

In slightly more detail, LN works as follows: To initially transfer coins from the blockchain to a new channel, the two parties collaboratively create a "joint account". This involves the two parties generating three Bitcoin transactions, with one submitted on-chain (the *funding transaction*) and the other two stored locally off-chain (the *commitment transactions*). The funding transaction transfers the initial coins to the joint account, whereas the commitment transactions safeguard each party's access to its coins: each party can publish on-chain one of the commitment transactions to move its coins back to the blockchain without needing the cooperation of its counterparty. This ensures that no trust between parties is needed. All in all, when published on-chain, the funding transaction opens the channel and the commitment transaction closes it.

On each payment, two things happen: Firstly a new pair of commitment transactions is generated, reflecting the balance of coins after the payment. Secondly, the previous pair of commitment transactions are *revoked*. A revoked commitment transaction corresponds to an old channel state and thus should never be published on-chain. A malicious party can nevertheless publish it — this could be beneficial if the older balance were in its favor. In such a case, the honest counterparty can publish the corresponding *revocation* transaction (which was generated when the offending commitment transaction was revoked) within a timeframe and punish its counterparty by confiscating all channel coins.

To sum up, a channel needs one on-chain transaction to open, one on-chain transaction to close and can support a practically unlimited number of off-chain payments. It offers low latency, high throughput, no fees and needs no extra trust requirements compared to the underlying blockchain.

The main focus of this work is the security of a single, two-party channel. Channel operations consist of funding a new channel, processing off-chain payments, and closing the channel in a way that the rightful funds are returned to the two parties on-chain. We do not model networks of multiple parties, nor the multi-hop payments supported by production implementations.

1.2 Why3

We implement and specify LN in Why3 [4], a framework for formally verifying high-level code. Why3 focuses on proof automation using SMT solvers. It is largely composed of two parts: Firstly, there is the WhyML language, used to express programs. It also supports expressing formal specifications in first-order logic in the form of inline assertions, separate lemmas, or function pre- and post-conditions; this allows for natural code constructs. Secondly, there is the *driver* which translates the verification conditions of WhyML code into SMT queries. The novelty of Why3 lies in the relatively seamless integration of a variety of SMT solvers. In the current work we have taken advantage of this feature by using both Alt-Ergo [5] and CVC4 [6].

Another important formal verification tool commonly used in the context of cryptographic protocols is EasyCrypt [7]. We note that we chose not to formalize LN using EasyCrypt because the latter is more geared towards low-level, probabilistic reasoning and computational assumptions — features that are used by most cryptographic protocols, but challenging for the general-purpose formal verification tools. In contrast, our approach assumes idealized low-level primitives (i.e., digital signatures) and proves security unconditionally (i.e., without a negligible probability of failure), therefore is largely incompatible with EasyCrypt. In fact, our approach of deterministically modeling signatures obviates the need for EasyCrypt, thereby simplifying our overall proof effort.

1.3 Preliminaries on Cryptography

LN relies on *digital signatures*, which consist of three algorithms: key generation, signing and verification. In our work, we adopt the *ideal functionality* defined in [8] to model these operations. Instead of running the algorithms, parties send queries to and receive outputs from the functionality, which internally keeps track of the signed messages and embodies the single source of truth, obviating the need for randomness and hard-to-model computational assumptions (see Subsec. 4.1). Since the functionality masks the need for such assumptions, we can strengthen our Why3 modeling of the adversary to an unbounded one. In fact, instead of modeling the adversary as a polynomially bounded Turing machine in the context of Why3, the security result proven by our code is against any sequence of choices made by the adversary, even non-computable ones. Although strengthening the adversary in this way might seem counterintuitive, it actually makes the verification effort simpler and clearer. The powers of such an adversary are directly expressed via *universal non-determinism*, which is readily available in Why3. We note that our high-level result (Def. 1) directly carries over to bounded adversaries. Therefore our definition also covers real-world systems which, due to their use of practical constructions of the digital signature algorithms, can only be secure against bounded adversaries.

1.4 Our Contributions

This work consists of the following contributions:

- We provide an implementation of the two-party LN interactions in Why3,
- We implement the subset of Bitcoin logic that is relevant to LN in Why3,
- We define *funds security* for an honest client in the presence of a Byzantine counterparty,
- We provide a machine-checkable proof of the funds security of LN channels,
- We demonstrate how to define *funds ownership* in the presence of multisig accounts and signed transactions stored off-chain,
- We identify crucial properties that any LN implementation should uphold to guarantee funds security and use them to modularize our proof. This provides a blueprint for proving the security of production LN implementations.

Limitations. The main simplifications in our implementation compared to the real Lightning Network (LN) is that it only supports direct payments between two parties. As such it does not support multi-hop payments or HTLCs. Furthermore, we use a simplified model of Bitcoin which, in particular, does not include on-chain fees (see Section 3 for more details).

1.5 Protocol Flow in Our Implementation

Our work focuses on a single, two-party LN channel over Bitcoin. We implement a fragment of the LN functionality, namely direct payments between two parties. We do not include HTLCs and multi-hop payments in our implementation.

The operation of our channel closely follows that of production LN implementations. When one of the two parties (a.k.a. the *funder*) is instructed to open a new channel, the funding process begins. The funder notifies its counterparty (a.k.a. the *fundee*) and the two parties engage in a series of messages, at the end of which the channel is established. All initial funds belong to the funder and they originate from the funder's on-chain bitcoins.

Once a channel has been established, a payment can be initiated by either side. The payer specifies the amount of in-channel funds to transfer, which is subtracted from the payer's channel balance. After a few messages, the payee's balance is increased by the payment sum. Channels support an arbitrary number of sequential payments.

At any moment, either party can initiate the closing procedure. An honest party also closes its channel if its counterparty misbehaves (e.g., stops responding). An honest closing party is guaranteed to receive its channel balance on-chain after bounded delay. Internally, channel closure is initiated by submitting a single transaction on-chain that reflects the channel balance. This starts a timer, during which the counterparty can dispute the claimed channel balance if the closing party submitted an outdated channel state. Honest client operation guarantees that the honest party loses no funds due to failed disputes.

1.6 Security Goals

We consider two standard properties of LN: The first one, which we call *funds security*, states that a party can always close the channel and transfer its rightful funds on-chain within a bounded time frame. For the sake of funds security, we assume that the honest party interacts with a malicious, byzantine party that also controls the environment. The second property called *liveness* states that channel funding and channel payments between two honest parties will always complete in a bounded amount of time. For the sake of liveness, we assume that two parties are honest, but the adversary controls the network and payment requests (see Subsec. 1.7 for more details).

In this work, we only prove funds security, not liveness. In fact, we are aware that in a scenario when the two honest parties try to initiate payments simultaneously in both directions, our implementation might deadlock. (This does not violate the guarantee of funds security, as the deadlock can be resolved by either

party closing the channel). Nonetheless, we do test that a simple channel opening and payment scenario completes successfully (see `twoHonestParties.mlw`). As the violation of funds security is far more detrimental than the violation of liveness, we believe that such an approach strikes a reasonable balance between proof complexity and practical relevance.

1.7 Adversarial Model

In our model, funds security is formalized via a 1-player game, following established cryptographic practice. The game is played by the adversary. Different moves in the game correspond to different actions that the adversary can carry out in real life. For example, it can choose to deliver a message to the ledger, or to have the corrupted channel party send arbitrary messages to its honest counterparty. These two moves correspond to the power to control the network and to corrupt one channel party respectively. Its control over the corrupted party is complete: it can send and sign arbitrary messages on its behalf at any point in time. On the other hand, its control over the network is encumbered with the responsibility to deliver transactions to the ledger within a specific time bound. Furthermore, it controls the moments in which the honest party is activated, but it must ensure that the delay between two activations does not exceed a specific time bound. These limitations correspond to the standard, reasonable network assumptions and the security of LN depends on them.

Last but not least, we give the adversary the power to choose the moment of channel opening and closure, as well as the timings, directions and amounts of payments while the channel is open. These additional adversarial powers correspond to the choices that the honest party can make in a practical deployment. We choose this approach as it models the worst-case scenario, making our results stronger as well as conformant with the standard approach in cryptography. Allowing these choices to be made by the adversary has the added benefit of ensuring that the game is strictly 1-player, keeping the model simpler. The adversary wins if a channel closure request is put forward but, after the aforementioned time bound, the honest party has not received its rightful funds on-chain.

1.8 Architecture Overview

We now discuss our code architecture at a high level — Appx. A contains a relevant diagram. We provide a simplified implementation of Bitcoin called Γ, which focuses on the capabilities related to LN (Sec. 3 and `gamma.mlw`[4]). Furthermore, we implement the signature functionality [8]. It is used by the honest party, the adversary, and Γ to sign messages and verify signatures (Subsec. 1.3 and `signaturesFunctionality.mlw`). Our LN client can read the state of Γ and send transactions to it via a network queue controlled by the adversary. The honest client receives instructions to perform actions, such as to open a channel

[4] The full code can be found at `https://anonymous.4open.science/r/LightingInWhy3-DD1C/` — `README.md` contains relevant documentation.

or carry out a payment. These instructions come from the adversary, but invalid instructions are ignored by the honest client (`honestPartyType.mlw` and `honestPartyInteractions.mlw`). Last but not least, we formally define funds security, as discussed in Sec. 2 (`honestPartyVsAdversary.mlw`).

In order to keep the proof tidy in the face of the complex interactions between the aforementioned components, we use a number of techniques (Sec. 4). We here highlight the two most salient ones. The LN client discussed above has to keep track of a complex state which includes many low-level details, such as which messages remain to be sent in order to conclude an in-flight payment. In order to keep these details separate from the proof, we identify a simple set of requirements on the client behavior (see Subsec. 4.2, Appx. C, and `honestPartyInterface.mlw`). We can thus separate our proof into two independent parts: We first prove that our LN implementation meets these requirements and we then prove that any implementation that satisfies these requirements enjoys funds security. We believe that our requirements must be satisfied by any secure LN implementation, therefore a proof of security of an existing production LN deployment boils down to just proving that it conforms to our requirements.

The second proof technique elucidates the meaning of coin ownership. Albeit a useful high-level concept, is not straightforward to define due to signed but non-processed transactions and multisig accounts. To that end, we define the `Evaluator` (Subsec. 4.4, Appx. B, and module `Evaluator` in `gamma.mlw`) which formally answers the question "How much money does a party own?" This abstraction greatly simplifies the proof effort. We believe this approach can be reused in similar verification projects.

1.9 Related Work

Formal verification of blockchain infrastructure and applications. Existing blockchain-related formal verification efforts revolve around two axes: Verifying consensus protocols and verifying smart contracts.

With respect to consensus protocols, HotStuff [9] has been formally specified and verified by [10] using TLA+ [11]. Furthermore, parts of the Tendermint [12] consensus reference implementation have been formally verified in [13] using TLA+. Similarly to both of these protocol analyses, the current work expresses the execution of multiple parties that communicate via the network, the delays of which are explicitly modeled. Moreover, similarly to [10] and [13], we verify end-to-end guarantees of the execution, not just static invariants.

Regarding the verification of smart contracts, a survey of tools for Ethereum smart contract analysis, including formal verification tools, can be found in [14]. Concrete examples are Manticore [15], EthVer [16], PRISM [17], and [18,19,20]. These tools focus on specific contract properties, whereas our work takes a whole-system approach, whereby the smart contract logic (i.e., the LN transactions) only form one part of the entire system, the other parts being the explicit modeling of parties, the communication network they use, and the process of signing transactions.

Tools and formal verification projects using Why3. A formal verification framework that has seen extensive use in a variety of major projects in the area is Why3 [4]. Some of these projects aim at providing higher-level, special-purpose verification tools. Michelson, the low-level language of smart contracts for the Tezos blockchain [21], can be automatically translated into WhyML with [22].

Security analyses of LN. A survey of results on LN security and attacks against it can be found in [23]. LN security has been formally modeled and proven in the UC setting in [24]. Most comparable to our work is [25], which builds upon [26] in an effort to formally verify the security of LN using TLA+.

Both [24] and [25] model the intricate details of funds security in the multi-hop payment setting, while our work focuses on a single channel. Due to the lack of multi-hop payments, the wormhole and griefing attacks [23] do not apply in our work. With respect to funds security, the main difference between our work and [24,25] is the framework used. Any small differences in the exact security guarantees result from framework differences, not from high-level security goals.

We now present a detailed comparison of our work with [25], starting with the similarities. Both works prove security for a specification of LN, as opposed to a specific implementation. Both approaches model signatures as ideal functionalities, removing their inherent randomness (the latter is problematic in the context of formal verification) in the process. Last but not least, in both cases non-determinism is used to model the adversary, including its exact leeway for arbitrary behavior.

On the other hand, the current work differs from [25] in the following respects: We model security using a game-based definition, proving specific security properties, whereas [25] uses a simplified version of UC-based modeling. We use the Why3 first-order proof system, which allows us to model infinite system states, whereas [25] uses the TLA+ model checker and only checks all possible scenarios up to a specific number of actions. Our modeling is limited to simple payments within a single channel, whereas [25] further models multi-hop payments over multiple channels. We provide a custom implementation of LN, upon which we base our specification of LN, which happens to be more abstract than the official specification (BOLT), whereas [25] verifies the BOLT specification itself. Last but not least, our formal verification effort is complete, whereas [25] constitutes an intermediate report of a proof effort that is still underway.

2 Security Model

The main security goal of this work is to guarantee that a Lightning party that honestly follows the protocol never loses money. In particular, an honest Lightning party that has sent and received a number of payments in the channel should be able to redeem on-chain its initial channel funds plus any funds received minus any funds sent within a bounded time after requesting channel closure. We call this *funds security*. Formally it is a property over 5 parameters:

– The honest party LN protocol `HonestParty`,
– The ledger protocol Γ,
– The party activation window `deltaWake` $\in \mathbb{N}$,
– The ledger delivery window `deltaNet` $\in \mathbb{N}$.
– The time needed for a channel to close `channelClosingTime` $\in \mathbb{N}$.

We define funds security in terms of a game, played between the adversary and the honest party. If the adversary cannot win no matter their actions, then funds security is guaranteed. This game is defined in the `honestPartyVsAdversary.mlw` file.

The game state consists of the following elements: the honest party state, the ledger state, the signature functionality state, and the collection of timestamped, pending ledger messages. It also tracks 4 variables: the last time the honest party was woken (`honestPartyLastWoken` $\in \mathbb{N}$), whether and when was the honest party first ordered to close its channel (`closeOrderTime` $\in \mathbb{N} \cup \{\bot\}$), the current time (`time` $\in \mathbb{N}$), and the expected funds of the honest party (`expectedAmount` $\in \mathbb{N}$).

Since channels are two-party constructions, the other party is presumed corrupted. We define an initial system state in which the ledger contains some coins for each of the two parties and the channel is not yet opened.

We next describe the types of moves that the adversary can make. All actions apart from `IncrementTime` are parametrized by additional arguments that the adversary chooses along with the move type, as explained below.

1. `SignMsg`: This message enables the adversary to sign any number of messages on behalf of the corrupted party. It also models signature malleability by allowing the adversary to create new signatures for messages already signed by the honest party (see Subsec. *Allowing public modification of signatures* in [27]).
2. `SendMsgToGamma`: The adversary generates an arbitrary message which is immediately processed by Γ. This models the ability of a *rushing* adversary to add transactions to the blockchain at any time, "skiping the queue". As we mentioned, Γ is a parameter of the game. Upon receiving a message, it reads the current time, verifies any signatures with the signature functionality, and returns its updated state.
3. `DeliverMsgToGamma`: This action gives the adversary control of the delivery of messages from `HonestParty` to Γ. The adversary chooses a message from the collection of pending messages. The message is removed from the collection and is immediately processed by Γ.
4. `IncrementTime`: This action increases the current system time by 1 unit. This action is only available if (i) there are no pending messages older than `deltaNet` $-$ 1 and (ii) `HonestParty` has been activated within the last `deltaWake` $-$ 1 rounds (i.e., `time` $-$ `honestPartyLastWoken` $<$ `deltaWake`).
5. `SendMsgToParty`: This action lets the adversary interact with `HonestParty` on behalf of the corrupted one. The adversary chooses a message to be delivered to `HonestParty`, which the latter handles according to its implementation. The result of the `HonestParty` activation is accounted for as follows.

The messages that `HonestParty` wants to send to Γ are added to the pending messages collection, `honestPartyLastWoken` is updated to `time`, and `expectedAmount` is updated if `HonestParty` has acknowledged receipt of a payment. During handling of the message, `HonestParty` may interact with the signature functionality. The exact format of messages that `HonestParty` expects depends on the concrete LN party protocol, which as mentioned before is a system parameter.

6. `ControlEnvironment`[5]: This action lets the adversary interact with `HonestParty` on behalf of the system. This action is handled by `HonestParty` in exactly the same way as adversarial instructions (i.e., `SendMsgToParty`) are. The only difference is that `ControlEnvironment` additionally may cause the system state to be directly updated. Here is an exhaustive list of all possible actions the system can prompt `HonestParty` to perform:
   (a) `EnvOpenChannel`: Orders `HonestParty` to initiate opening a channel with the counterparty.
   (b) `TransferOnChain`: Orders `HonestParty` to transfer its funds using an on-chain transaction, i.e., by spending coins from a public-key account. It fulfills two functions: (i) funding of a new channel and (ii) direct on-chain payment to the counterparty[6]. The system then decreases the party's `expectedAmount` accordingly.
   (c) `TransferOnChannel`: Similar to above, but using an already open channel. Once again, the party's `expectedAmount` is decreased.
   (d) `CloseNow`: Orders `HonestParty` to initiate the closing procedure of a previously opened channel. The variable `closeOrderTime` is set to `time`.
   (e) `JustCheckGamma`: Wakes up `HonestParty` to give it the chance to do any recurrent bookkeeping. This is needed to satisfy the periodic `HonestParty` wake-up requirement, even when there is no concrete instruction for the party.

Those actions ensure that the interactions between the adversary and the honest party are modeled accurately. In particular, `expectedAmount` is tracked by the experiment (and not simply reported by the honest party), which guarantees that the value of `expectedAmount` matches its intuitive meaning: it decreases whenever the party is asked to pay the counterparty and increases every time the party acknowledges receiving a payment.

Last but not least, let us define the adversary's winning condition. It is also formally defined as `adversaryWinningState` in `honestPartyVsAdversary.mlw`.

Definition 1 (Funds Security). *The winning state for an adversary is one that satisfies the following:*

1. $\mathit{closeOrderTime} \neq \bot$,

[5] Actually, in the implementation both `SendMsgToParty` and `ControlEnvironment` are handled by `SendMsgToParty`, with appropriate arguments to distinguish between the two. We separate them here for clarity.

[6] Our work only needs the former function of `TransferOnChain` — still, the latter makes our model more complete for essentially no added code complexity.

2. *time* $\geq$ *closeOrderTime* + *channelClosingTime*,
3. *The honest party has less coins than* **expectedAmount** *on-chain (i.e., not counting those in the channel), as output by* Γ *when its function* **immediateAmountOnChain** *is called.*

We say that a system parametrized with the honest party LN protocol **HonestParty**, *the ledger protocol* Γ, **deltaWake**, **deltaNet**, *and* **channelClosingTime** *achieves* funds security *if no adversary can reach the winning state, no matter which sequence of actions it follows.*

Realistic LN implementations use practical constructions of digital signatures, therefore they can only achieve Funds Security against *probabilistic polynomial-time* (PPT) adversaries. As discussed earlier however, we model digital signatures as an ideal functionality, the security of which cannot be broken even by unbounded or non-computable adversaries. In the context of Why3 we model the adversary as unbounded for simplicity, but our ultimate security guarantee is with respect to practical digital signature constructions and therefore against bounded, PPT adversaries. Weakening the adversary from unbounded to PPT strictly shrinks the set of admissible adversaries, therefore the security guarantee obtained by the Why3 model is directly transferable to the bounded adversarial setting.

Let `channelTimelock` be the timelock after which a party can reclaim its funds from a commitment transaction. `channelTimelock` is a parameter of the `HonestParty` protocol. For any `deltaWake`, `deltaNet`, `channelClosingTime` $\in \mathbb{N}$: `channelClosingTime` $\geq 3 \cdot$ `deltaNet` $+ 2 \cdot$ `deltaWake` $+$ `channelTimelock` $+ 1$ and `channelTimelock` $\geq$ `deltaWake` $+$ `deltaNet` $+ 1$, we provide a concrete implementation of Γ that models a subset of the functionality of Bitcoin (Sec. 3), as well as an implementation of a subset of LN (Subsec. 1.5) such that funds security is guaranteed, as we formally verify (the main security result is lemma `honestPartyWins` in `honestPartyVsAdversary.mlw`).

3 Modeling Bitcoin

We use a high-level modeling of the ledger called Γ which captures the fragment of Bitcoin that is relevant to Lightning channels. In this section, we give a brief introduction to Bitcoin operation and then explain how we simplify it in order to express just what is necessary for the protocol.

In practical implementations, Bitcoin transactions are organized in a *directed acyclic graph* (DAG). Each transaction is a node with at least one *input* and one *output*. Each input is connected to exactly one output and each output to at most one input. Each output specifies the number of coins it contains. At any point in time, the DAG has some outputs that are not connected to any input: these are the *unspent transaction outputs* (UTXOs) and model all available coins. A new transaction can be added to the DAG if all its inputs are connected to UTXOs and the sum of the coins of its outputs are equal to the sum of the coins of the outputs that its inputs spend — this guarantees new coins are not created out of

thin air. Furthermore, outputs contain *Script* that specify spending semantics. Script is a simple non-Turing-complete language with a limited expressiveness, including signature verification, hash checking and time checking. Each output locks its coins by specifying a Script statement. It can later be unlocked by providing an input with a corresponding witness. When a valid transaction is added to the DAG, the outputs it spends are not UTXOs anymore and its own outputs become UTXOs.

Our modeling keeps track of the set of transaction outputs, as well as whether they are unspent. It forgoes the DAG and the linearization of the transaction history. We limit our attention to a fixed set of scripts:

- The public key account, known as "pay-to-public-key-hash" (P2PKH) in Bitcoin implementations[7],
- The 2-out-of-2 multisig — this is used to store the coins of an open Lightning channel during normal operation,
- The commitment transaction conditional output — this is used to unilaterally close an open channel.

Bitcoin has no official specification — this role is instead fulfilled by the reference implementation. A formal model for Bitcoin is presented in [28]. In the current work we intentionally avoid modeling parts of Bitcoin execution that are irrelevant to the Lightning participants. More specifically, we ignore all public keys other than those of the two channel parties and any script other than the aforementioned ones. We are confident that our simplified modeling is applicable to a full model of all Bitcoin transitions [28] when non-Lightning transactions are filtered out. We however leave proof of that our model is an abstract interpretation of [28] as future work.

The aforementioned scripts are enough to model all possible Lightning interactions. In particular, each script is used for the following transitions:

- P2PKH allows simple transfers of coins by providing the owner's signature. There are three valid destinations of such transfers: the counterparty (for direct payments), a 2-out-of-2 multisig (for funding the channel), and a special destination for burning coins. The latter is useful for the completeness of our formal model but is not used in the actual Lightning protocol.
- The 2-out-of-2 multisig can be spent in exactly one way, namely by a commitment transaction that has been signed by both counterparties. Each commitment transaction has two outputs: the aforementioned commitment transaction conditional output and a P2PKH of the publisher's counterparty.
- The commitment transaction conditional output can be spent in two ways: by its publisher after a fixed timeout, or at any time by its counterparty if it presents a valid revocation transaction.
- The adversarial counterparty can create arbitrary P2PKH accounts. The adversary is allowed to do this without explicitly spending any output.

[7] Similarly to [27], our modeling foregoes the use of public keys, opting for destination identifiers instead — this simplifies the model by avoiding the explicit formalization of the mechanics of cryptographic signatures.

We refer the reader to our implementation for further details (`gamma.mlw`).

For simplicity, we do not model on-chain fees in our work (this is in line with [28]). However, we believe that extending our work to include fees is conceptually straightforward. See Section 5 for more details.

Note that in the full Bitcoin a 2-out-of-2 multisig can be spent in an arbitrary way as long as both implicated parties agree. Since our security experiment assumes that at least one of the two channel counterparties is honest[8], only the aforementioned way to spend the 2-out-of-2 multisig is possible in our protocol. We deduce that this constraint of 2-out-of-2 multisigs does not artificially restrict the adversarial capabilities.

4 Proof Strategy

LN consists of multiple moving parts that are interconnected in intricate ways. In order to manage the complexity of these interactions, we adhere to the following principle. The system does not need to maintain the entire history of the past states, neither for the honest LN party nor for Γ. All useful knowledge on past states is instead compressed into constant-size invariants. The alternative would be to keep track of the entire execution history, which would be more aligned with human reasoning about distributed systems. Unfortunately, quantification over time complicates formally verified proofs, as it would require a number of interconnected inductive proofs. Our approach is more natural in the context of Hoare logic, as we adhere to a single induction step per protocol step.

4.1 Digital Signatures Ideal Functionality

From a cryptographic perspective, the security of any digital signatures construction is only *computational*, i.e., it can be broken by a sufficiently powerful adversary. As a standard, cryptographic literature only considers PPT adversaries. This is normally necessary because of the underlying computational assumptions, which are breakable by an unbounded adversary. In fact, replacing the aforementioned ideal functionality with a real signature scheme is only secure against such PPT adversaries. Unfortunately, quantifying over all PPT machines has historically proven exceedingly difficult [29] in the context of formal verification. We would thus like to categorically rule out the exponentially small probability of a successful forgery in a principled manner. We achieve this in two steps: Firstly, we allow the adversary in Why3 to be any function (even non-computable) using the `any` keyword of Why3. We then replace the signatures construction with an *idealized* digital signatures functionality (`signaturesFunctionality.mlw`) which never permits forgeries (not even with *negligible* probability), as defined in [27]. At a high level, this functionality plays the role of a "trusted signatures server" that is common for all parties. Parties request the signing and verification of messages. The functionality stores the message-party pair on signature

[8] Assuming that both parties are malicious and no party is honest is arguably not interesting, as security guarantees are only meaningful for honest parties.

request and responds whether the given message-party pair has been stored on verification request. Crucially, the functionality is incorruptible, making forgeries impossible even by a non-computable adversary.

A salient question is: why is the replacement of the construction with the functionality valid? The answer, coming from the *simulation-based cryptographic paradigm*, employs an *indistinguishability argument*, according to which *every* external PPT observer is unable to distinguish between interactions with the construction versus the functionality, except with negligible probability. Indeed, our ultimate security guarantee is provided with respect to a realistic signatures scheme (not an ideal functionality) and a PPT adversary (not an unbounded one). Using a non-computable adversary is merely a proof technique, necessary to tractably model the adversarial behavior in Why3. The indistinguishability argument enables us to transfer the ideal-world security property (formally verified by Why3) to the real world of practical digital signatures and PPT adversaries.

4.2 Encapsulation of Implementation-aware Properties

Our work includes a novel implementation of a subset of the LN mechanism, an implementation of Bitcoin on which our LN implementation relies, as well as a formally verified proof of the *funds security* of our implementation. We chose to organize our proof so as to clearly separate the particularities of our custom LN implementation on the one hand and the general logic that every secure LN implementation must uphold on the other. Our proof is therefore separated into three parts: an implementation-specific part (`honestPartyType.mlw` and `honestPartyImplementation.mlw`) which is aware of the complexities and details of our LN implementation, an abstract specification which encapsulates a set of simpler invariants that any secure LN implementation must uphold (`partyInterface.mlw`), and an implementation-agnostic part of the proof that depends only on the specification (`honestPartyVsAdversary.mlw`).

Our abstract specification focuses on the high-level properties that clients have to satisfy in order to uphold security, such as the need to publish on-chain a known revocation transaction as soon as it becomes valid. This is very different from BOLT, which specifies the required behavior of clients, (e.g., reactions to specific messages), but does not stipulate logical properties. For more details on the interface, see Appx. C or `partyInterface.mlw`.

4.3 `goodSimpleParty`

As our `HonestParty` LN protocol implementation progressed, it became increasingly complex and highly coupled with the rest of the system. Changes to low-level details of `HonestParty` LN protocol would ripple to the invariants and from there to the rest of the system. To mitigate this issue, we decided to introduce a simple, implementation-agnostic invariant that captures the essence of the properties that any secure LN implementation should satisfy (`goodSimpleParty` in `partyInterface.mlw`). This invariant is aware of only a subset of the information held by the `HonestParty` implementation, such as the received revocations

and the commitment transaction that can be used to close the channel. This is the data that any reasonable LN implementation should be able to produce. (It is represented by the type `simplePartyT` in `partyInterface.mlw`.)

Unfortunately, the invariant `goodSimpleParty` cannot be proven directly by induction — one cannot prove that the invariant is preserved by the `HonestParty` transitions without additional dependencies. For this reason we also define a stronger, implementation-aware invariant (`partyInvariant` in `honestPartyType.mlw`) that implies `goodSimpleParty`, and that could be proven directly by induction. This invariant is not visible to the rest of the system and is only used internally in the implementation of the `HonestParty`. To highlight the difference in complexity let us mention that the core logic of `partyInvariant` spans approximately 65 lines of code, whereas that of `goodSimpleParty` is just around 15 lines.

4.4 Funds Ownership

In this subsection we discuss what it means for an `HonestParty` to own coins in Γ. This is a subtle point, since, from the perspective of the chain, all channel funds are locked behind a 2-of-2 multisig, which usually does not imply exclusive ownership. However, an honest channel party should be able to consider some of these funds as its own. As a further example of funds ownership complications, consider an on-chain output controlled by a single public key pk and a signed, valid, published transaction that spends this output but is not (yet) part of the chain. These funds should not be considered as owned by the holder of the private key of pk. Our approach to resolving such ambiguities to ownership is to consider as owned those coins that a party can reliably and unilaterally move to its public key account — this way the tension in both of these scenarios vanishes.

To formalize this concept of ownership, consider a specific state of the Bitcoin ledger. Let an *Extractor* be a party which has some local state (e.g., private keys, signatures received by others) and can communicate with the ledger. We also consider an *Obstructor* with access to the ledger and the ability to take any action of the adversary, as described in Sec. 2. The Extractor has a single ID with respect to the signature functionality, whereas the Obstructor controls all other IDs. The Extractor's goal is to maximize the funds it can extract from the ledger into its private account(s), whereas the Obstructor tries to minimize them. Let *extractable value* be the value of such a game when both parties play optimally. We can now define the value owned by a given party at any instance of the execution of some Bitcoin-based protocol by identifying this party with the Extractor (thus allowing the party to diverge from the protocol) and the rest of the parties (along with the adversary) with the Obstructor. We then say that the funds owned by the party are equal to the extractable value. Observe that this definition of ownership does not depend on any specific protocol thus it can be used in settings other than LN.

Next, we discuss how the concept of ownership is leveraged in our proof of funds security. We say that the `HonestParty` has *solvency* at some point during the execution of the funds security game if the funds it owns are at least equal

to `expectedAmount` (Sec. 2). Solvency has the intuitively appealing property that it refers to any given moment *during* the execution, as opposed to funds security which only concerns the *final state*. It is not hard to see that solvency implies funds security when the channel is closed. More precisely, funds security follows from solvency and the fact that the `HonestParty` can close the channel in bounded time. The proof of the latter fact is relatively straightforward (it is formalized as lemma `closingWorks` in `honestPartyVsAdversary.mlw`). The solvency part of the proof is more involved and it is discussed in Subsec. 4.5.

Finally, let us point out an important technical detail: Although the game-based definition of ownership is, in our opinion, both intuitive and theoretically appealing, it is also challenging to compute and reason about, as it involves evaluating an arbitrary min-max tree. Therefore, in our proof, we replace funds ownership with a more straightforward (i.e., free of min-max trees) but less intuitive `Evaluator` function (implemented as `partyExpectationsFull` in `gamma.mlw`), which provides a pessimistic under-approximation of the funds owned by the party. For further details, see Appx. B. Similarly, we replace solvency with *evaluator solvency*, which states that the `Evaluator` never drops below `expectedAmount` during the execution of the protocol.

4.5 Evaluator Solvency Overview

As discussed in Subsec. 4.4, a central point of our effort is proving the preservation of evaluator solvency across state transitions. In this section we focus on the crux of the proof, which lies in the transitions of Γ and the `HonestParty`.

For the transitions of Γ, it is sufficient to prove that the `Evaluator` does not decrease (as `expectedAmount` does not change during such transitions). This is not hard to prove as the `Evaluator` mirrors the crucial aspects of funds ownership which, by definition, does not decrease during transitions of Γ. (The proof is formalized as `gammaProcessFreshMsgPreservesEvaluatorG` in `gamma.mlw`.)

The transitions of the `HonestParty` are harder to handle for two reasons: Firstly, they may change both the `Evaluator` and the `expectedAmount` (see moves 5 and 6 in Sec. 2). Secondly, the transitions of the `HonestParty` are more involved than the ones of Γ. To help us with the proof, we define a relation `goodTransition` (in `partyInterface.mlw`), which connects the states of the `simplePartyT` before and after the transition, as well as the messages sent by the party to Γ during the transition. This relation abstracts the properties that any LN implementation must guarantee to ensure evaluator solvency. It concerns the obligations of `HonestParty` during a state transition. For example, in case of dispute, the party has to publish a suitable stored revocation to Γ.

The relation `goodTransition` has two desirable properties that make it useful in the proof. Firstly, it is agnostic of the exact implementation of `HonestParty`, which allows us to separate the proof from the implementation details. This is achieved by splitting the proof into two steps: in the first step we prove that any party update that satisfies `goodTransition` preserves evaluator solvency (formalized as `totalEvaluatorMinusTotalBalanceMonotone` in `partyInterface.mlw`), and in the second step we show that the transitions of `HonestParty`

satisfy `goodTransition` (formalized as a postcondition of `partyProcessMsg` in `honestPartyInteractions.mlw`). The other desirable property of `goodTransition` is its transitivity, which allows us to split the second step of the proof into smaller, modular substeps.

5 Conclusion & Future Work

In this work we successfully modeled Lightning [3] channels using first-order logic in Why3 and proved funds security for honest channel parties. First-order logic allowed us to express naturally the security property and functionality of Lightning channels, as well as the low-level invariants encountered during formalization. During the proof effort, we realized that an in-depth understanding of funds ownership was necessary, which led us to define it robustly and formally. We believe that this approach can be reused in similar efforts in the future. Another relevant outcome is the `partyInterface` design, which was the result of a number of modularization attempts. It cleanly separates LN party implementation details with the interface it should provide, simplifying the proof.

There are a number of future directions for strengthening our modelling and architecture. To begin with, we believe that our work could be extended to two-party channels supporting HTLCs with a reasonable amount of effort. This will be a significant stepping stone towards formalizing the security of real-world LN implementations. Moreover, our modeling of Bitcoin can be improved to bring it to parity with already existing formalizations such as [28]. We expect that this task also needs a reasonable effort, but could face more unexpected roadblocks than adding HTLCs.

Another way of improving our Bitcoin model is to include transaction fees, which, for simplicity, we currently do not model. This would require the honest parties to set aside some on-chain funds to pay any fees that may arise. These funds would not be transferable by the `TransferOnChain` environment order. Thanks to the small maximum number of per-channel on-chain transactions (at most 3) and given a (weak) assumption of a maximum per-transaction fee, the amount to set aside is fixed. Thus, funds security could also be proved in the presence of transaction fees.

Last but not least, a more ambitious project would be to extend the security analysis to multi-hop payments over a network of LN channels. To that end, new ideas and insights would be required.

Acknowledgements. This work has been partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 innovation program (grant PROCONTRA-885666). Furthermore, this work was partly supported by the German Research Foundation (DFG) via the DFG CRC 1119 CROSSING (project S7), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

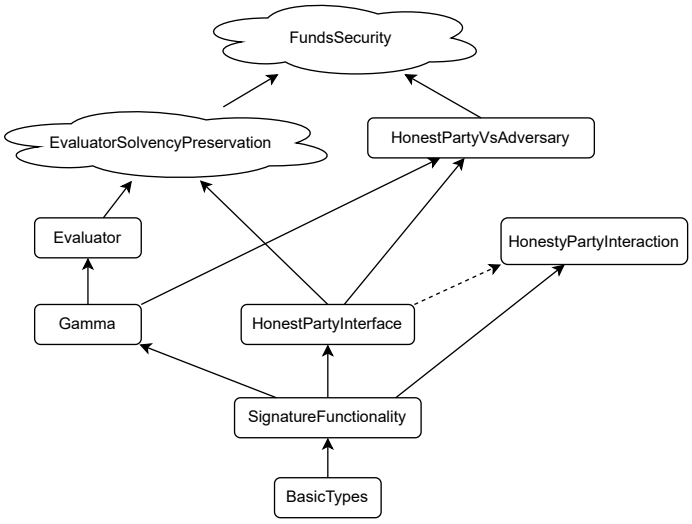
1. Nakamoto S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
2. Croman K., Decker C., Eyal I., Gencer A. E., Juels A., Kosba A., Miller A., Saxena P., Shi E., Sirer E. G., et al.: On scaling decentralized blockchains. In Financial Cryptography and Data Security: pp. 106–125: Springer (2016)
3. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf (2016)
4. Bobot F., Filliâtre J. C., Marché C., Paskevich A.: Why3: Shepherd Your Herd of Provers. In Boogie 2011: First International Workshop on Intermediate Verification Languages: pp. 53–64: Wrocław, Poland: https://hal.inria.fr/hal-00790310 (2011)
5. Conchon S., Coquereau A., Iguernlala M., Mebsout A.: Alt-Ergo 2.2. In SMT Workshop: International Workshop on Satisfiability Modulo Theories: Oxford, United Kingdom: URL https://inria.hal.science/hal-01960203 (2018)
6. Barrett C. W., Conway C. L., Deters M., Hadarean L., Jovanovic D., King T., Reynolds A., Tinelli C.: CVC4. In G. Gopalakrishnan, S. Qadeer (editors), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings: vol. 6806 of *Lecture Notes in Computer Science*: pp. 171–177: Springer: https://doi.org/10.1007/978-3-642-22110-1_14: URL https://doi.org/10.1007/978-3-642-22110-1_14 (2011)
7. Barthe G., Grégoire B., Heraud S., Béguelin S. Z.: Computer-Aided Security Proofs for the Working Cryptographer. In P. Rogaway (editor), Advances in Cryptology – CRYPTO 2011: pp. 71–90: Springer Berlin Heidelberg, Berlin, Heidelberg: ISBN 978-3-642-22792-9 (2011)
8. Canetti R.: Universally composable signature, certification, and authentication. In Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.: pp. 219–233: https://doi.org/10.1109/CSFW.2004.1310743 (2004)
9. Yin M., Malkhi D., Reiter M. K., Gueta G. G., Abraham I.: HotStuff: BFT Consensus with Linearity and Responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing: PODC '19: p. 347–356: Association for Computing Machinery, New York, NY, USA: ISBN 9781450362177: https://doi.org/10.1145/3293611.3331591: URL https://doi.org/10.1145/3293611.3331591 (2019)
10. Kukharenko V., Ziborov K., Sadykov R., Rezin R.: Verification of HotStuff BFT Consensus Protocol with TLA+/TLC in an Industrial Setting. In R. Silhavy (editor), Informatics and Cybernetics in Intelligent Systems: pp. 77–95: Springer International Publishing, Cham: ISBN 978-3-030-77448-6 (2021)
11. Lamport L.: Specifying Concurrent Systems with TLA+. Calculational System Design: pp. 183–247: URL https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/ (1999)
12. Buchman E.: Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Ph.D. thesis: University of Guelph (2016)
13. Braithwaite S., Buchman E., Konnov I., Milosevic Z., Stoilkovska I., Widder J., Zamfir A.: Tendermint Blockchain Synchronization: Formal Specification and Model Checking. In T. Margaria, B. Steffen (editors), Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: pp. 471–488: Springer International Publishing, Cham: ISBN 978-3-030-61362-4 (2020)
14. di Angelo M., Salzer G.: A Survey of Tools for Analyzing Ethereum Smart Contracts. In 2019 IEEE International Conference on Decentralized Applica-

tions and Infrastructures (DAPPCON): pp. 69–78: https://doi.org/10.1109/DAPPCON.2019.00018 (2019)
15. Mossberg M., Manzano F., Hennenfent E., Groce A., Grieco G., Feist J., Brunson T., Dinaburg A.: Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE): pp. 1186–1189: https://doi.org/10.1109/ASE.2019.00133 (2019)
16. Mazurek Ł.: EthVer: Formal Verification of Randomized Ethereum Smart Contracts. In M. Bernhard, A. Bracciali, L. Gudgeon, T. Haines, A. Klages-Mundt, S. Matsuo, D. Perez, M. Sala, S. Werner (editors), Financial Cryptography and Data Security. FC 2021 International Workshops: pp. 364–380: Springer Berlin Heidelberg, Berlin, Heidelberg: ISBN 978-3-662-63958-0 (2021)
17. Kwiatkowska M., Norman G., Parker D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In G. Gopalakrishnan, S. Qadeer (editors), Computer Aided Verification: pp. 585–591: Springer Berlin Heidelberg, Berlin, Heidelberg: ISBN 978-3-642-22110-1 (2011)
18. Yang Z., Dai M., Guo J.: Formal Modeling and Verification of Smart Contracts with Spin. Electronics: vol. 11(19): ISSN 2079-9292: https://doi.org/10.3390/electronics11193091: URL https://www.mdpi.com/2079-9292/11/19/3091 (2022)
19. Park D., Zhang Y., Rosu G.: End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract. In S.K. Lahiri, C. Wang (editors), Computer Aided Verification: pp. 151–164: Springer International Publishing, Cham: ISBN 978-3-030-53288-8 (2020)
20. Hildenbrandt E., Saxena M., Zhu X., Rodrigues N., Daian P., Guth D., Moore B., Zhang Y., Park D., Ştefănescu A., Roşu G.: KEVM: A Complete Semantics of the Ethereum Virtual Machine. In 2018 IEEE 31st Computer Security Foundations Symposium: pp. 204–217: IEEE (2018)
21. Allombert V., Bourgoin M., Tesson J.: Introduction to the Tezos Blockchain. In 2019 International Conference on High Performance Computing & Simulation (HPCS): pp. 1–10: https://doi.org/10.1109/HPCS48598.2019.9188227 (2019)
22. Arrojado da Horta L. P., Santos Reis J., de Sousa S. M., Pereira M.: A tool for proving Michelson Smart Contracts in Why3. In 2020 IEEE International Conference on Blockchain (Blockchain): pp. 409–414: https://doi.org/10.1109/Blockchain50366.2020.00059 (2020)
23. Tian A., Ni P., Liu Y., Huang L.: Blockchain-Based Payment Channel Network: Challenges and Recent Advances. In 2023 International Conference on Blockchain Technology and Information Security (ICBCTIS): pp. 187–194: https://doi.org/10.1109/ICBCTIS59921.2023.00036 (2023)
24. Kiayias A., Litos O. S. T.: A Composable Security Treatment of the Lightning Network. In 2020 IEEE 33rd Computer Security Foundations Symposium (CSF): pp. 334–349: https://doi.org/10.1109/CSF49147.2020.00031 (2020)
25. Grundmann M., Hartenstein H.: Towards a Formal Verification of the Lightning Network with TLA+ (2023)
26. Grundmann M., Hartenstein H.: Verifying Payment Channels with TLA+. In 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC): pp. 1–3: https://doi.org/10.1109/ICBC54727.2022.9805487 (2022)
27. Canetti R.: Universally Composable Signature, Certification, and Authentication. In 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA: p. 219: IEEE Computer Society: https://

doi.org/10.1109/CSFW.2004.24: URL https://doi.ieeecomputersociety.org/10.1109/CSFW.2004.24 (2004)
28. Atzei N., Bartoletti M., Lande S., Zunino R.: A Formal Model of Bitcoin Transactions. In S. Meiklejohn, K. Sako (editors), Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers: vol. 10957 of *Lecture Notes in Computer Science*: pp. 541–560: Springer: https://doi.org/10.1007/978-3-662-58387-6_29: URL https://doi.org/10.1007/978-3-662-58387-6_29 (2018)
29. Liao K., Hammer M. A., Miller A.: ILC: a calculus for composable, computational cryptography. In K.S. McKinley, K. Fisher (editors), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019: pp. 640–654: ACM: https://doi.org/10.1145/3314221.3314607: URL https://doi.org/10.1145/3314221.3314607 (2019)

A System and proof components

In the following picture, we present the key components of our system and proofs. The solid boxes represent parts of the implementation — they are covered in Subsec. 1.8. The two cloud-shaped boxes represent the two main parts of the proof: `EvaluatorSolvencyPreservation` (Subsec. 4.5) and `FundsSecurity` (Sec. 2).



B `Evaluator`

As mentioned in Subsec. 4.4, funds ownership is very complicated to keep track of and even harder to reason about. In particular, it involves calculating the

minimax value that emerges from arbitrary transaction trees with possibly conflicting transactions. We thus opted for a much simpler, pessimistic approximation of the owned funds and implemented it in the `Evaluator` function (namely `partyExpectationsFull` in `gamma.mlw`). Instead of proving solvency, we prove *evaluator solvency*, i.e., that at any point during the funds security game it holds that `expectedAmount` $\geq$ `Evaluator`. (The original concepts of funds ownership and its solvency do not appear in our code.) We then show that, when the channel is closed, evaluator solvency implies funds security (see Lemma `fundsSecurityAux` in `honestPartyVsAdversary.mlw` – the assumption about the channel being closed is derived from the time assumptions combined with Lemma `closingWorks`). Note that this proof strategy matches the one described in Subsec. 4.4, with funds ownership replaced by `Evaluator`.

Let us now elaborate on the difference between funds ownership and `Evaluator`. Interestingly, their main difference lies in the way they treat the simplest Bitcoin outputs, i.e., those locked with a single public key. We will introduce the difference by example. Consider a simple, on-chain output with 10 coins and two competing, fully signed transactions that spend this output. Neither of these transactions is part of the blockchain. The first transaction pays out 1 coin and returns the other 9 to the original public key, whereas the second pays out 2 coins and returns 8 (Fig. 1). To calculate the funds owned by the original public key, we observe that, since only one of these transactions can be added to the chain, the best strategy for the Obstructor is to choose the transaction with the *maximum* payout, which in this example is 2 coins. Therefore the funds owned are equal to $10 - 2 = 8$ coins. In contrast, the `Evaluator` does not take into account the fact that the two transactions are mutually exclusive. It instead calculates the total payout as the *sum* of the two payouts, i.e., $1 + 2 = 3$ coins. Therefore the `Evaluator` outputs $10 - 1 - 2 = 7$ coins[9]. This difference does not affect the `HonestParty` protocol, as the latter never signs two conflicting transactions that spend its public key output.

As another example, consider an on-chain output O_1 locked by a single public key pk with 10 coins and two valid, signed transactions that are not yet on-chain. The first, $\mathtt{tx}_1$, pays out 1 coin from O_1 and returns the remaining 9 coins to O_2, also controlled by pk. The second, $\mathtt{tx}_2$, pays out the remaining 9 coins from O_2 (Fig. 2). In this case, funds ownership and `Evaluator` agree that pk has 0 coins. This example highlights how both funds ownership and the `Evaluator` take into account transactions that spend outputs that are not yet on-chain. Now consider the same scenario but where $\mathtt{tx}_1$ is instead *not* signed (Fig. 3). In this case, both funds ownership and the `Evaluator` ignore $\mathtt{tx}_1$. However, funds ownership takes into account the fact that O_2 is *unreachable*, i.e., it cannot appear on-chain without the missing signature, and decides that 10 coins are owned by pk. On the other hand, the `Evaluator` does *not* take reachability into account. It considers $\mathtt{tx}_2$ as potentially valid and thus outputs $10 - 9 = 1$. Once

[9] The keen reader might here observe that the `Evaluator` may, in the general case, output negative values. We clarify that, even though this runs contrary to intuition, it does not constitute a problem.

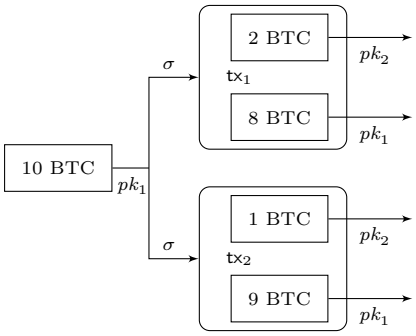


Fig. 1. Conflicting transactions. tx_1 pays 2 coins from pk_1 to pk_2, whereas tx_2 pays 1 coin from pk_1 to pk_2. Since they spend the same output, at most one can be included on-chain. Funds ownership takes into account that the two transactions are mutually exclusive and only considers the maximum payout, 2, concluding that pk_1 owns 8 coins. In contrast, the `Evaluator` considers the sum of the two payouts, $1+2=3$, concluding that pk_1 owns 7 coins.

again, the `HonestParty` protocol is not affected by this discrepancy, since it never signs transactions that spend unreachable outputs that are locked by a single public key.

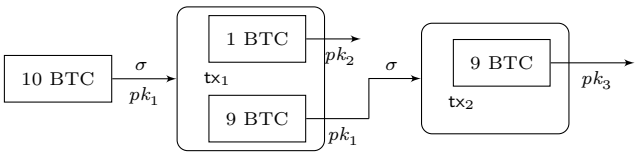


Fig. 2. tx_2 is reachable, since tx_1 is signed. Both funds ownership and the `Evaluator` agree that pk_1 has 0 coins.

As these examples imply, funds ownership is calculated very differently to the `Evaluator` output. In particular, funds ownership is calculated by following all possible paths of signed transactions, whether or not they are on-chain, and choosing the one that corresponds to the minimum coins for the public key in question. On the other hand, the `Evaluator` first calculates the total on-chain coins that belong to the public key and then subtracts the sum of all funds that are paid out to other public keys by pending transactions which are signed by this key. This causes mutually exclusive and invalid transaction paths to be also counted as payouts, but greatly simplifies the proof effort by not considering transaction paths. We note that `HonestParty` and other useful protocols over Bitcoin never sign conflicting transaction paths, therefore in practice funds own-

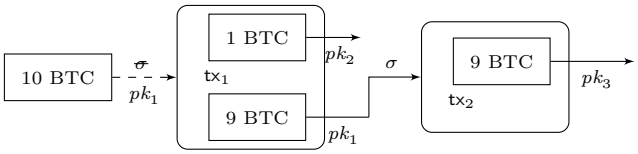


Fig. 3. tx_2 is unreachable, since tx_1 is unsigned. Both definitions do not take the payout of the invalid tx_1 into account. Nevertheless, according to funds ownership, pk_1 has 10 coins, but the `Evaluator` ignores the unreachability of tx_2 and thus decides that pk_1 has $10 - 9 = 1$ coin.

ership and the `Evaluator` output coincide. Importantly, the `Evaluator`-based approach is of independent interest, as it can be used to analyze other Bitcoin protocols, as long as the above considerations are taken into account.

Let us now discuss how the `Evaluator` handles LN-specific outputs and transactions. Contrary to the case of simple on-chain coin transfers, LN requires parties to sign multiple mutually exclusive commitment transactions, all of which spend the same multisig output, i.e., the output of the funding transaction of the channel, which corresponds to the "mutually exclusive transactions" example above (Fig. 1). Furthermore, each revocation transaction that an `HonestParty` signs corresponds to the "unreachable transaction" example above (Fig. 3). Therefore, specifically for LN, the `Evaluator` cannot follow the same aggressive approximation approaches, lest an honest party be usually assigned 0 coins (or less). Instead, during processing of LN-related transactions, the `Evaluator` follows the funds ownership approach more closely.

First, let us discuss how the `Evaluator` treats commitment transaction outputs. There are only two possible ways that such an output can be spent: either by a revocation transaction, or by the public key of the commitment transaction's publisher after a timelock. The `Evaluator` decides which of the two paths are available to the adversary given the signatures signed by the `HonestParty`. This decision process includes checking whether there is still enough time so that revocations are guaranteed to be delivered before the timelock expires[10]. Finally, the `Evaluator` computes the highest value enforceable by the `HonestParty` given the signatures that it has signed and received.

Next, we discuss how the `Evaluator` treats funding transaction outputs. Such outputs can only be spent by a commitment transaction, so in order to calculate their value, the `Evaluator` takes into account all possible commitment transactions available to the adversary (i.e., those signed by the `HonestParty`) and singles out the one that minimizes the `HonestParty`'s payout using the approach of the previous paragraph. Since the adversary may remain idle and not publish any commitment transaction, the `HonestParty` might need to unilaterally close

[10] This includes reasoning both about already sent revocations and about revocations that the `HonestParty` will certainly be able to send in the future thanks to wake-up assumptions.

the channel. The `Evaluator` therefore returns the minimum between the previously discussed best adversarial action and the valuation of the commitment transaction currently stored by the `HonestParty`.

In other words, the definition of the LN-specific part of the `Evaluator` boils down to the following observation: Since the resolution paths of LN have length at most 2, the `Evaluator` only needs to consider transaction trees of depth ≤ 2. This allows us to provide precise reasoning specifically for the case of LN without adding too much generality and code complexity.

It is important to point out that `Evaluator` does not assume that the `HonestParty` satisfies any invariants guaranteed by the `HonestParty` implementation. For example, the `Evaluator` does not depend on the assumption that the commitment transaction stored by the `HonestParty` is always unrevoked. Moreover, the `Evaluator` is aware of only a subset of the information of `HonestParty`, stored in the type `simplePartyT` (Subsec. 4.3): the received revocations and a commitment transaction that can be used to close the channel. This is data that any reasonable LN implementation should be able to produce. In particular, the `Evaluator` does not take into account old commitment transactions that were once stored by the `HonestParty` — they are no longer relevant, as the `HonestParty` has intentionally chosen to (revoke and) forget them.

In the big picture, the `Evaluator` offers a convenient abstraction between the formal properties provided by the low-level details of the Γ semantics, including its state evolution over time, and the high-level guarantees that honest LN parties should enjoy. Without the `Evaluator`, a direct proof would require analyzing the set of reachable states, making it dependent on the implementation of both the `HonestParty` and Γ. Such a proof would quickly become complicated due to the interactions of multiple subsystems. The `Evaluator` prevents these complications by crystallizing all Γ state transitions into a single, relevant property.

C Honest party interface

In this appendix, we describe the *honest party interface* (see Subsecs. 4.2 and 1.8 for context). The interface consists of a type `simplePartyT` (see Subsec. 4.3) and two predicates: `goodSimpleParty` (see Subsec. 4.3) and `goodTransition` (see Subsec. 4.5), which we define below. The definitions presented in this section are sometimes simplified for readability; for the full definitions, refer to `partyInterface.mlw`. We start with the record `simplePartyT`, which contains the following fields:

1. `onChainBalance` of type `amountT`. It represents the amount the party reports that it has in its public key address on the blockchain.
2. `channelBalanceExt` of type `amountT`. It represents the amount the party reports that it has in the channel.
3. `closingChannel` of type `bool`. It is set to `true` if the party is in the process of closing the channel.
4. `channelInfo` which is a record containing the following fields:

(a) `recordOwnerG` of type `partyT`. It represents the party's identity — either `A` or `B`. (We do not keep it directly in `simplePartyT` to avoid redundancy);
(b) `bestSplitReceivedG` of type `option halfSignedSplitT`. It is the commitment transaction that the party will use if it wants to close the channel, equipped with the counterparty's signature. It is equal to `None` if the party is not committed to any channel.
(c) `receivedRevocationsG` of type `revokedSplitsListT`. It is a list of revoked commitment transactions that the party has received. They are also equipped with the counterparty's signatures.

Before discussing `goodSimpleParty`, let us introduce some terminology regarding commitment transactions. When a commitment transaction is submitted to blockchain, one of the parties receives a payout immediately, while the other party can reclaim its coins after a certain timelock, if the other party does not submit a revocation. We call those parties respectively the *unconditional party* and the *conditional party*. Additionally, we call the amount received immediately by the unconditional party the *unconditional amount*, and the amount that can be reclaimed by the conditional party the *conditional amount*.

The predicate `goodSimpleParty` takes a `simplePartyT` along with the state of the signature functionality as inputs, then checks that the following conditions hold:

1. `channelBalanceExt` $\leq$ `balanceOur`. Here `balanceOur` is the conditional amount in `bestSplitReceivedG`, or 0 if it is `None`. Intuitively, it is the amount that the party should be able to extract from the channel.
2. The revocations in `receivedRevocationsG` satisfy the following conditions:
   (a) They are all correctly signed by the counterparty (as reported by the signature functionality);
   (b) The counterparty is the conditional party in all of them.
3. The `bestSplitReceivedG` satisfies the following conditions:
   (a) It has not been revoked (as reported by the signature functionality);
   (b) `recordOwnerG` its conditional party;
   (c) It is correctly signed by the counterparty.
4. Every commitment transaction s signed by the `recordOwnerG` (as reported by the signature functionality) satisfies at least *one* of the following conditions:
   (a) `receivedRevocationsG` contains a revocation for s;
   (b) `recordOwnerG` is the unconditional party in s, and the unconditional amount in s is at least equal to `balanceOur` (see Item 1);
   (c) `recordOwnerG` is the conditional party in s, s has not been revoked by `recordOwner`, and the conditional amount in s at least equal to `balanceOur`.

Next, let us discuss the predicate `goodTransition`. It is a conjunction of four sub-predicates:

1. `goodTransitionForChannel` — contains the properties that guarantee evaluator consistency in the channel;
2. `goodTransitionForClosing` — contains the properties that guarantee the timely closure of the channel (when required);
3. `goodTransitionForChain` — contains the properties that guarantee evaluator consistency on-chain;
4. `goodTransitionAccounting` — contains a few general properties that guarantee the consistency of the party's internal bookkeeping.

In the interest of brevity, we only describe `goodTransitionForChannel` — for the rest of the predicates, please refer to `partyInterface.mlw`. First, let us introduce some more terminology: Observe that a channel is uniquely identified by any of its commitment transactions. We say that a party *tracks* a channel if its `bestSplitReceivedG` contains a commitment transaction for that channel.

The predicate `goodTransitionForChannel` takes as its primary inputs a `simplePartyT` before and after a transition. Additionally, it takes as inputs the system state prior to the transition (including the blockchain state and the signature functionality) and a record describing how the party influences the system state (including messages sent to the blockchain and updates to the signature functionality). It then checks that the following conditions hold:

1. The `recordOwnerG` did not change during the transition.
2. If the party was tracking a channel before the transition, then it is still tracking the same channel after the transition.
3. The party did not forget any revocation from `receivedRevocationsG`.
4. If the party traced a channel in the `DisputeOpen` state, it did not sign any new revocations.
5. If all the following conditions hold, the party sent a valid revocation to the blockchain:
   (a) The channel traced by the party was in the `DisputeOpen` state;
   (b) The party has a revocation to send;
   (c) Messages that are sent now are guaranteed to be delivered to the blockchain before the channel's deadlock is released;
6. One of the following conditions holds:
   (a) The party started tracking a channel in a normal state (this case is used for channel opening);
   (b) The party was tracking a channel in a normal state before and after the transition (this case is used during normal channel operation);
   (c) `balanceOur` (see Item 1 in `goodSimpleParty`) did not change during the transition (this case is used for channel closing).